# INAUGURAL - DISSERTATION

zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Ruprecht-Karls-Universität
Heidelberg

vorgelegt von
M.Sc. Carsten Binnig
aus Buchen

Tag der mündlichen Prüfung: 15.4.2008

# Generating Meaningful Test Databases

Gutachter: Prof. Dr. Donald Kossmann
Prof. Dr. Barbara Paech

# Abstract

Testing is one of the most time-consuming and cost-intensive tasks in software development projects today. A recent report of the NIST [RTI02] estimated the costs for the economy of the Unites States of America caused by software errors in the year 2000 to range from $\$22.2$ to $\$59.5$ billion. Consequently, in the past few years, many techniques and tools have been developed to reduce the high testing costs. Many of these techniques and tools are devoted to automate various testing tasks (e.g., test case generation, test case execution, and test result checking). However, almost no research work has been carried out to automate the testing of database applications (e.g., an E-Shop application) and relational database management systems (DBMSs). The testing of a database application and of a DBMS requires different solutions because the application logic of a database application or of a DBMS strongly depends on the contents of the database (i.e., the database state). Consequently, when testing database applications or DBMSs new problems arise compared to traditional software testing.

This thesis focuses on a specific problem: the *test database generation*. The test database generation is a crucial task in the functional testing of a database application and in the testing of a DBMS (also called test object further on). In order to test a certain behavior of the test object, we need to generate one or more test databases which are adequate for a given set of test cases. Currently, a number of academic and commercial test database generation tools are available. However, most of these generators are general-purpose solutions which create the test databases independently from the test cases that are to be executed on the test object. Hence, the generated test databases often do not comprise the necessary data characteristics to enable the execution of all test cases.

In this thesis we present two innovative techniques (*Reverse Query Processing* and *Symbolic Query Processing*), which tackle this problem for different applications (i.e, the functional testing of database applications and DBMSs). The idea is to let the user specify the constraints on the test database individually for each test case in an explicit way. These constraints are then used directly to generate one or more test databases which exactly meet the needs of the test cases that are to be executed on the test object.

# Zusammenfassung

In heutigen Softwareentwicklungsprojekten ist das Testen eine der kosten- und zeitintensivsten Tätigkeiten. Wie ein aktueller Bericht des NIST [RTI02] zeigt, verursachten Softwarefehler in den USA im Jahr 2000 zwischen $22,2$ und $59,5$ Milliarden Dollar an Kosten. Demzufolge wurden in den letzten Jahren verschiedene Methoden und Werkzeuge entwickelt, um diese hohen Kosten zu reduzieren. Viele dieser Werkzeuge dienen dazu die verschiedenen Testaufgaben (z.B. das Erzeugen von Testfällen, die Ausführung von Testfällen und das Überprüfen der Testergebnisse) zu automatisieren. Jedoch existieren fast keine Forschungsarbeiten zur Testautomatisierung von Datenbankanwendungen (wie z.B. eines E-Shops) oder von relationalen Datenbankmanagementsystemen (DBMS). Hierzu sind neue Lösungen erforderlich, da das Verhalten der zu testenden Anwendung stark vom Inhalt der Datenbank abhängig ist. Folglich ergeben sich für den Test von Datenbankanwendungen oder von Datenbankmanagementsystemen neue Probleme und Herausforderungen im Vergleich zum traditionellen Testen von Anwendungen ohne Datenbank.

Die vorliegende Arbeit diskutiert ein bestimmtes Problem aus diesem Umfeld: Die *Generierung von Testdatenbanken*. Die Generierung von Testdatenbanken ist eine entscheidende Tätigkeit für den erfolgreichen Test einer Datenbankanwendung oder eines Datenbankmanagementsystems (im weiteren Verlauf auch Testobjekt genannt). Um eine bestimmte Funktionalität des Testobjekts zu testen, müssen die Daten in den Testdatenbanken bestimmte Charakteristika aufweisen. Zur Erzeugung einer Testdatenbank existieren verschiedene Forschungsprototypen wie auch kommerzielle Datenbankgeneratoren. Jedoch sind die existierenden Datenbankgeneratoren meist Universallösungen, welche die Testdatenbanken unabhängig von den auszuführenden Testfällen erzeugen. Demzufolge weisen die generierten Testdatenbanken meist nicht die notwendigen Datencharakteristika auf, die zur Ausführung einer bestimmten Menge von Testfällen notwendig sind.

Die vorliegende Doktorarbeit stellt zwei innovative Ansätze vor (*Reverse Query Processing* und *Symbolic Query Processing*), die dieses Problem für unterschiedliche Anwendungen (d.h. für das funktionale Testen von Datenbankanwendungen und Datenbankmanagementsystemen) lösen. Die generelle Idee beider Ansätze ist, dass der Benutzer explizit für jeden Testfall die notwendigen Bedingungen an die Testdaten formulieren kann. Diese Bedingungen werden dann dazu genutzt, um eine oder mehrere Testdatenbanken zu generieren, die die gewünschten Datencharakteristika aufweisen, welche zur Ausführung der Testfälle notwendig sind.

# Acknowledgments

First of all, I would like to express my deep gratitude to Professor Donald Kossmann who supervised and guided my research work in the last three years together with Professor Barbara Paech. From the very beginning Professor Donald Kossmann put trust in me and motivated me to strive for the highest goals. While I was struggling with some hard research problems, he encouraged me to continue and he showed me that I need to step back and see the big picture. Moreover, he spent much time with me to discuss the pros and cons of doing a PhD even before I started to work with him.

Furthermore, I would also like to thank Professor Barbara Paech. Without her help it would not have been possible for me to conduct my research work at the University of Heidelberg. She always put trust in me and gave me all the support that I needed to follow my research interests.

Moreover, there are a lot of other people who supported me during the last three years:

In the first place, I would like to thank Eric Lo. Together with him, I spent a vast amount of time behind closed doors in productive discussions about our crazy research ideas. We constantly motivated each other which finally made our both dreams come true.

Alike, I would also like to thank José A. Blakeley who supported me in applying my research ideas to an industrial environment at Microsoft Corporation in Redmond. Moreover, he introduced me to many other nice and supportive colleagues who assisted me in realizing my research ideas for Microsoft in a very short time.

Furthermore, life would not have been as much fun without all the other colleagues and students at the University of Heidelberg and at the ETH Zurich. Especially, I would like to mention my colleague Peter Fischer in Zurich who helped me in many technical and administrative issues in a completely unselfish way. Alike, I would like to thank two students, Nico Leidecker and Daniil Nekrassov, who helped me in implementing some of my concepts.

Finally, I would like to thank my wife Daniela and my family for their enduring support while I was working on my research ideas in the past few years.

# Contents

# Part I

# Preliminaries

# Chapter 1

# Introduction

*I think and think for months and years, ninety-nine times, the conclusion is false.*
*The hundredth time I am right.*

*– Albert Einstein, 1879-1955 –*

## 1.1 Motivation

Testing is one of the most time-consuming and cost-intensive tasks in the software development projects today. A recent report of the NIST [RTI02] estimated the costs for the economy of the Unites States of America caused by software errors in the year 2000 to range from $\$22.2$ to $\$59.5$ billion (or about 0.6 percent of the gross domestic product). While one half of these costs result from error avoidance and mitigation activities of the users, the other half is borne by software developers due to inadequate testing techniques and tools. Another study [Erl00] in the E-Business sector stated that roughly 90 percent of the total software costs are spent on system maintenance and evolution which includes development costs to identify and correct software defects.

Consequently, in the past few years many techniques and tools have been developed by industry and academia to reduce the costs caused by software errors and the time spent for testing activities. Many of these techniques and tools are devoted to automate various testing tasks (e.g., the test case generation, the test case execution, and the test result checking). According to [Bal06], the worldwide market for Automated Software Quality Tools was about $948 million in 2005 and will be higher than $1 billion in 2006, and $1.8 billion in 2010.

However, almost no research work has been carried out to automate the testing of database applications and relational database management systems (DBMSs). The testing of a database application or of a DBMS needs different solutions because the application logic of a database application or of a DBMS strongly depends on the content of the database (database state)[1]. Consequently, when testing database applications or DBMSs, new problems and opportunities arise compared to traditional software testing. For example, [HKL07] showed that traditional test case scheduling techniques in the test execution phase do not work optimally for database applications. Moreover, [HKL07] illustrated that specialized scheduling strategies can reduce the total running time of the test execution phase dramatically.

This thesis focuses on another specific problem: the *test database generation*. The test database generation is a crucial task in the functional testing of a database application or a DBMS (called test object further on). In order to test a certain behavior of the test object, we need to generate a database state that satisfies certain data characteristics.

A simple example is a *login* function of an E-Shop like Amazon which rejects users to log in after having tried to log in more than three times with an incorrect password. In order to test that function thoroughly, the test database should comprise a user who has not yet tried to log in wrongly more than three times to test the positive case where the user is not rejected. Moreover, the test database should also comprise another user who has already entered an incorrect password more than three times, to test the negative case where the user is rejected by the *login* function.

Another example is the testing of a DBMS. Most of the functionality of a DBMS strongly depends on the data characteristics of the stored data; e.g., the optimizer of a query execution engine chooses the physical execution plan depending on the data characteristics of the underlying database and the data characteristics of the intermediate query results. If we want to test the functionality of the query optimizer thoroughly, it is necessary to vary the data characteristics which are used to calculate the costs of the alternative query plans.

Currently, a number of academic and commercial tools are available which generate test databases. These tools can be classified into two categories: either the test database is extracted from a live database, or a synthetic test database is generated according to a given database schema. The existing tools which extract the test database from a live database suffer from various problems: One problem is that using a live database may not always be possible, because of data protection regulations; another problem is that a live database often does not comprise all the necessary data characteristics to enable the execution of all interesting test cases (e.g., there is no user in the live database who has tried to log in more than three times with the incorrect password).

Consequently, generating a synthetic test database seems to be the panacea to solve these problems. However, existing tools which generate synthetic test databases suffer from the same problem; i.e., the generated test databases do not comprise all the data characteristics necessary to execute a given set of test cases. The reason is that the existing tools are general-purpose solutions which take constraints on the complete database state as input (e.g., table sizes and value distributions

[1]In this thesis we use the terms *database state*, *database instance*, and *test database* as synonyms.

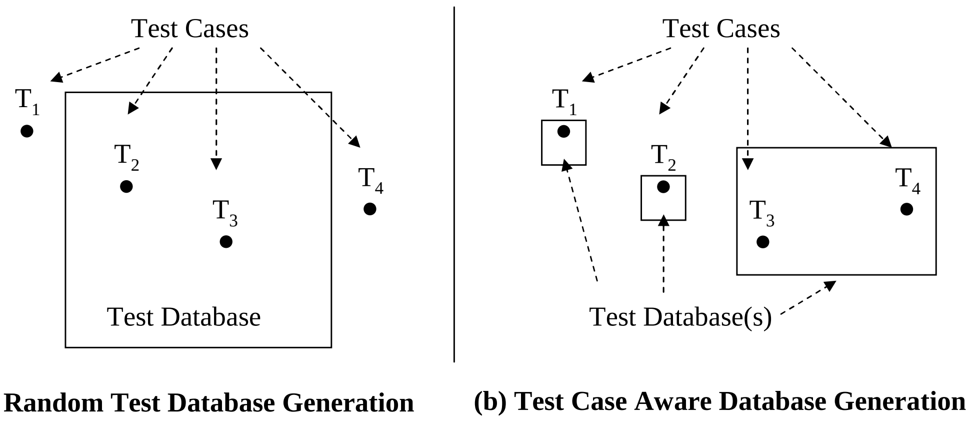

Test Cases
T1
T2
T3
T4
Test Database
(a) Random Test Database Generation
Test Cases
T1
T2
T3
T4
Test Database(s)
(b) Test Case Aware Database Generation

Figure 1.1: Test Database Generation Problem

of individual attributes). However, these constraints are not suitable to express the relevant data characteristics necessary to execute each individual test case.

Consequently, the test databases are usually generated independently from the needs of the individual test cases. We call these approaches which generate the test database independently from the test cases *Random Test Database Generation* techniques. Conceptually, this problem is demonstrated in Figure 1.1 (a): This figure shows a test database which is generated independently from a set of given test cases $\{T_1, T_2, T_3, T_4\}$. Using this randomly generated test database, only some test cases (e.g., $T_2$ and $T_3$) can be executed, while the other test cases (e.g., $T_1$ and $T_4$) cannot be executed at all.

In order to deal with this problem in practice, the generated test databases are often modified manually in order to fit to the needs of all test cases. As a result, the maintenance of a test database becomes hard because a manual modification of the test database for a new or a modified test case often corrupts the test database for other test cases that are to be executed on the same test database.

In this thesis we present two innovative techniques which tackle the *test database generation* problem in a different way by enabling a *Test Case Aware Database Generation*; i.e., one or more test databases are generated which exactly fit to the needs of the test cases that are to be executed on the test object (as shown in Figure 1.1 (b), the generated test databases enable the execution of all test cases).

The main idea of the *Test Case Aware Database Generation* is to let the user specify constraints on the database state individually for each test case in an explicit way. These constraints are then used directly to generate one or more test databases which exactly satisfy the constraints specified by the test cases that are to be executed on the test object.

For example, when testing a database application, it is necessary to formulate constraints on the values of individual tuples (and not on the complete database state); e.g., to test the *login* function of the E-Shop application, the tester needs to specify that two different users exist in the test database (i.e., one who is allowed to log in and one who is not allowed to log in). Alike, when testing a DBMS, it is important that the data characteristics of the intermediate query results of a test query, and not only the characteristics of the base tables, can be controlled explicitly in order to test a particular behavior of the DBMS.

## 1.2 Contributions and Overview

The main contributions of this thesis are the formal concepts and prototypical implementations of two new test database generation frameworks (called *Reverse Query Processing* [BKL06b; BKL06a; BKL07b] and *Symbolic Query Processing* [BKL07a; BKLO07]) which generate *test case aware databases* for the functional testing of OLAP applications (e.g., a reporting application) and for the functional testing of DBMS components (e.g., the cardinality estimation component). Both frameworks are extensible and thus not bound to a specific application, even though they are motivated by the particular applications mentioned above.

As a further contribution, we discuss two more applications of Reverse Query Processing in detail; i.e., the functional testing of OLTP applications (like an E-Shop) [BKL08] as well as the functional testing of a query language [BKLSB08]. Moreover, we also present the required extensions of RQP to support these two applications. Furthermore, we show how the extensions of RQP for the functional testing of a query language can be used in an industrial environment. Finally, we sketch some other applications of Reverse Query Processing which need additional research.

For both frameworks we carried out a set of experiments to analyze the performance and the effectiveness of our prototypical implementations under a variety of workloads.

The remainder of the thesis is structured as follows:

- In the next chapter, we present the background in software testing and illustrate the state of the art in the testing of database applications and DBMSs as well as the state of the art in test database generation.

- In Part II of this thesis, we discuss the first framework which enables a test case aware database generation (called Reverse Query Processing or RQP for short). The main application of RQP is the functional testing of OLAP applications.

- In Part III we illustrate two further applications of RQP in detail (i.e., the functional testing of OLTP applications as well as the functional testing of a query language) and discuss the extensions of RQP which are necessary to support these applications. Moreover, we sketch other potential applications of RQP.

- Subsequently, in Part IV we describe the second framework which enables a test case aware database generation (called Symbolic Query Processing or SQP for short). SQP is designed to generate test databases for the functional testing of individual DBMS components.
- Finally, Part V contains the conclusions (i.e., the current state of this work and its limitations) as well as suggestions for future work (i.e., research problems and potential improvements for a better industrial applicability).

A more detailed discussion of the individual contributions and a detailed outline will be given separately for each part.

Chapter 2

# Background

*Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information upon it.*

*– Samuel Johnson, 1709-1784 –*

## 2.1 Software Testing: Overview and Definitions

*Software Testing* is the execution of a component or a system using combinations of input and state to reveal defects [Bin99] by verifying the actual output. The component or system under test is called the *test object*. Depending on the *test level* the test object is of different granularity: In *Unit Testing* the test object is usually a method or a class, in *Integration Testing* the test object is the interface among several units, and in *System Testing* the test object is a complete integrated application.

In software testing the terminology is very often not clear. In this thesis we refer to the terminology defined in [IST06]. Especially, the terms failure, defect, and error are often used as synonyms while a having a different meaning: A *failure* is the inability of a test object to perform a function within certain limits; i.e., the system or component returns an incorrect output, terminates abnormally, or does not terminate within certain time constraints. A *defect* is the missing or incorrect code that caused the failure of the test object. An *error* is the human action that produced a defect (e.g., by coding). Testing can only show the presence of defects in a test object but never their absence.

Software testing activities can have different *testing objectives*: One possible objective is to reveal defects through failures (which is called *fault-directed* testing). Another objective is to demonstrate the conformance of a test object to the required capabilities (which is called *conformance-directed* testing). A conformance-directed test type is *functional testing* which checks the conformance of the test object with the specification of the functionality. Another conformance-directed test type is usability testing which checks the conformance of the user interface to some usability guidelines or performance testing which checks the conformance to some time restrictions etc.

A *test case* usually specifies the test object, the inputs that are used to exercise the test object as well as the state of the test object before the test case is executed (called *precondition*); e.g., external files, contents of the database. Moreover, a test case defines the expected output and the expected state of the test object after the test case execution (called *postcondition*). The expected output and the postcondition are often called *test oracle*. A *test suite* is a set of test cases that are related; e.g., by a common test objective.

A *test run* is the execution of a test suite which comprises a set of test cases. A test case is executed as follows: First, in a *set-up* phase the precondition is used to the set the state of the test object. Afterwards, the *test object is exercised* using the given input. Finally, the actual output and the state after the execution of the test case are compared to the expected output and the expected state (postcondition) in order to *verify the test results* and decide whether a test case passes or not.

Executing all possible test cases (which is called *exhaustive testing*) by using all combinations of input values and preconditions to execute the test object is practically impossible because the number of all combinations of input values and preconditions is usually too huge. For, example assume that we want to test a method that takes ten $8$-bit integer values as input. The possible input space would be $(2^8)^{10}$. If we could execute 1000 test cases per second then it would take approximately $41.2^{10}$ days which is roughly $38334786263782$ years to run all test cases for exhaustive testing.

In order to deal with that problem various *test design techniques* can be used to derive and select the test cases that shall be exercised on a test object. *Black-box* test design techniques are based on an external view of the test object, and not on its implementation; i.e., black-box test design techniques analyze the external behavior to derive test cases. One example of a black-box test design technique are equivalence classes. Equivalence classes partition the domain of the individual input values into sub-domains for which the behavior of the test object is assumed to be the same. The idea of equivalence classes is that the tester can pick one value out of each class instead using all possible input values.

In contrast to black-box test design techniques, *white-box* test design techniques are based on the internal structure of a test object which can be the result of a source code analysis. One example of a white-box test design technique is *control-flow* testing which uses the information about the control-flow to execute different code paths of the test object. While white-box test design techniques are more often used for fault-directed testing (e.g., to find a division by zero), black-box test design techniques are more often used for conformance-directed testing (e.g, to check whether the test object behaves as specified or not).

The *coverage* of a test suite is measured by a *coverage metric*. A typical coverage metric for white-box testing is the *statement coverage* which defines the percentage of executable statements that have been exercised by a test suite. A coverage metric for black-box testing is the *equivalence class coverage*. The equivalence class coverage is defined as ratio of the number of tested equivalence classes and the number of all equivalence classes; i.e., the metric shows the percentage of equivalence classes that have been exercised by a particular test suite [IST06].

The goal of *test automation* is to minimize the manual overhead necessary to execute certain test activities; e.g., test case generation, test case execution and test result checking. In most cases test automation can be seen as a system engineering problem to implement a special kind of software which executes the test activity automatically.

Test automation has several benefits compared to manual testing. For example, the automation of the test case execution helps to run more test cases in a certain time span and thus to increase the test coverage. Moreover, test automation makes testing repeatable because humans tend to vary the test cases during manual execution. Thus, automating the test case execution is a precondition for effective *regression testing* while the goal of regression testing is to execute one or more test suites after the test object has changed and compare its behavior before and after the changes.

## 2.2 State of the Art

This section gives an overview of the state of the art related to this thesis: First, we discuss some general problems that arise when testing a database application or a DBMS and show several solutions to these problems. Afterwards, we study the problem of generating test databases in Section 2.2.2 in detail. While some of these approaches deal with a similar problem statement as this thesis (i.e., the generation of test case aware databases), some other approaches discuss orthogonal aspects (i.e., efficient algorithms for generating huge data sets or algorithms for generating various data distributions).

### 2.2.1 Testing Database Applications and DBMSs

In the past years many techniques and tools have been developed by academia and industry to automate the different testing activities [Bin96]. Surprisingly, relatively little attention has been given to developing systematic techniques to support the individual testing tasks for database applications and DBMSs [Kap04]. In the following, we discuss specific problems and opportunities that arise when testing database applications and DBMSs. Moreover, we briefly illustrate some of the existing work.

**Test Database Generation:** Before a test suite of a database application or a DBMS can be executed, it is necessary to create an initial database state that is appropriate for each test case. For example, as mentioned in the introduction, in order to execute a test case of an E-Shop application which executes the *login* function, different types of users need to be created.

Currently, some industrial tools (e.g., [IBM; DTM; dbM]) and research prototypes (e.g., [BC05; SP04; HTW06; NML93; CDF$^+$04; MR89; ZXC01; WE06]) are available which generate test databases. However, most of these tools generate the test databases independent from the test cases that are to be executed on the test database. Consequently, in many cases the generated test databases are not appropriate to execute all intended test cases. The reason is that the existing tools are general-purpose solutions which offer only very limited capabilities to constrain the generated test database (i.e., most tools take only the database schema as input and generate random data over that schema). However, these constraints are not adequate to express the needs of the individual test cases which should to be executed on the database application or the DBMS.

Consequently, as the test databases are generated independent from the test cases there has also been no work on the evolution of the test database if the test suite changes (e.g., new test cases are added or existing test cases are modified). Currently, the only way to deal with the evolution of a test suite is to regenerate the test database completely.

As this thesis focuses on the problem of generating test case aware databases, we present some of the existing tools in more detail separately in the next Section 2.2.2.

**Test Case Generation:** In order to generate test cases for database applications and DBMSs, new test design techniques need to be developed because existing techniques cannot deal with the semantics of database applications and DBMSs.

When testing a DBMS, for example, a test case usually comprises one or more SQL queries that are issued against the test database. Traditional test design techniques like equivalence classes are hardly applicable to automatically create test cases (i.e., SQL queries) for DBMS systems and the huge domain of possible SQL queries. Thus, different tools like RAGS [Slu98] and QGEN [PS04] have been developed to quickly generate SQL queries that cover interesting query classes for a given database schema and other input values (e.g., a parse tree, statistical profiles). In order to extend a given test suite with further interesting test cases, [TCdlR07] devises some mutation operators for SQL queries to generate new test queries from a given set of test queries.

Another work [BGHS07] (which was used to test the SQL Server 2005) presents a genetic approach to create a set of test queries (i.e., a test suite) for DBMS testing. The initial set of test queries is created randomly; e.g., by using approaches like RAGS and QGEN. Moreover, each time before a test suite is executed, a new test query is generated by mutating the queries of the existing test suite. Afterwards, the queries of test suite and the new query are executed on the DBMS and execution feedback (e.g., query results, query plan, traces that expose internal DBMS state) is collected. Based on the execution feedback a fitness function determines whether a newly created test case will be added to the test suite or not. For example, the fitness function could use existing code coverage metrics to decide whether the new test query increases the coverage of the test suite or not.

When testing database applications instead of DBMSs, a test suite should cover test cases that exercise the different execution paths of the application. However, standard test design techniques do not work properly because they do not consider the database state when test cases are created. As a result, the test cases may not cover all interesting execution paths. For example, when testing the *login* function of an E-Shop application then not all interesting test cases might be created (e.g., to test different users where one user has already tried to log in more than three times with the incorrect password and another user has not).

In [yCC99] the authors argue that existing white-box test design techniques generate test cases that do not cover all interesting code paths because the semantics of SQL statements that are embedded in a database application are rarely considered. Thus, the authors suggest to transform the declarative SQL statements into imperative code and then to apply existing white-box test design techniques to create test cases. The objective of the transformation is to include the semantics of the SQL statements into the imperative code so that more test cases are generated to reveal defects that result from different internal database states.

For example, a function of an E-Shop application that displays the books of a particular author could use a 2-way join query on authors and books that is embedded in the code to extract the necessary data from the database. In order to test that function, the 2-way join is transformed into a nested loop statement in the application code. Using the transformed code as input, white-box design techniques will generate test cases that cover different database states: one test case could execute the function for an author with no books which means that the nested loop is not executed at all, and another test case could execute that function for an author with $n$ books which means that the nested loop is executed $n$ times.

Another drawback of many existing test design techniques is that they create test cases which do not specify the database state before and after the execution of a test case (i.e., as pre- and postconditions). Consequently, the test cases cannot be used to set the database state before the execution and to check the database state after the execution.

The work in [RDA04] presents a framework for the black-box testing of database applications called AutoDBT. AutoDBT takes a specification of the user navigation (as a finite state machine), a data specification which defines constraints on the database for each transition in the user navigation, as well as an initial database state as input and generates test cases that can be executed on the given database state. Using the data specification, AutoDBT can track the database changes of each test case (i.e., AutoDBT can calculate a set of pre- and postconditions on the database state).

Consequently, AutoDBT can decide whether the precondition of a test case holds (i.e., if the test case can be executed on the current database state) and whether the postcondition is satisfied when the test case was executed (i.e., if the database is in the expected state). For example, for a test case which deletes a given book of an E-Shop, AutoDBT will check (1) if the book exists before the test case is executed and (2) if the book was deleted successfully after the test case was executed.

**Coverage metrics:** As we discussed in Section 2.1 the *coverage* of a test suite is measured by a *coverage metric*. However, existing test coverage metrics cannot deal with the semantics of database applications and DBMSs.

In order to tackle that shortcoming, [KS03] proposed a new family of coverage metrics for the white-box testing of database applications which capture the interactions of a database application with a database at multiple levels of granularity (attribute, tuple, relation, database) . The test coverage metric uses the dataflow information that is associated with the different entities in a relational database (i.e., how many percent of the attributes, tuples, or relations are read or updated by a given test suite). The empirical study in [KS03] confirms that a significant number of important database interactions are overlooked by traditional coverage metrics.

Another work [CT03] proposed a new coverage metric for testing SQL statements. The idea is to apply an existing coverage metric, the multiple condition coverage [MS77], to SQL statements. This coverage metric analysis if a given predicate is evaluated thoroughly in all possible ways for a given test database. For a SQL query, the metric in [CT03] analysis if the join and selection predicates of a given SQL query are evaluated to true and false for the different tuples in the test database. If a predicate is a complex predicate with conjunctions and disjunctions then the coverage metric analysis each simple predicate. For example, if a SQL query contains the predicate $b\_aid = a\_id \wedge a\_name = `Knuth`$ then the coverage metric checks if the complete predicate evaluates to true and false for different tuples of the test database and if each simple predicate ($b\_aid = a\_id$ and $a\_name = `Knuth`$) does so, too. Based on that information the value of the coverage metric is calculated.

**Test Case Execution:** When executing test cases for database applications and DBMSs a particular database state has to be reset before each test case can be executed in order to guarantee a deterministic behavior of the test object. For example, assume that we want to execute a test case $T_1$ of an E-Shop application that lists all books of a particular author (who has written 100 books) and the test cases passes if all 100 books are displayed. However, if another test case $T_2$ (which deletes all books of that author) is executed before test case $T_1$, then $T_1$ will fail because no books are displayed (i.e., the expected result is different from the actual result). Thus, a trivial solution to avoid this problem is to set the appropriate database state each time before the test case is executed. However, this can take very long if the test database is huge: e.g., it already takes about two minutes to reset a $100$MB database [HKK05]. Moreover, traditional execution strategies do not consider that fact when scheduling the test cases for a test run. Consequently, if a test suite should be used for nightly regression tests, then not all test cases might be executed because of the unexpected long running time.

Consequently, the authors in [HKL07] devised several scheduling algorithms which try to find an optimal order of the test cases in a test suite with the goal to minimize number of database resets. This work assumes that all test cases of a test suite can use the same database state. Consequently, if no test case of a test suite updates the test database, then the database state has to be reset only once at the beginning of a test run.

Thus, the basic idea of the algorithms in [HKL07] is to apply the database reset lazily; i.e., a test case of a test suite is executed without setting the appropriate database state. If the test case execution fails, then the database state is reset and the test case is re-executed. If the test case passes afterwards, then the test case has a conflict with a previously executed test case which updated the database state. Otherwise, the test case detected a failure. During a test run (i.e., the execution of a test suite), the algorithms learn which test cases are possibly conflicting with each other (i.e., which test case might have updated the database state so that another test case fails). As a result, the scheduling algorithms reorder the conflicting test cases of a test suite for the next test run with the goal to reduce the number of necessary database resets.

For example, the following test suite is executed in the given order: $T = \{T_1, T_2, T_3\}$ . Assume that only $T_3$ fails because of a “wrong” database state (which was caused by an update of $T_1$ and/or $T_2$). Then for the next test run, the scheduling algorithms would reorder the test suite to avoid the conflict of the test case $T_3$ with test cases $T_1$ and/or $T_2$. A new order could be $T = \{T_3, T_1, T_2\}$.

**Test Result Verification/Test Oracle:** When executing a test case on a database application or a DBMS, the actual test results (actual output and state of the test database) have to be verified in order to decide whether a test case passes or fails.

In the regression testing of database applications, the expected output of a test suite is created by executing the test cases on the test object and recording the behavior of the test object [HKK05] (called recording phase). During the recording phase, the test object is expected to work correctly. After modifications of the database application, the test suite is re-executed (called playback phase) and the actual results are compared to the expected (recorded) results. While regression testing needs a running application to create the test oracle, other test techniques derive the expected results from specifications of the test object (e.g., as discussed in [RDA04]).

Another idea for verifying the result when testing the query processing engine of a DBMS is illustrated in [Slu98]. In order to verify the actual results of the test queries, the author of [Slu98] propose to execute the test queries on a comparable DBMS which returns the expected results for verification. This idea can be generalized and used for the test result verification of other kinds of test objects (not only DBMSs), too.

### 2.2.2 Generating Test Databases

In this section we present several test database generation tools. These tools can be classified into two categories: either a synthetic test database is generated or the test database is extracted from a live database.

**Synthetic Test Databases:** Currently, there are a number of commercial tools available (e.g., [IBM; DTM; dbM]) which generate a random test database over a given database schema. Beside the database schema, some tools also support the input of the table sizes, data repositories and additional constraints used for data instantiation (e.g. statistical distributions, value ranges). Most of the commercial tools are not extensible (e.g., the set of supported data distributions is fixed).

Additionally, a number of academic tools are available which generate test databases. Some of them are designed to be extensible in a few aspects. For example, [BC05] tackles the problem that most existing tools support only a fixed set of data distributions. However, in order to thoroughly evaluate new DBMS techniques (e.g., new access methods, histograms, and optimization strategies) varying data distributions need to be generated. Consequently, this work presents a flexible framework to specify and generate test databases using rich data distributions as well as intra- and inter-table correlations for a given database schema. The framework is based on composable iterators that generate data values, whereas the set of iterators can be extended by the user.

[SP04] presents another test database generation framework (called MUDD) that can also be extended by complex user defined data distributions. MUDD was designed to generate test databases for the TPC-DS benchmark [TPCa] and thus is intended for use in the performance evaluation of DBMS decision support solutions. MUDD also supports varying database schemes.

Moreover, [HTW06] also developed a database generator which is also intended to be used in the performance evaluation of DBMS decision support solutions. Again, the user can easily add new data types and distributions. In addition, their tool takes a graph model and some data dependencies as input: The graph model specifies the database schema and thus defines the order how the tables are populated. The data dependencies (e.g., foreign-key constraints) further constrain the database state.

A different tool which takes a set of user defined predicates as input to generate the test database is presented in [NML93]. The tool supports a subset of the first-order-logic and thus allows the definition of more complex constraints as the tools discussed before which only take the database schema and some data distributions as input. However, [NML93] showed that their approach to generate a test database which meets a set of arbitrary constraints formulated in first-order-logic does not scale for large test databases and complex constraints.

All the tools discussed before generate test databases independent of the test cases that are to be executed on the test object. The tools that we illustrate in the sequel try to tackle this problem by taking some information about the test cases as input (i.e., the application queries of the database application or the test queries that are to be executed on the DBMS).

In [CDF$^+$04] a set of tools for testing database applications (called AGENDA) is presented. One tool of AGENDA is a database generator which takes a database schema (with integrity constraints), an application query and some sample values as input. The selection predicate of the application query is used to partition the domains of all attributes that participate in the predicate into equivalence classes. For example, if a SQL query defines a filter predicate $10 \leq b\_price \leq 100$ then three partitions are generated for the attribute $b\_price$: $]-\infty, 10[$, $[10, 100]$, and $]100, \infty[$. For other attributes not in the selection predicate, the user can define the equivalence classes manually. The database generator offers different heuristics to guide the test database generation process: one heuristic is to generate boundary values for the specified equivalence classes; another heuristic is to generate NULL values if possible, etc.

Another work which also takes a SQL query as input is presented in [MR89]. The goal of this work is to generate a test database for a given relational query (limited to simple select-project-join queries) so that the query result is unique for the given test query; i.e., no other non-equivalent query exists that returns the same result for the generated test database. A test database which satisfies this criteria can be used for the testing of a query language; i.e., for such a test database it is easier to decide whether the actual result of a test query is the expected result or not because the expected result can be returned for only one particular test query (i.e., two non-equivalent queries must have different expected results).

In [ZXC01] the authors study the generation of test databases for the white-box testing of database applications. The goal of this work is to generate a test database which returns a result that has certain characteristics for a given SQL query in order to execute a particular code path of the application. The tool supports only select-project-join queries as input and the user can specify that the result of such a query should be empty or not and she can also add domain constraints on the result attributes (e.g., all values of the $b\_price$ attribute should be greater $0$). In order to generate the test database, all the constraints on the query result are translated into a constraint satisfaction problem which can be solved by existing constraint solvers.

A similar tool is presented in [WE06]. The only difference to [ZXC01] is that the constraint formula which is used to generate the test database for a given database schema is constructed more systematically; i.e., the SQL query is translated into a relational algebra expression and the query operators transform the constraints on the query result into constraints on the database schema. For example, a projection operator adds the deleted attributes to the constraint formula to let the constraint solver instantiate values for those attributes. Again, only select-project-join queries are supported as input.

In contrast to all tools discussed before, [GSE+94] focuses on particular problems that arise when huge synthetic databases need to be generated. This work is orthogonal to all tools presented above. In particular, this work discusses how parallelism can be used to get generation speed-up and scale-up and presents algorithms to generate huge data sets that follow various distributions (e.g., uniform, exponential, normal). Moreover, solutions to generate indexes concurrent to the base table are discussed, too.

**Extracts from Live Databases:** Another alternative to generate test databases is to extract the data from a live database. However, extracting data from a live database might be problematic because the use of data from live databases has the potential to expose sensitive data to an unauthorized person. Moreover, a live database may not cover all interesting data characteristics adequate to test a particular behavior of the test object.

In [WSWZ05], the authors investigate a method to generate a so called *mock database* based on some a-priori knowledge about the live database without revealing any confidential information of the live database. The techniques of this work guarantee that the mock database will have almost identical statistics compared to the live database. Consequently, the mock database can be used to evaluate the performance of a database application.

A similar approach can be found in [BGB05]. The authors of this work devise a formal framework for database sampling. Their initial motivation was to generate a test database for testing new features of a database application. The framework extracts a test database from a live database that meets the same integrity constraints as the live database and includes all the “data-diversity” found in the live database. The resulting database is expected to better support the development of new features of a database application than a synthetic test database.

### 2.2.3 Resume

Some approaches for generating test databases that we presented in Section 2.2.2 [CDF+04; MR89; ZXC01; WE06] discuss the same problem statement as this thesis; i.e., generating test case aware databases. However, all these approaches fall short in many aspects tackled by this work:

- The main drawback of all these approaches is that they generate test databases for only a small subset of the SQL queries (like [MR89; ZXC01; WE06]) or they only consider certain fragments of the test query like the selection and/or the join predicates (like [CDF+04]). Consequently, these approaches cannot deal with all classes of possible SQL queries not to mention the complex semantics of database applications in general.

- Moreover, these approaches give ad-hoc solutions for the supported query classes so that the presented solutions cannot be extended easily.
- Another problem is that these approaches are not designed to generate huge amounts of data. For example, [ZXC01] and [WE06] first create one constraint formula and then instantiate this formula to generate the complete test database. However, the running time of a constraint solver is exponential to the input size of the constraint formula. Consequently, these approaches cannot deal with test databases for many practical problems when huge amounts of data are necessary (e.g., for the testing of OLAP applications).

All other approaches discussed in Section 2.2.1 and in Section 2.2.2 focus on orthogonal problems.

# Part II

# Reverse Query Processing

Chapter 3

# Motivating Applications

*All our dreams can come true – if we have the courage to pursue them.*

*– Walt Disney, 1902-1966 –*

When designing a completely new database application or modifying such an application (e.g., a reporting application or an E-Shop) it is necessary to generate one or more test databases in order to carry out all the necessary functional tests on the application logic to guarantee a certain quality of the application under test. As discussed in Section 2.2.2, there are a number of commercial and academic tools which enable the generation of a test database for a given database schema. Beside the database schema, those tools usually take value ranges, data repositories, or some constraints (e.g., the table sizes, statistical distributions) as input and generate a test database accordingly.

However, these tools generate test databases which do not reflect the semantics of the application logic that should be executed by a certain test case. For example, if a test case for a reporting application issues a complex SQL query against such a synthetic test database, it is likely that the SQL query returns no or non-meaningful results for testing that query. An example of a typical reporting query is shown below. The query lists the total sales of ordered line items per day, if the discounted price was less than a certain average and more than a certain sum (the database schema of the application is given in Figure 4.2 (a)):

```
SELECT o_orderdate, SUM(l_price*(1-l_discount)) as sum1
FROM lineitem, orders WHERE l_oid=o_id
GROUP BY o_orderdate
HAVING AVG(l_price*(1-l_discount))<=100
AND SUM(l_price*(1-l_discount))>=150;
```

The following tables show a real excerpt of the test database generated by a commercial test database generation tool[1] for the example application:

| l_id | l_name | l_price | l_discount | l_oid |
|---|---|---|---|---|
| 103132 | Kc1cqZlf | 810503883 | 0.7 | 1214077 |
| 126522 | hcTpT8ud34 | 994781460 | 0.1 | 1214077 |
| 397457 | 5SwWn9q3 | 436001336 | 0.0 | 1297288 |
| ... | ... | ... | ... | ... |

Table *lineitem*

| o_id | o_orderdate |
|---|---|
| 1214077 | 1983-01-23 |
| 1297288 | 1995-01-01 |
| ... | ... |

Table *orders*

Obviously, the query above returns an empty result for that test database because none of the generated tuples satisfies the complex `HAVING` clause (including different aggregations on arithmetic functions). Even though some tools allow the user to specify additional rules in order to constrain the generated databases (e.g., constraining the domain of the attribute $l\_discount$), those constraints are defined on the base tables only and there are no means to control the query results of a certain test query explicitly. Therefore, those tools can hardly deal with complex SQL queries used for reporting not to mention the complex semantics of database applications in general.

In order to generate meaningful test databases, this thesis proposes a new technique called *Reverse Query Processing* or RQP, for short. RQP takes a SQL query and the expected query result (in addition to the database schema) as input and generates a database that returns that result if the query is executed on that database. More formally, given a Query $Q$ and a Table $R$, RQP generates a Database $D$ (a set of tables) such that $Q(D) = R$.

One application of RQP is the regression testing of reporting applications (i.e., OLAP applications): The main use case of a reporting application is that a user executes ad-hoc reports on the business data. In order to test various types of reports, the tester could extract the SQL queries which implement the different reports from the application. Furthermore, the tester provides one or several sample results for each report that are interesting for the functional testing. A combination of a SQL query (i.e., a report) and a result of that report specify a test case for the reporting application. Such a test case can then be used to generate a test database by RQP which is adequate for that test case. The thus generated test databases can be used as a basis for the regression testing of the reporting application: i.e., if the reporting application is modified, the queries (i.e., reports) defined by the test cases can be re-executed on the corresponding test database and it can be checked if the actual result of a particular report is the same as the expected result that is defined by the test case.

Another important use case of a reporting application is that the user wants to display the results of a report in different formats by executing some actions like pivoting. Consequently, the functionality which shows the results of the reports on the screen strongly depends on the data that

[1]We do not disclose the name of the tool for legal reasons.

should be displayed. Consequently, in order to test the display functionality thoroughly, we can use RQP to generate different test databases for various reports and results of these reports that are to be displayed.

There are also several other applications of RQP: One application that we will describe in detail in Part III of this thesis is the generation of a test database for the functional testing of OLTP applications. While one SQL query and one result is usually sufficient to specify the database state to execute a test case for a reporting application, we usually need more than one SQL query to specify the characteristics of the test database to execute a test case of an OLTP application. The reason is that OLTP applications usually implement use cases which consist of sequences of actions whereas each action reads or updates different entities of the database (e.g., a use case of an E-Shop application that creates a new order would first read the relevant customer and product data from the database and then insert a new order using that data).

Another application that will be presented in Part III is the functional testing of a query language where it is important to verify the actual query result of an arbitrary test query to reveal defects in the query processing functionality. For that application we extend RQP to generate also an expected result for a given test query following certain input parameters (e.g., the result size). The expected result and the corresponding test query can then be used to generate a test database by RQP which returns the expected query result. During the test execution phase the expected result of a test query is used to verify the actual result of executing the test query on the generated test database.

**Contributions:** The main contribution of this part is the conceptual framework for RQP and a prototype implementation called SPQR (System for Processing Queries Reversely) which takes one SQL query and one expected result as input to generate a test database. Furthermore, this part gives the results of some performance experiments for the TPC-H benchmark [TPCb] using SPQR in order to demonstrate how well the proposed techniques scale for complex queries coming from typical OLAP applications. The other applications (i.e., functional testing of an OLTP applications and functional testing of a query language) will be discussed separately in Part III.

**Outline:** The remainder of this part is organized as follows: Chapter 4 defines the problem statement and gives an overview of the solution. Chapter 5 describes the reverse relational algebra (RRA) for RQP which is used to generate test databases for arbitrary SQL queries. Chapter 6 to 8 present the techniques implemented in SPQR, our prototype implementation for RQP. Chapter 9 describes the results of the experiments carried out using SPQR and the TPC-H benchmark. Chapter 10 discusses related work.

# Chapter 4

# RQP Overview

*My way is to seize an image that moment it has formed in my mind, to trap it as a bird and to pin it at once to canvas. Afterward I start to tame it, to master it. I bring it under control and I develop it.*

*– Joan Miró, 1893-1983 –*

In the last thirty years, a great deal of research and industrial effort has been invested in order to make query processing more powerful and efficient. New operators, data structures, and algorithms have been developed in order to find the answer to a query for a given database as quickly as possible. This thesis turns the problem around and presents methods in order to efficiently find out whether a table can possibly be the result of a query or not and, if so, what the corresponding database might look like.

Reverse query processing is carried out in a similar way as traditional query processing. At compile-time, a SQL query is translated into an expression of the relational algebra, this expression is rewritten for optimization and finally translated into a set of executable iterators [HFLP89]. At run-time, the iterators are applied to input data and produce outputs [Gra93]. What makes RQP special are the following differences:

- Instead of using the relational algebra, RQP is based on a reverse relational algebra. Logically, each operator of the relational algebra has a corresponding operator of the reverse relational algebra that implements its reverse function.
- Correspondingly, RQP iterators implement the operators of the reverse relational algebra which requires the design of special algorithms. Furthermore, RQP iterators have one input and zero or more outputs (think of a query tree turned upside down). As a consequence, the

best way to implement RQP iterators is to adopt a push-based run-time model, instead of a pull-based model which is typically used in traditional query processing [Gra93].

- An important aspect of reverse query processing is to respect integrity constraints of the schema of the database. Such integrity constraints can impact whether a legal database instance exists for a given query and query result. In order to implement integrity constraints during RQP, this work proposes to adopt a two-step query processing approach and make use of a model checker at run-time in order to find reverse query results that satisfy the database integrity constraints.
- Obviously, the rules for query optimization and query rewrite are different because the cost tradeoffs of reverse query processing are different. As a result, different rewrite rules and optimizations are applied.

As will be shown, reverse query processing for SQL queries is challenging. For instance, reverse aggregation is a complex operation. Furthermore, model checking is an expensive operation even though there has been significant progress in this research area in the recent past. As a result, optimizations are needed in order to avoid calls to the model checker and/or make such calls as cheap as possible.

## 4.1 Problem Statement and Decidability

As mentioned before, this thesis addresses the following problem for relational databases. Given a SQL Query $Q$, the Schema $S$ of a relational database (including integrity constraints), and a expected result $R$ (called $RTable$), find a database instance $D$ such that:

$$R = Q(D)$$

and $D$ is compliant with $S$ and its integrity constraints.

In general, there are many different database instances which can be generated for a given $Q$ and $R$. Depending on the application some of these instances might be better than others. In order to generate test databases for functional testing a reporting application, for instance, it might be advantageous to generate a small $D$ so that the running time of test cases is reduced. While the techniques presented in the following chapters try to be minimal, they do not guarantee any minimality. The purpose of this thesis is to find any viable solution. Studying techniques that make additional guarantees is one avenue for future work.

**Theorem 4.1** *Given an arbitrary SQL query $Q$, a result $R$, and a database schema $S$, it is not possible to decide whether a database instance $D$ exists that satisfies $S$ and returns $Q(D) = R$ or not.*

**Proof (Sketch) 4.2** *In order to show that RQP is undecidable, we reduce the query equivalence problem to RQP. However, as shown in [Klu80], the equivalence of two arbitrary SQL queries is undecidable*[1]. *As a result, RQP for SQL must be also undecidable; that is, in general it is not possible to decide whether a $D$ exists, if $Q$ does not follow the rules discussed in [Klu80].*

*An arbitrary instance of the query equivalence problem can be reduced to an instance of RQP as follows. Let $Q_1$ and $Q_2$ be two arbitrary SQL queries. In order to decide whether $Q_1$ and $Q_2$ are equivalent, we can use RQP to decide whether a database instance $D$ exists for the query $Q = \chi_{COUNT(*)}((Q_1 - Q_2) \cup (Q_2 - Q_1))$, a result $R$ of $Q$ which defines $COUNT(*) > 0$. Moreover, $D$ should meet the constraints of the database schema $S$. If RQP can find such a database instance $D$, then $Q_1$ and $Q_2$ are not equivalent (i.e., if $Q_1$ and $Q_2$ would be equivalent, the result of $Q$ must be empty). Otherwise, if RQP can not find such a database instance $D$, then it immediately follows that $Q_1$ and $Q_2$ are equivalent.*

Furthermore, there are obvious cases where no $D$ exists for a given $R$ and $Q$ (e.g., if tuples in $R$ violate basic integrity constraints). The approach presented in this thesis, therefore, cannot be complete. It is a best-effort approach: it will either fail (return an *error* because it could not find a $D$) or return a valid $D$.

## 4.2 RQP Architecture

Figure 4.1 gives an overview of the proposed architecture to implement reverse query processing. A query is (reverse) processed in four steps by the following components:

**Query Compilation:** The SQL query is parsed into a query tree which consists of operators of the relational algebra. This parsing is carried out in exactly the same way as in a traditional SQL processor. What makes RQP special is that that query tree is translated into a *reverse query tree*. In the reverse query tree, each operator of the relational algebra is translated into a corresponding operator of the *reverse relational algebra*. The reverse relational algebra is presented in more detail in Chapter 5. In fact, in a strict mathematical sense, the reverse relational algebra is not an algebra and its operators are not operators because they allow different outputs for the same input. Nevertheless, we use the terms *algebra* and *operator* in order to demonstrate the analogies between reverse and traditional query processing.

**Bottom-up Query Annotation:** The second step is to propagate schema information (e.g., data types, attribute names, and integrity constraints) to the operators of the query tree. Furthermore,

[1]Two arbitrary SQL queries $Q_1$ and $Q_2$ are equivalent, iff $Q_1$ and $Q_2$ return the same result $R$ for all possible database instances $D$.

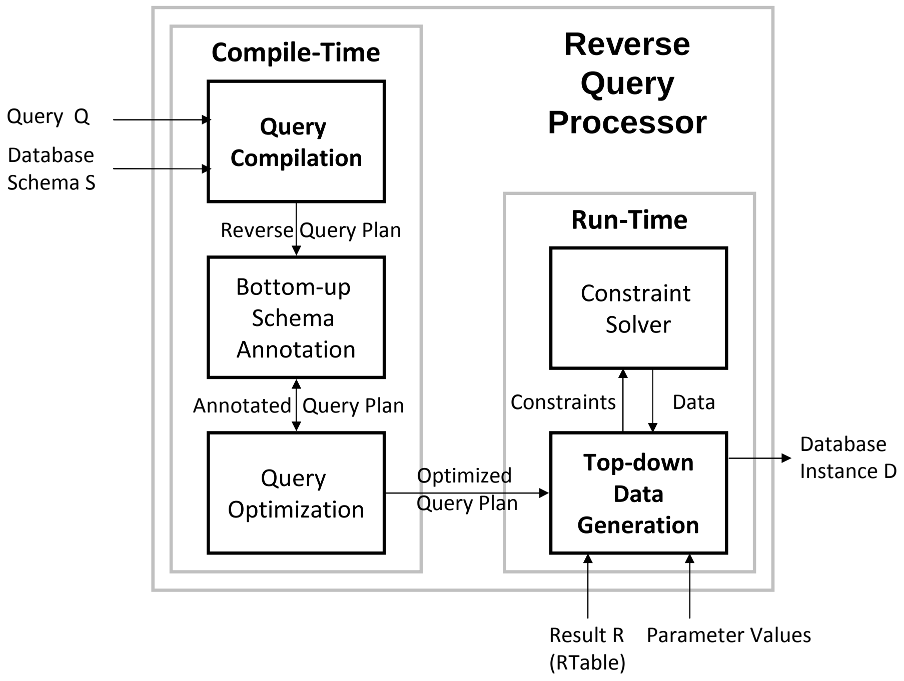


Figure 4.1: RQP Architecture

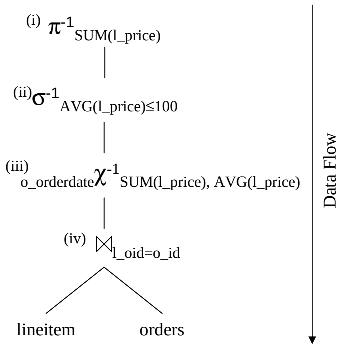
properties of the query (e.g., predicates) are propagated to the operators of the reverse query tree. As a result, each operator of the query tree is annotated with constraints that specify all necessary conditions of its output. Chapter 6 describes this process in more detail. That way, for example, it can be guaranteed that a top-level operator of the reverse query tree does not generate any data that violates one of the database integrity constraints.

**Query Optimization:** In the last step of compilation, the reverse query tree is transformed into an *equivalent* reverse query tree that is expected to be more efficient at run-time. An example optimization is the unnesting of queries. Unnesting and other optimizations are described in Chapter 8.

**Top-down Data Instantiation:** At run-time, the annotated reverse query tree is interpreted using the result $R$ ($RTable$) as input. Just as in traditional query processing, there is a physical implementation for each operator of the reverse relational algebra that is used for reverse query execution. In fact, some operators have alternative implementations which may depend on the application (e.g., test database generation involves different algorithms than testing data security, see Part III). The result of this step is a valid database instance $D$. As part of this step, we propose to use a model checker (more precisely, the decision procedure of a model checker) in order to

```
CREATE TABLE lineitem (
l_id INTEGER PRIMARY KEY,
l_name VARCHAR(20),
l_price FLOAT,
l_discount FLOAT
 CHECK (1>= discount >=0),
l_oid INTEGER);

CREATE TABLE orders(
o_id INTEGER PRIMARY KEY,
o_orderdate DATE);

SELECT SUM(l_price)
FROM lineitem, Orders
WHERE l_oid=o_id
GROUP BY o_orderdate
HAVING AVG(l_price)<=100;
```

**(a) Example Schema and Query**


(i) $\pi^{-1}_{\text{SUM(l\_price)}}$
(ii) $\sigma^{-1}_{\text{AVG(l\_price)}\leq 100}$
(iii) $_{\text{o\_orderdate}}\chi^{-1}_{\text{SUM(l\_price), AVG(l\_price)}}$
(iv) $\bowtie_{\text{l\_oid=o\_id}}$
lineitem
orders
Data Flow

**(b) Reverse Relational Algebra Tree**

| SUM(l_price) |
|---|
| 100 |
| 120 |

(i) $RTable$

| o_orderdate | SUM(l_price) | AVG(l_price) |
|---|---|---|
| 1990-01-02 | 100 | 100 |
| 2006-07-31 | 120 | 60 |

(ii) Output of $\pi^{-1}$; Input of $\sigma^{-1}$

| o_orderdate | SUM(l_price) | AVG(l_price) |
|---|---|---|
| 1990-01-02 | 100 | 100 |
| 2006-07-31 | 120 | 60 |

(iii) Output of $\sigma^{-1}$; Input of $\chi^{-1}$

| l_id | l_name | l_price | l_discount | l_oid | o_id | o_orderdate |
|---|---|---|---|---|---|---|
| 1 | productA | 100.00 | 0.0 | 1 | 1 | 1990-01-02 |
| 2 | productB | 80.00 | 0.0 | 2 | 2 | 2006-07-31 |
| 3 | productC | 40.00 | 0.0 | 2 | 2 | 2006-07-31 |

(iv) Output of $\chi^{-1}$; Input of $\bowtie^{-1}$

| l_id | l_name | l_price | l_discount | l_oid |
|---|---|---|---|---|
| 1 | productA | 100.00 | 0.0 | 1 |
| 2 | productB | 80.00 | 0.0 | 2 |
| 3 | productC | 40.00 | 0.0 | 2 |

lineitem

| o_id | o_orderdate |
|---|---|
| 1 | 1990-01-02 |
| 2 | 2006-07-31 |

orders

**(c) Input and Output of Operators**

Figure 4.2: Example Schema and Query for RRA

generate data [CGP00]. How this Top-down data instantiation step is carried out is described in more detail in Chapter 7.

In many applications, queries have parameters (e.g., bound by a host variable). In order to process such queries, values for the query parameters must be provided as input to Top-down data instantiation. The choice of query parameters again depends on the application; for test database generation, for instance, it is possible to generate several test databases with different parameter settings derived from the program code. In this case, the first three phases of query processing need only be carried out once, and the Top-down data instantiation can use the same annotated reverse query tree for each set of parameter settings.

It is also possible to use constraint formulas on variables in the $RTable$ $R$. That way, it is possible to specify tolerances. For example, a user who wishes to generate a test database for a decision support application could specify an example report for sales by product. Rather than specifying exact values in the example report, the user could say that the sales for, say, tennis rackets are $x$ with $90K \leq x \leq 110K$. This additional constraint for variable $x$ would be considered during the execution of Top-down data instantiation. Specifying such tolerances has two important advantages. First, depending on the SQL query it might not be possible to find a test database that generates a report with the exact value of 100K for the sales. That is, the RQP instance might simply not be satisfiable. Second, specifying tolerances (if that is acceptable for the application) can significantly speed-up reverse query processing because it gives the model checker more options to find solutions.

## 4.3 RQP Example

Figure 4.2 gives an example of reverse query processing. Figure 4.2a shows the database schema (definition of the $lineitem$ and $orders$ tables with their integrity constraints) and a SQL query that asks for the sales (i.e., $SUM(l\_price)$) grouped by $o\_orderdate$. The query is parsed and optimized and the result is a reverse query tree with operators of the reverse relational algebra. The resulting reverse query tree is shown in Figure 4.2b. This tree is very similar to the query tree used in traditional query processors. The differences are that (a) operators of the reverse relational algebra (Section 5) are used and (b) that the data flow through that tree is from the top to the bottom (rather than from the bottom to the top).

The data flow at run-time is shown in Figure 4.2 (c). Starting with an $RTable$ that specifies that two result tuples should be generated (Table (i) at the top of Figure 4.2 (c), each operator of the reverse relational algebra is interpreted by the Top-down data instantiation component in order to produce intermediate results of reverse query processing. In this phase, RQP uses the decision procedure of a model checker in order to guess appropriate values (e.g., possible order dates). Of course, several solutions are possible and the decision procedure of the model checker chooses possible values that match all constraints discovered in the Bottom-up annotation step randomly: depending on the application, alternative heuristics could be used in order to generate values that are more advantageous for the application. The final result of RQP in this example are possible instantiations for the $lineitem$ and $orders$ tables. It is easy to see that these instantiations meet the integrity constraints of the database schema and that (forward) executing the SQL query using these instantiations gives the $RTable$ as a result.

Figure 4.2 does not demonstrate how the Bottom-up query annotation component annotates the reverse query tree using the integrity constraints of the database schema and properties of the query. The example, however, does show the effects of that step. For example, the result of reverse projection (Table (ii) in Figure 4.2 (c) generates values for the $AVG(price)$ column which are compliant with the predicate of the `HAVING` clause of the query. This process is described in more detail in Chapter 6.

Chapter 5

# Reverse Relational Algebra

*Algebra is generous; she often gives more than is asked of her.*

*– Jean Baptiste le Rond d'Alembert, 1717-1783 –*

The Reverse Relational Algebra (RRA) is a reverse variant of the traditional relational algebra [Cod70] and its extensions for group-by and aggregation [GMUW01]; i.e., each operator of the relational algebra has a corresponding operator in the reverse relational algebra. The symbols of the operators are the same (e.g., $\sigma$ for selection), but each operator $op$ of the RRA are marked as $op^{-1}$ (e.g., $\sigma^{-1}$). Furthermore, the following equation holds for all operators and all valid tables R:

$$op(op^{-1}(R)) = R$$

However, reverse operators in RRA should not be confused with *inverse* operators because the following formula is **not** necessarily true for some valid tables $S$: $op^{-1}(op(S)) = S$

In the traditional relational algebra, an operator has 0 or more inputs and produces exactly one output relation. Conversely, an operator of the RRA has exactly one input and produces 0 or more output relations. Just as in the traditional relational algebra, the operators of the RRA can be composed. As shown in Figure 4.2 (b), the composition is carried out according to the same rules as for the traditional relational algebra. As a result, it is very easy to construct a reverse query plan for RQP by using the same SQL parser as for traditional query processing.

The close relationship between RRA and the traditional relational algebra has two consequences:

- *Basic Operators:* The reverse variants of the basic operators of the (extended) relational algebra (selection, projection, rename, cartesian product, union, aggregation, and minus) form the basis of the RRA. All other operators of the RRA (e.g., reverse outer joins) can be expressed as compositions of these basic operators.

- *Algebraic Laws:* The relational algebra has laws on associativity, commutativity, etc. on many of its operators. Analogous versions of most of these laws apply to the RRA. Some laws are not applicable for the RRA (e.g., applying projections before joins); these laws are listed in [Klu80] and must be respected for RQP optimization (Section 8).

The remainder of this Chapter defines the seven basic operators of the reverse relational algebra, which form the basis for a complete implementation of a reverse query processor. A physical implementation (e.g., algorithms) of the RRA operators for generating test databases is described in Chapter 7.

## 5.1 Reverse Projection

The reverse projection operator $\pi^{-1}$ generates new columns according to its *output schema*. The output schema of an operator is defined as the set of attributes and constraints (from the database schema and the query) of the output relation generated by the operator. The output schema of each operator is created in the Bottom-up annotation phase (Chapter 6). Again, as for all operators of the reverse relational algebra, $\pi(\pi^{-1}(R)) = R$ must apply for all valid $R$.

In Figure 4.2, the reverse projection creates the $o\_orderdate$ and $AVG(l\_price)$ columns. In order to generate correct values for these columns, the reverse project operator needs to be aware of the constraints imposed by the aggregations (SUM and AVG) and the HAVING clause of the query. That is, the values in the $AVG(l\_price)$ column must be smaller or equal to 100 so that the $\sigma^{-1}$ does not fail. Furthermore, the value of the $o\_orderdate$ column must be unique and the values in the $AVG(l\_price)$ and $SUM(l\_price)$ columns must match so that the reverse aggregation ($\chi^{-1}$) does not fail. In this specific example, there are no integrity constraints from the database schema or functional dependencies that must be respected as part of the reverse projection. In general, such constraints must also be respected in an implementation of the $\pi^{-1}$ operator.

An algorithm to implement the $\pi^{-1}$ operator is presented in Chapter 7. This algorithm is based on calls to the decision procedure of a model checker in order to fulfill all constraints or fail (i.e., return *error*), if the constraints cannot be fulfilled.

## 5.2 Reverse Selection

The simplest operator of the reverse relational algebra is the reverse selection ($\sigma^{-1}$): It either returns *error* or a superset (or identity) of its input. *Error* is returned if the input of the reverse select operator does not match the selection predicate. For example, if the query asks for all employees with salary greater than 10,000 and the $RTable$ contains an employee with salary 1,000, then *error* is returned. Another example of $\sigma^{-1}$ is given in Figure 4.2 (c). Table (ii) in Figure 4.2 (c) (the output of $\pi^{-1}$) is the input of $\sigma^{-1}$. Since the input of $\sigma^{-1}$ is compliant with its output schema, the output of $\sigma^{-1}$ (Table (iii) in Figure 4.2c) is the same as its input.

## 5.3 Reverse Aggregation

Like the $\pi^{-1}$ operator, the reverse aggregation operator $\chi^{-1}$ generates columns. Furthermore, the reverse aggregation operator possibly generates additional rows in order to meet all constraints of its aggregate functions. Again, as for all RRA operators, the goal is to make sure that $\chi(\chi^{-1}(R)) = R$ and that the output is compliant with all constraints of the output schema (e.g., functional dependencies, predicates, etc.). If this is not possible, then the reverse aggregation fails and returns *error*. An algorithm to implement the $\chi^{-1}$ operator using the decision procedure of a model checker is presented in Section 7.

Tables (iii) and (iv) of Figure 4.2 (c) show the input and output of reverse aggregation for the running example. In that example, the values of the $l\_id$, $l\_name$, and $l\_discount$ columns are generated obeying the integrity constraints of the $lineitem$ table (top of Figure 4.2 (a). The value of the $l\_price$ column is generated using the input (the result of the reverse selection) and the intrinsic mathematical properties of the aggregate functions. The values of the $l\_oid$ and $o\_id$ columns are generated obeying the constraints imposed by the join predicate of the query and the *primary-key* constraint of the $orders$ table.

## 5.4 Reverse Join, Cartesian Product

The reverse join operator $\bowtie^{-1}$ completes the running example. It takes one relation as input and generates two output relations. Like all other operators, the reverse join makes sure that its outputs meet the specified output schemata (the database schema for the $lineitem$ and $orders$ tables in the example of Figure 4.2) and that the join of its outputs gives the correct result. If it is not possible to fulfill all these constraints, then an *error* is raised. Really, the only thing that is special about the $\bowtie^{-1}$ operator is that it has two outputs. Again, an efficient algorithm to implement a reverse join is presented in Chapter 7. The reverse Cartesian product is a variant of the reverse join with *true* as a join predicate.

## 5.5 Reverse Union

Like the reverse join, the reverse union operator ($\cup^{-1}$) takes one relation as input and generates two output relations. According to the constraints of the output schemata of the two output relations,

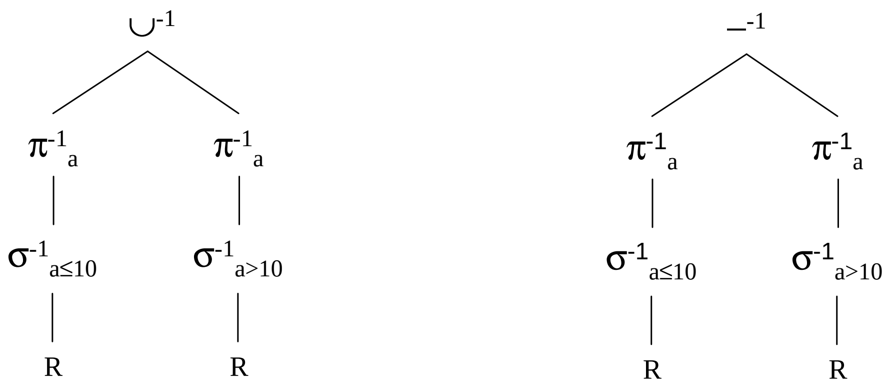


Figure 5.1: Reverse Union (left); Reverse Minus (right)

the reverse union distributes the tuples of the input relation to the corresponding output relations. An example is given in the left part of Figure 5.1. Both relations $R$ and $S$ have an attribute $a$. Let the input for the reverse union be three tuples: $\{\langle 2\rangle, \langle 12\rangle, \langle 8\rangle\}$. In this case, the reverse union must output $\langle 2\rangle$ to the left reverse selection operator and output $\langle 12\rangle$ to the right selection operator. $\langle 8\rangle$ can be output to either the left or the right selection operator. If the input of a reverse union involves a tuple that does not fulfill the constraints of any branch (this is not possible in the example of Figure 5.1), then the reverse union fails and returns *error*.

## 5.6 Reverse Minus

An example of a reverse minus operator ($-^{-1}$) is shown in the right part of Figure 5.1. Input tuples are always routed to the left branch or result in an error. Furthermore, it is possible that the $-^{-1}$ generates new tuples for both branches in order to meet all its constraints. In this example, the reverse minus would output an input tuple $\langle 2\rangle$ (or any other input with $a \leq 5$) to its left branch, and it would return *error* if its input contains a tuple with $a > 5$. No new tuples need to be generated in this example.

## 5.7 Reverse Rename

The reverse rename operator has the same semantics as in the traditional relational model. Thus, only the output schema is affected; no data manipulation is carried out.

Chapter 6

# Bottom-up Query Annotation

*The more constraints one imposes, the more one frees one's self. And the arbitrariness of the constraint serves only to obtain precision of execution.*

*– Igor Stravinsky, 1882-1971 –*

The bottom-up query annotation phase in Figure 4.1 annotates each operator $op^{-1}$ of a reverse query tree with an *output schema* $S^{OUT}$ and an *input schema* $S^{IN}$. This way, each operator can check the correctness of the input and ensure that it generates valid output data.

**Definition 6.1** *(Input/Output Schema $S$:) A schema $S$ (input and output) in RQP is formally defined as the following four-tuple:*

$$S = (A, C, F, J)$$

*The tuple defines (1) the attributes $A$, (2) the integrity constraints $C$, (3) the functional dependencies $F$ and (4) the* join dependencies *$J$ (as well as multivalued dependencies as special cases of $J$).*

The set of attributes $A$ defines the attribute names $name(a)$, the data type $type(a)$, and the frequency $|a|$ for each attribute $a \in A$.

**Definition 6.2** *(Attribute Frequency $|a|$:) The frequency $|a|$ of an attribute $a \in A$ defines how often the same attribute instance (i.e. value of a tuple) can be used in a relation instance that satisfies $S$. The frequency is either given by a constant $c$ (i.e., $|a| = c$) or as $|a| \geq 1$.*

| Notation | Description |
|---|---|
| $S$ | Schema $S$ |
| $S.A$ | Attributes |
| $S.C$ | Integrity Constraints |
| $S.F$ | Functional dependencies |
| $S.J$ | Join dependencies |
| $C_{CK}$ | Check constraints |
| $C_{UN}$ | Unique constraints |
| $C_{PK}$ | Primary-key constraints |
| $C_{NN}$ | Not null constraints |
| $C_{AGG}$ | Aggregation constraints |

Table 6.1: Notations used in the bottom-up phase

For example, the frequency of the attribute $o\_id$ in the input schema of the reverse union operator of the reverse query expression $(orders \cup^{-1} orders)$ is two (i.e., $|orders.o\_id| = 2$) because the same value will be used twice in the result of the reverse query expression.

The *join dependencies* used in that work are a generalization of those known from textbooks like [GMUW01].

**Definition 6.3** *(Join Dependency $JD$:) A* join dependency $JD$ *in that work is defined as follows:*

$$JD = (A_1, A_2, p)$$

*A $JD$ defines that the projection of the relation $R$ to the attributes in $A_1 \cup A_2$ must represent a lossless join on of the two projections of $R$ to $A_1$ and $R$ to $A_2$ using $p$ as join predicate: $\pi_{A_1 \cup A_2}(R) = (\pi_{A_1}(R)) \bowtie_p (\pi_{A_2}(R))$.* Join dependencies *with more than two sets of attributes can be represented as a recursive combination of these* join dependencies. *More details on how the* join dependencies *are calculated for each reverse operator will be given in the corresponding sections.*

Moreover, RQP considers the integrity constraints of SQL (*Primary-Key*, *Unique*, *Foreign-key*, *Not Null*, and *Check*) as well as *Aggregation* constraints [RSSS94]. In order to denote the different constraint types in a schema $S$, we use $S.C_{CK}$, $S.C_{UN}$, $S.C_{PK}$, $S.C_{NN}$ and $S.C_{AGG}$. The notations used in this chapter are summarized in Table 6.1.

Obviously, a unary operator (e.g., $\sigma^{-1}$) in the RRA has only one output schema ($S^{OUT}$) whereas a binary operator (e.g., $\bowtie^{-1}$) has two output schemata. In a reverse query tree, the output schema of an operator must match the input schema of the reverse operator which consumes the data from that operator in in a reverse query tree. For example, the input schema of the $\sigma^{-1}$ is the same as the output schema of the $\pi^{-1}$ in the example of Figure 4.2 (b) because the reverse selection consumes data from the reverse projection in the reverse query tree.

In order to annotate each operator of a reverse query tree the annotation phase operates in a bottom-up way. It starts with the output schemata of the leaves of the reverse query tree (e.g., the operators that read the $lineitem$ and $orders$ in Figure 4.2 (b)). Consequently, the output schemata of these leaves are defined by the database schema (e.g., the SQL DDL code of Figure 4.2 (a). Then, for each operator, the input schema is computed from the output schema of the operator. This input schema is then used to initialize the output schema of the operator at the next level up.

[Klu80] showed that the problem of calculating constraints that hold on the intermediate results of arbitrary relational queries is undecidable. In this chapter, we discuss a set of best-effort rules to calculate the constraints that hold on the intermediate results ($S^{IN}$ and $S^{OUT}$) of an arbitrary relational query.

The remainder of this chapter defines the set of best-effort rules used for the annotation of each RRA operator and shows how the bottom-up phase works for each operator of the example reverse query tree in Figure 4.2. Furthermore, we also show how the bottom-up phase works for nested queries. In this regard, our work is an extension of the work presented in [Klu80]; that work describes how *functional dependencies* and *check* constraints expressing the equality can be propagated for expressions of the relational algebra. We extend that work for all elements ($A$, $C$, $F$ and $J$) contained in a schema $S$ and add rules for the aggregation operator ([Klu80] did not discuss the aggregation operator). As shown later, the *primary-key* and the *unique* constraints in $C$ can be derived from $F$, the attribute frequency, and the *not null* constraints in $S$. Furthermore, the rules introduced in the sequel use full qualified attribute names (relation name and attribute name) instead of the position of an attribute in a relation (which is used in [Klu80]) in order to identify the attributes uniquely. Another extension is that we assume *bag* semantics, as in SQL.

## 6.1 Leaf initialization

As stated above, the output schemata of the leaves of the reverse query tree are initialized using the database schema $S$. We assume that a database schema which is used as input of the bottom-up annotation phase (see Figure 4.1) defines a schema $S_R = (A, C, F, J)$ for each relation $R$. For each attribute $a \in A$ we set $|a| = 1$ if $a$ has a *unique* or a *primary-key* constraint. Otherwise we set $|a| \geq 1$. In order to initialize a leaf of a RRA expression representing a relation $R$, the bottom-up phase must extract the corresponding schema $S_R$ out of the database schema $S$.

Foreign-key constraints defined in the output schema are treated specially in the bottom-up phase. They are rewritten as a reverse equi-join with a join predicate representing the *primary-key*/foreign-key relationship.

**Example:** Assume the table $lineitem$ in the example of Figure 4.2 (a) defines a *foreign-key* on the attribute $l\_oid$ which refers the *primary-key* attribute $o\_id$ of the relation $orders$ and we want to reverse process the following query:

```
SELECT l_name
FROM lineitem
WHERE price>100
```

This query would then be rewritten as:

```
SELECT l_name
FROM lineitem, orders
WHERE l_price>100 and l_oid=o_id
```

For the rest of the input schema elements of a leaf, they are the same as the elements of the leaf's output schema. For example, the input and output schemata of the $lineitem$ and $orders$ table in Figure 4.2 (b) can be represented in the following way (there are no *unique*, no *foreign-key*, and no *not null* constraints in this example): the attributes $A$ of both schemata $S_{lineitem}$ and $S_{orders}$ define the attribute name, the type and the frequency of each attribute.

| | |
|---|---|
| $A$: | $l\_id$; `INTEGER`; $\lvert l\_id \rvert = 1$, |
| | $l\_name$; `VARCHAR(20)`; $\lvert l\_name \rvert \geq 1$, |
| | $\ldots$ |
| | $l\_oid$; `INTEGER`; $\lvert l\_oid \rvert \geq 1$ |
| $C$: | `PRIMARY KEY(`$l\_id$`)` |
| | `CHECK(1` $\geq$ `l_discount` $\geq$ `0)` |
| $F$: | $\{l\_id\} \rightarrow \{l\_name, l\_price, ..., l\_oid\}$ |
| $J$: | $\emptyset$ |

$S_{lineitem}$

| | |
|---|---|
| $A$: | $o\_id$; `INTEGER`; $\lvert o\_id \rvert = 1$ |
| | $o\_orderdate$; `DATE`; $\lvert o\_orderdate \rvert \geq 1$ |
| $C$: | `PRIMARY KEY(` $oid$`)` |
| $F$: | $\{o\_id\} \rightarrow \{o\_orderdate\}$ |
| $J$: | $\emptyset$ |

$S_{orders}$

## 6.2 Reverse Join

The reverse join has two output schemata called $S_{left}^{OUT}$ and $S_{right}^{OUT}$. Its input schema $S^{IN}$ is computed from these two output schemata by the following rules:

(1) $S^{IN}.A = S_{left}^{OUT}.A \cup S_{right}^{OUT}.A$;

- If $p$ is an equi-join predicate $a_1 = a_2$ ($a_1 \in S_{left}^{OUT}.A$ and $a_2 \in S_{left}^{OUT}.A$) and there is a *primary-key* constraint or a *unique* constraint in the database schema $S$ on the attribute $a_1$ (or $a_2$), then for each $a \in S^{IN}.A \cup S_{right}^{OUT}.A$ (or for each $a \in S^{IN}.A \cup S_{left}^{OUT}.A$) set $|a| \geq 1$
- Else for each $a \in S^{IN}.A$ set $|a| \geq 1$

(2) $S^{IN}.F = closure(S_{left}^{OUT}.F \cup S_{right}^{OUT}.F \cup createFD(p))$;

- $p$ denotes the join predicate
- $createFD(p)$ is a function to create *functional dependencies* from predicates (see Figure 6.1).

  - the function $closure$ is the function to compute the closure of a given set of *functional dependencies* in [Klu80].

(3) $S^{IN}.J = S^{OUT}_{left}.J \cup S^{OUT}_{right}.J \cup JD(S^{OUT}_{left}.A, S^{OUT}_{right}.A, p)$

(4) $S^{IN}.C$ is defined for each type as follows:

  (4.1) $S^{IN}.C_{CK} = S^{OUT}_{left}.C_{CK} \cup S^{OUT}_{right}.C_{CK} \cup p$;

    - $p$ denotes the join predicate;

  (4.2) $S^{IN}.C_{NN} = S^{OUT}_{left}.C_{NN} \cup S^{OUT}_{right}.C_{NN}$;

  (4.3) $(S^{IN}.C_{PK}, S^{IN}.C_{UN}) = createPKAndUnique(S^{IN}.F, S^{IN}.C_{NN}, S^{IN}.A)$

    - $createPKAndUnique$ is a function to create *primary-key* and *unique* constraints from functional dependencies, *not null* constraints, and attributes (see Figure 6.2).

  (4.4) $S^{IN}.C_{AGG} = S^{OUT}_{left}.C_{AGG} \cup S^{OUT}_{right}.C_{AGG}$;

The set of attributes $S^{IN}.A$ of the input schema is the union of the set of attributes from the reverse join's output schemata (rule 1). The frequency of each attribute $a \in S^{IN}.A$ is set to $|a| \geq 1$ if the join predicate $p$ is not an equi-join predicate on an attribute in $S^{OUT}_{left}$(or $S^{OUT}_{right}$) with a *primary-key* or a *unique* constraint in the database schema $S$. Otherwise, we set $|a| \geq 1$ only for those attributes in $S^{IN}.A$ that come from $S^{OUT}_{right}$(or $S^{OUT}_{left}$).

The *functional dependencies* $S^{IN}.F$ of the input schema are defined as the closure of the union of the *functional dependencies* in the reverse join's output schemata and the *functional dependencies* computed from the join predicate by the function $createFD$ in Figure 6.1 (rule 2). The function $createFD$ takes a predicate $p$ as input and outputs a set of derivable *functional dependencies*. This function deals with arbitrary predicates by transforming the given predicate into conjunctive normal form (Line 3 in Figure 6.1). The conjunctive normal form of a predicate consists of one or more conjuncts, each of which is a disjunction (OR) of one or more literals (simple predicates with no boolean operator). Afterwards, each conjunct is analyzed separately (Line 5 in Figure 6.1). In case that the conjunct only consists of a simple predicate expressing the equality, it is transformed into a set of functional dependencies (Line 9 to 15).

The *join dependencies* $S^{IN}.J$ of the input schema are defined as a union of the *join dependencies* in the reverse join's output schemata and a new *join dependency* computed from the attributes of both output schemata and the join predicate $p$ (rule 3). Thus we are able to express joins on joined relations.

The *check* constraints (rule 4.1) are the union of the *check* constraints from the output schemata and the join predicate. The *not null* constraints (rule 4.2) are the union of the *not null* constraints of the output schemata. The *unique* constraints and *primary-key* constraints can be derived from $F$, the *not null*, and the attributes in $S^{IN}$ (rule 4.3). The function $createPKAndUnique$ (see Figure 6.2) used by that rule takes the *functional dependencies* $F$, the *not null* constraints $NN$, and the of attributes $A$ as input and outputs all *primary-key* and *unique* constraints implied by $F$,

**createFD(Predicate $p$)**

**Output:**

-Set $F$ // Set of *functional dependencies*

```
(1)  //transform $p$ to conjunctive normal form
(2)  //$cnf\_p = p_{OR1} \wedge ... \wedge p_{ORn}$
(3)  $cnf\_p$ = CNF($p$)
(4)  //Analyze each conjunct $p_{ORi}$
(5)  FOREACH $p_{OR}$ in $cnf\_p$
(6)   //domain equality:  $a_i = a_j$
(7)   //value equality $a_i = c$;
(8)   //$a_i$, $a_j$ are attributes; $c$ is a constant
(9)   IF($p_{OR}$ is domain equality)
(10)    //e.g.  add ($\{a_i\} \rightarrow \{a_j\}$), ($\{a_j\} \rightarrow \{a_i\}$)
(11)    $F$.add($\{p_{OR}.leftAtt()\} \rightarrow \{p_{OR}.rightAtt()\}$)
(12)    $F$.add($\{p_{OR}.rightAtt()\} \rightarrow \{p_{OR}.leftAtt()\}$)
(13)  ELSE IF($p_{OR}$ is value equality)
(14)    //e.g.  add ($\emptyset \rightarrow \{a_i\}$)
(15)    $F$.add($\emptyset \rightarrow \{p_{OR}.leftAtt()\}$)
(16)  //ELSE do nothing for complex predicates
(17)  END IF
(18) END FOR
(19) RETURN F
```

Figure 6.1: Function *createFD*

$NN$, and $A$. A *functional dependency* $f$ expresses a *unique* or *primary-key* constraint on the set of attributes $A$, if all attributes $A$ appear in the right side of $f$ and all attributes in the left side of $f$ have a frequency of one (Line 6 in Figure 6.2). When there are *not null* constraints on the left side of $f$, then a *primary-key* constraint is added for the attributes; else a *unique* constraint is added for the attributes (Line 7-10 in in Figure 6.2).

The *aggregation* constraint (rule 4.4) is a new type of constraint which is explained in the following section. These constraints are also computed as union of the *aggregation* constraints of the two output schemata.

Going back to the example of Figure 4.2, the two output schemata of the $\bowtie^{-1}$ are given by the input schemata of the relations $lineitem$ and $orders$. Following the complete set of rules for $\bowtie^{-1}$, the resulting input schema of the $\bowtie^{-1}$ can be represented as follows:

```
createPKAndUnique(Functional dependencies $F$,Not null constraints $NN$,
                  Attributes $A$)
Output:
 -Set $PK$ // Set of primary-key constraints
 -Set $UN$ // Set of unique constraints
(1)  $PK = UN = \emptyset$
(2)  //analyze $F$
(3)  FOREACH $f$ in $F$
(4)   //if all attributes $A$ are in right side of $f$
(5)   //and each attribute $a$ in left side has $|a| == 1$
(6)   IF($A$-$f$.rightAtts() == $\emptyset$ && $|a| == 1$ for each attribute $a \in A$)
(7)    IF($NN$ has a constraint for $f$.leftAtts())
(8)     $PK$.add(PK($f$.leftAtts()))
(9)    ELSE
(10)    $UN$.add(UNIQUE($f$.leftAtts()))
(11)   END IF
(12)  END IF
(13) END FOR
(14) RETURN ($PK$,$UN$)
```

Figure 6.2: Function *createPKAndUnique*

$A$: $l\_id$; INTEGER; $|l\_id| = 1$,
$\ldots$,
$l\_oid$; INTEGER; $|l\_oid| \geq 1$,
$o\_id$; INTEGER; $|o\_id| \geq 1$,
$o\_orderdate$; DATE; $|o\_orderdate| \geq 1$
$C$: PRIMARY KEY($l\_id$), /*from lineitem*/
CHECK(1 $\geq$ l_discount $\geq$ 0), /*from lineitem*/
CHECK($o\_id = l\_oid$) /*join predicate*/
$F$: $\{l\_id\} \rightarrow \{l\_name, \ldots, o\_id, o\_orderdate\}$ ,
$\{o\_id\} \rightarrow \{o\_orderdate\}$ ,
$\{l\_oid\} \rightarrow \{o\_id\}$ ,
$\{o\_id\} \rightarrow \{l\_oid\}$
$J$: $JD(\{l\_id, ..., l\_oid\}, \{o\_id, o\_orderdate\}, (l\_id = l\_oid))$ ,

## 6.3 Reverse Aggregation

The input schema of a reverse aggregation operator is defined by the following rules:

(1) $S^{IN}.A = A_{gr} \cup A_{agg}$;

- $A_{gr}$ denotes the GROUP BY attributes,
- $A_{agg}$ denotes the attributes of the new aggregate columns of the SELECT and HAVING clause
- If $A_{gr} \neq \emptyset$ then for each $a \in A_{gr}$ set $|a| = 1$ and for each $a \in A_{agg}$ set $|a| \geq 1$; Else for each $a \in A_{agg}$ set $|a| = 1$

(2) $S^{IN}.F = closure(cleanFD(S^{OUT}.F, A_{gr}) \cup \{A_{gr} \rightarrow A_{agg}\})$;

– $cleanFD$ is a function to filter unrelated FDs (see Figure 6.3).

(3) $S^{IN}.J = cleanJD(S^{OUT}.J, A_{gr})$

– $cleanJD$ is a function to filter unrelated JDs (see Figure 6.4).

(4) $S^{IN}.C$ is defined for each type as follows:

(4.1) $S^{IN}.C_{CK} = cleanConstraints(S^{OUT}.C, (S^{IN}.A \cup A_{agg}.atts()), CK)$;

– $cleanConstraints$ is a function to clean constraint (see Figure 6.5).

(4.2) $S^{IN}.C_{NN} = cleanConstraints(S^{OUT}.C, (S^{IN}.A \cup A_{agg}.atts()), NN) \cup$ $createNotNull(A_{agg}, S^{OUT}.C_{NN})$;

– $createNotNull$ is a function to create *not null* constraints (see Figure 6.7).

(4.3) $S^{IN}.C_{AGG} = cleanConstraints(S^{OUT}.C, (S^{IN}.A \cup A_{agg}.atts()), AGGREGATION) \cup$ $AGGREGATION(A_{gr}, A_{agg})$

(4.4) $(S^{IN}.C_{PK}, S^{IN}.C_{UN}) = createPKAndUnique(S^{IN}.F, S^{IN}.C_{NN}, S^{IN}.A)$;

– $createPKAndUnique$ is a function to create *primary-key* and *unique* constraints from *functional dependencies*, *not null* constraints, and attributes (see Figure 6.2).

The attributes $A$ of $S^{IN}$ are given by the attributes in the `GROUP BY` clause of the query plus the aggregate columns specified in the `SELECT` and `HAVING` clause of the query (rule 1). The frequency for each attribute in the `GROUP BY` clause is set to $|a| = 1$ and to $|a| \geq 1$ for the aggregate columns. If the query has no `GROUP BY` clause, then the frequency for the aggregate columns is set to $|a| = 1$.

The computation of $F$ is listed in rule 2. It first uses the function $cleanFDs$ (Figure 6.3) to keep only *functional dependencies* $f$ with at least one of the attributes of the left side of $f$ in the input schema (Line 5 to 6). Then a new *functional dependency* which expresses that all aggregate columns are functional dependent from the attributes in the `GROUP BY` clause is added. If no `GROUP BY` clause exists, an empty set is used as left side of the new *functional dependency*.

The computation of $J$ is shown by rule 3. It uses the function $cleanJD$ (Figure 6.4) to keep only those attributes in a *join dependency* $j$ with at least one of the attributes in the `GROUP BY` clause.

The *check* and *not null* integrity constraints (rule 4.1 and rule 4.2) are inherited from the output schema only if they are correlated to any attribute in the input schema or a metric attribute $A_{agg}.atts()$ of the aggregation functions $A_{agg}$. The function $cleanConstraints$ (Figure 6.5) takes a set of integrity constraints $C^{OUT}$, a set of attributes $A$, and the constraint type as input and outputs those integrity constraints $C^{IN}$ of the given type which are correlated to any attribute in $A$. In order to find correlated integrity constraints, the function $cleanConstraints$ invokes a function $createConstraintGraph$ (Figure 6.6) to create a constraint graph (Line 2 in Figure 6.5) whose vertices represent the given integrity constraints in $C^{OUT}$ and whose edges show if

**cleanFD(Functional dependencies $F_{OUT}$, Attributes $A_{IN}$)**
**Output:**
-Set $F_{IN}$ // Cleaned *functional dependencies*

```
 (1)  $F_{IN} = \emptyset$
 (2)  //analyze FDs in $F^{OUT}$
 (3)  FOREACH $f$ in $F^{OUT}$
 (4)   //if left attributes of $f$ are in $A^{IN}$
 (5)   IF($f$.leftAtts() $\cap$ $A^{IN}$ != $\emptyset$)
 (6)    $f$.rightAtts() = $f$.rightAtts() $\cap$ $A^{IN}$
 (7)    $F^{IN}$.add($f$)
 (8)   END IF
 (9)  END FOR
(10)  RETURN $F^{IN}$
```

Figure 6.3: Function *cleanFD*

**cleanJD(Join dependencies $J_{OUT}$, Attributes $A_{IN}$)**
**Output:**
-Set $J_{IN}$ // Cleaned *join dependencies*

```
 (1)  $J_{IN} = \emptyset$
 (2)  //analyze each JDs in $J^{OUT}$
 (3)  FOREACH $j$ in $J^{OUT}$
 (4)   //analyze $A_1$ and $A_2$ in $j$ given by $j$.atts()
 (5)   FOREACH set $A$ in $j$.atts()
 (6)    //remove attributes not in $A^{IN}$ from $j$
 (7)    $j.A_1 = j.A_1 \cap A^{IN}$; $j.A_2 = j.A_2 \cap A^{IN}$
 (8)    //add $j$ to $J_{IN}$ if $A_1$ and $A_2$ is not empty
 (9)    IF($j.A_1 \neq \emptyset$ && $j.A_2 \neq \emptyset$) $J_{IN} = J_{IN} \cup j$
(10)   END FOR
(11)  END FOR
(12)  RETURN $J^{IN}$
```

Figure 6.4: Function *cleanJD*

**cleanConstraints(Constraints $C^{OUT}$, Attributes $A$, Type $t$)**

**Output:**

-Set $C^{IN}$ // Cleaned integrity constraints

```
 (1) //create constraint graph of $C^{OUT}$
 (2) $G^{OUT}$ = createConstraintGraph($C^{OUT}$)
 (3) $G^{IN}$ = ($\emptyset$, $\emptyset$)
 (4) //analyze attributes $A$
 (5) FOREACH $a$ in $A$
 (6)  //analyze constraints of $G^{OUT}$
 (7)  FOREACH $c$ in $G^{OUT}.V$
 (8)   //if $a$ is in attributes of $c$
 (9)   IF($a \in c.atts()$)
(10)    //subgraph calculates all constraints
(11)    //connected to vertex c in $G^{OUT}$
(12)    $G^{SUB}$ = $G^{OUT}$.subgraph($c$)
(13)    //add constraints to $G^{IN}$
(14)    $G^{IN}$.add($G^{SUB}$)
(15)    $G^{OUT}$.remove($G^{SUB}$)
(16)   END IF
(17)  END FOR
(18) END FOR
(19) //the vertices of $G^{IN}$ are the constraint
(20) $C^{IN}$ = $G^{IN}.V$
(21) IF(t!=$\emptyset$) RETURN $C_t^{IN}$
(22) ELSE RETURN $C^{IN}$
```

Figure 6.5: Function *cleanConstraints*

two constraints refer to at least one common attribute. The function keeps all integrity constraints which are connected to an integrity constraint, which refers to at least one attribute in $A^{IN}$ (Line 9 to 16).

The *aggregation* constraint (represented by $S^{IN}.C_{AGG}$) in Rule 4.3 is a new type of constraint introduced in [RSSS94]. An aggregation constraint specifies the requirements of the aggregation functions and the `GROUP BY` clause. They are also computed by the function $cleanConstraints$. Additionally, a new aggregation constraint for that operator is added, too.

The *primary-key* and *unique* constraints (rule 4.4) can be derived from $F$, as already described for the reverse join.

In the example of Figure 4.2, the output schema of the $\chi^{-1}$ is given by the input schema of the $\bowtie^{-1}$. The input schema of the $\chi^{-1}$ is specified as follows.

```
createConstraintGraph(Set $C$)
Output:
 -Graph $G = (V, E)$ // Graph of correlated constraints

 (1)  $V$ = $\emptyset$
 (2)  $E$ = $\emptyset$
 (3)  //integrity constraints in set $C$
 (4)  FOREACH $c$ in $C$
 (5)   FOREACH $c'$ in $V$
 (6)    //if $c$ and $c'$ have common attributes
 (7)    IF($c.atts() \cap c'.atts() != \emptyset$)
 (8)     $E$.add($c$, $c'$)
 (9)    END IF
(10)   END FOR
(11)   $V$.add($c$)
(12)  END FOR
(13)  RETURN $G = (V, E)$
```

Figure 6.6: Function *createConstraintGraph*

$A$: $o\_orderdate$; DATE; $|o\_orderdate| = 1$,
SUM($l\_price$); FLOAT; $|SUM(l\_price)| \geq 1$,
AVG($l\_price$); FLOAT; $|AVG(l\_price)| \geq 1$
$C$: PRIMARY KEY ($o\_orderdate$),
AGGREGATION(GROUP BY $o\_orderdate$,
{SUM($l\_price$), AVG($l\_price$)} )
$F$: $\{o_orderdate\} \rightarrow \{$SUM($l\_price$), AVG($l\_price$)$\}$
$J$: $\emptyset$

## 6.4 Reverse Selection

The input schema of a reverse selection inherits $A$, $F$, $C$, and $J$ from its output schema. The only difference between the output and input schema is that the selection predicate is added to the *check* constraints of the input schema. The selection predicate is translated into corresponding *functional dependencies* in the same way as for the predicates of a reverse join (see Figure 6.1).

In the example of Figure 4.2, the input schema of the $\sigma^{-1}$ is almost identical with the input schema of the $\chi^{-1}$ (previous paragraph): only the *check* constraint with the predicate $AVG(l\_price) \leq 100$ is added to the constraints $C_{CK}$.

## 6.5 Reverse Projection

The $\pi^{-1}$ operator has similar rules as the $\chi^{-1}$ operator. The rules are as follows:

```
createNotNull(Aggregation constraints $C_{AGGS}$, Not null constraints $C_{NN}$)
Output:
 //Not null constraints for aggregation functions
 -Set $C_{NN'}$

 (1)  $C_{NN'} = \emptyset$
 (2)  //analyze aggregation functions in $AGGS$
 (3)  FOREACH $agg$ in $C_{AGGS}$
 (4)   //if all metrics are NOT_NULL
 (5)   IF($agg$.atts() - $C_{NN}.atts()$ = $\emptyset$)
 (6)    $C_{NN'}$.add(NOT_NULL($agg$))
 (8)   END IF
 (9)  END FOR
(10)  RETURN $C_{NN'}$
```

Figure 6.7: Function *createNotNull*

(1) $S^{IN}.A = A_{proj}$;

- $A_{proj}$ denotes the projected attributes.

(2) $S^{IN}.F = cleanFD(S^{OUT}.F, S^{IN}.A)$;

- $cleanFD$ is the same function as before (see Figure 6.3)

(3) $S^{IN}.J = cleanJD(S^{OUT}.F, S^{IN}.A)$;

- $cleanJD$ is a function to filter unrelated JDs (see Figure 6.4).

(4) $S^{IN}.C = cleanConstraints(S^{OUT}.C, S^{IN}.A, \emptyset)$;

- $cleanConstraints$ is the same function as before, see Figure 6.5.

The attributes of the input schema (rule 1) are derived from the attributes in the `SELECT` clause (the projection does not change the frequency). The *functional dependencies* and the *join dependencies* (rule 2 and 3) are calculated by the functions $cleanFD$ and $cleanJD$ just like in reverse aggregation. Also, the integrity constraints (rule 4) are calculated by the function $cleanConstraints$ which keeps all constraints correlated to the attributes in the input schema.

In the example of Figure 4.2, the input schema of the $\pi^{-1}$ is as follows.

$A$: SUM($l\_price$); FLOAT; $|SUM(l\_price)| \geq 1$
$C$: CHECK(AVG($l\_price$) $\leq 100$),
AGGREGATION(GROUP BY $o\_orderdate$,
{SUM($l\_price$), AVG($l\_price$)} )
$F$: $\emptyset$
$J$: $\emptyset$

In the example, the *check* constraint is correlated to the *aggregation* constraint. Thus, it is kept in the input schema, although the attribute $AVG(l\_price)$ itself is not kept. The reason is that the function *createConstraintGraph* which is used in order to calculate the correlated constraints calls a method $atts()$ of each integrity constraint in the output schema (see Figure 6.6, Line 7). This method call on an *aggregation* constraint returns all aggregated columns (e.g., $AVG(price)$) plus all metrics of the aggregation functions (e.g., $l\_price$). Thus constraints correlated to all aggregation functions and metrics are kept in the input schema.

## 6.6 Reverse Union

The reverse union has two output schemata like the reverse join. Its input schema is computed from the two output schemata by the following rules.

(1) $S^{IN}.A = S^{OUT}_{left}.A$;

- For each $a \in S^{IN}.A$ set $|a| = S^{OUT}_{left}.|a| + S^{OUT}_{right}.|a|$. If $S^{OUT}_{left}.a$ or $S^{OUT}_{right}.a$ has a frequency $\geq 1$ then set $|a| \geq 1$.

(2) $S^{IN}.F = S^{OUT}_{left}.F \cap S^{OUT}_{right}.F$;

(3) $S^{IN}.J = \emptyset$;

(4) $S^{IN}.C$ is defined for each type as follows:

(4.1) $S^{IN}.C_{CK} = (S^{OUT}_{left}.C_{CK}) \vee (S^{OUT}_{right}.C_{CK})$;

(4.2) $S^{IN}.C_{NN} = S^{OUT}_{left}.C_{NN} \cap S^{OUT}_{right}.C_{NN}$;

(4.3) $S^{IN}.C_{AGG} = S^{OUT}_{left}.C_{AGG} \cap S^{OUT}_{right}.C_{AGG}$;

(4.4) $(S^{IN}.C_{PK}, S^{IN}.C_{UN}) = createPKAndUnique(S^{IN}.F, S^{IN}.C_{NN}, S^{IN}.A)$; (see Figure 6.2)

The set of attributes $A$ of the input schema is equal to the set of attributes of its left output schema (rule 1), if the attribute types of both output schemata match. The frequency of the attributes in $A$ is the sum of the input frequencies. An example is given in the introduction of this chapter.

The *functional dependencies* in the input schema (rule 2) are computed by the intersection of the *functional dependencies* of the two output schemata. Rule 3 states that the *join dependencies* are initialized with an empty list. Obviously, at this point we *loose* some constraints (i.e., our rule set is not complete) as we discussed before.

The derivation of the *check* constraints is more complex (rule 4.1): the set of *check* constraints of the input schema is computed by combining the set of *check* constraints from the left output schema with the set of *check* constraints from the right output schema disjunctively. However, as the attribute names could be different in the right output schema they have to be renamed to the corresponding attribute of the left output. The *not null* and *aggregation* constraints are computed by an intersection of these constraints of both output schemata (rule 4.2 and rule 4.3). The *primary-key* and *unique* constraints (rule 4.4) are again derived from $F$, as already described for the reverse join.

**Example:** In the reverse union example in Figure 5.1, the new *check* constraints with the predicate $(R.a \leq 10) \vee (R.a > 5)$ of the input schema of the reverse union is derived from the predicates of the *check* constraints in the two output schemata ($(R.a \leq 10)$ and $(S.a > 5)$). We can see that the attributes of the *check* constraint of the right output schema are renamed.

## 6.7 Reverse Minus

The reverse minus operator has also two output schemata. To derive the input schema we generally consider its left output schema only. The schema computation for the input schema of the reverse minus operator is given by the following rules:

(1) $S^{IN}.A = S^{OUT}_{left}.A$

(2) $S^{IN}.F = S^{OUT}_{left}.F$

(3) $S^{IN}.J = \emptyset;$

(4) $S^{IN}.C = S^{OUT}_{left}.C \wedge \neg S^{OUT}_{right}.C_{CK};$

The set of attributes of the input schema as well as all *functional dependencies* and other integrity constraints are equal to the left output schema (rule 1, 2 and 4). The *join dependencies* are again initialized with an empty list (rule 3). In addition to these rules, (rule 4) states that a *check* constraint which is the negation of the conjunction of all *check* constraint predicates of the right output schema is added. In the reverse minus example in Figure 5.1, a *check* constraint with the predicate $!(b > 5)$ is added to the input schema of reverse minus.

## 6.8 Reverse Rename

To derive the input schema we only rename the corresponding attribute respectively relation names of the output schema in $A$, $C$, $F$ and $J$.

## 6.9 Annotation of Nested Queries

In order to reverse process a nested query, SPQR uses the concept of nested iterations (sometimes called apply operators) which are known from traditional query processing [GLJ01], in a reverse way (see Section 7.10). A nested query has the following general structure:

```
OUTER QUERY
bind predicate
INNER QUREY
correlation predicate
```

In many cases, the bottom-up phase can be applied to the outer and inner query block separately. However, if the inner query is a query connected by equality (*bind predicate*) to the outer query, then the reverse apply operator adds an additional *functional dependency* to the outer query. In case that the inner query is correlated to the outer query, the *correlation predicate* must express the equality, otherwise no *functional dependency* is added to input schema of the reverse selection of the outer query. The *functional dependency* added by the reverse apply operator has the following structure (*correlation attribute* and *bind attribute* are the attributes of the outer query used in the *correlation predicate* and the *bind predicate*):

$$\{correlation\ attribute\} \rightarrow \{bind\ attribute\}$$

**Example:** Assume the following query is given:

```
SELECT s_age, s_salary
FROM Student
WHERE s_age =
  SELECT MAX(p\_age)
  FROM Professor
  WHERE p_salary=s_salary
```

In that example, a *functional dependency* $\{s\_salary\} \rightarrow \{s\_age\}$ is added to the input schema of the reverse selection operator of the outer query.

Chapter 7

# Top-down Data Instantiation

*I'd like our software somehow automatically recognizing your data and your situation and respond to that without you having to set it up.*

*– Scott Cook –*

The Top-down data instantiation component in Figure 4.1 interprets the optimized reverse query execution plan using an $RTable\ R$ and query parameters as input. It generates a database instance $D$ as output. The generated database $D$ fulfills the constraints of the database schema and the overall correctness criterion of RQP under the decidability concerns as mentioned in Section 4.1. If this is not possible, then *error* is returned.

A reverse query execution plan consists of a set of physical RRA operators. As in traditional query processing, the set of physical RRA operators is called the physical reverse relational algebra. Each logical RRA operator may have different counterparts in the physical RRA. The choice may be application dependent; for example, different physical implementations are used for SQL debugging and for scalability testing. This chapter presents the physical algebra of SPQR, a prototype of RQP. The physical algebra of SPQR tries to keep the generated database as small as possible.

Moreover, there is a limitation on implementing some physical RRA operators: If the same database table is referenced multiple times in a reverse query tree, then the physical implementations of $\sigma^{-1}$, $\bowtie^{-1}$ and $-^{-1}$ are not allowed to generate additional tuples for that table. This limitation does not affect the physical RRA in this thesis as these operators generate no additional tuples in order to keep $D$ as small as possible. But this limitation does affect physical algebras which generate additional tuples (e.g., a physical algebra for performance testing).

**Example:** That problem can be shown by the following example query which should be reverse processed disregarding the rule above (Table $S$ has the attributes $A, B$). We see that table $S$ is referenced multiple times.

```
SELECT S1.A,S1.B,S2.A,S2.B
FROM S as S1, S as S2
WHERE S1.B=S2.B AND
S1.A>5 AND S2.A<=5;
```

Assume that a result $R$ is given which has only one tuple `<6,1,5,1>`. The reverse query tree for that query contains two reverse selections (one on $S1$ and one on $S2$). The reverse selection $A > 5$ on $S1$ pushes `<6,1>` down to $S1$ and creates an additional tuple which satisfies $!(A > 5)$, e.g. `<5,2>` for $S1.A, S1.B$. The reverse selection $A <= 5$ on $S2$ pushes `<5,1>` down to $S2$ and creates an additional tuple which satisfies $!(A <= 5)$, e.g. `<6,2>` for $S2.A, S2.B$. So at the end $S$ would contain four tuples `{<6,1>,<5,2>,<5,1>,<6,2>}`. If we run the query above on the generated tuples in $S$ the result would contain two tuples `{<6,1,5,1>, <6,2,5,2>}` and not only one tuple as defined by $R$.

The remainder of this chapter is organized as follows. At the beginning we introduce the general architectural model used to implement the physical RRA operators. Afterwards, a non blocking implementation is shown for each RRA operator which can be used in most cases. Some special cases which need a blocking implementation, as well as the reverse processing of nested queries are discussed afterwards. Finally, optimizations for some RRA operators are presented.

## 7.1 Iterator Model

As in traditional query processing, each operator is implemented as an iterator [Gra93]. Unlike traditional query processing, the iterators are push-based. That is, whenever an operator produces a tuple, it calls the *pushNext* method of the relevant child (output) operator(s) and continues processing once the child operator(s) is (are) ready. Thus, the whole data instantiation is started by scanning the $RTable$ $R$ and pushing each tuple of $R$ one at a time to the root operator of the reverse query plan. A push-based model is required because operators of the RRA can have multiple outputs; the alternative would be to implement a pull-based model with buffering which is significantly more complex [MF02]. All iterators have the same interface which contains the following three methods:

- *open*: prepare the iterator for producing data (as in traditional query processing).

- *pushNext(Tuple $t$)*: (a) receive a tuple $t$, (b) check if $t$ satisfies the input schema $S^{IN}$ of the operator, (c) produce zero or more output tuples, and (d) for each output tuple, call the *pushNext* method of the relevant children operators.
- *close*: clean up everything (as in traditional query processing).

The following subsections show how the operators produce tuples in their *pushNext* method. All other aspects (e.g., *open* and *close*) are straightforward so that the details are omitted for brevity.

## 7.2 Reverse Projection

In SPQR, the reverse project operator produces exactly one output tuple for each input tuple. In order to generate values for new columns, the reverse project operator calls the decision procedure of a model checker. The idea is to create a constraint formula which represents the constraints which have to be satisfied by the output. These constraints represent the values known from the input tuple on the one hand and the output schema on the other hand. For example, if the input schema has one column ($A$), the input tuple is `<3>`, and the output schema has two columns ($A$ and $B$) and an additional constraint that $A + B < 30$, then the following constraint formula is generated:

```
A = 3 & A+B < 30
```

This constraint formula is passed to the model checker which in turn generates values for all variables or *error* if no instantiations that satisfy the formula can be found. In this example, the model checker would return, say, $A = 3, B = 20$ and these values would be used to generate an output tuple.

Figure 7.1 shows the pseudocode of how the $\pi^{-1}$ operator which generates an output tuple from an input tuple. The most important statement is the call of the *instantiateData* function (Line 2) which does the actual work. Since this function is also used by the implementation of the $\chi^{-1}$ operator, it has two return parameters: one which defines the instantiated data (variable, value pairs) and another which indicates how many tuples are used to solve aggregations which might be part of the formula (see below). The second return value is only needed for the $\chi^{-1}$ operator so that it can be ignored for the moment. If the call to *instantiateData* was successful (i.e., $I \neq NULL$ in Line 3), then a new output tuple is created according to the output schema of the $\pi^{-1}$ operator and passed to the next reverse operator (Lines 6 to 8). Otherwise, *error* is returned (Line 4).

The pseudocode of a simplified version of the *instantiateData* function is shown in Figure 7.2. This function creates a constraint formula $L$ (Line 9) following the semantics of the reverse operator and executes the decision procedure of the model checker on $L$ (Line 10). As part of the creation of the constraint formula, restrictions of the model checker need to be taken into account. For

```
$\pi^{-1}$.pushNext(Tuple $t$)
//Instantiate output data
($I$,$count$)=instantiateData($t$,$S^{OUT}$)
IF($I$=NULL) //no instantiation found
   RETURN error;
ELSE
   $t_{out}$=createTuple($I$,$S^{OUT}$,1)
   //push down the new tuple $t_{out}$
   nextOperator.pushNext($t_{out}$)
END IF
```

Figure 7.1: Method *pushNext* of $\pi^{-1}$

```
instantiateData(Tuple $t$, Schema $S^{OUT}$)
Output:
 -Instantiation $I$ //data instantiation
 -int $n$ //number of tuples for aggregation
//number of tuples for aggregation
IF $t$ includes COUNT of aggregation
   $count$,$maxcount$=COUNT value in $t$
ELSE //USER_THREHOLD=1 if no aggregation
   $count$=1; $maxcount$=USER_THRESHOLD
END IF
FOR($n$=$count$ TO $maxcount$)
   //Create constraint formula $L$
   $L$=createConstraint($t$,$S^{OUT}$,$n$)
   $I$=decisionProcedure($L$)
   IF($I$!=NULL) RETURN ($I$,n)
END FOR //Trial-and-error
RETURN (NULL,0)
```

Figure 7.2: Function *instantiateData* (simplified)

example, the model checker used in the performance experiments (Section 9) does not support SQL numbers and dates. As a result, all SQL numbers and dates must be converted into (long) integers and the constraints must be adjusted accordingly. Furthermore, arithmetic expressions (e.g., $A + B$) which might appear in the input and output schema of the reverse projection must be taken into account.

The most complex part of the *instantiateData* function deals with the generation of columns that involve aggregations. In Figure 4.2, for example, the $\pi^{-1}$ operator needs to generate values for the $AVG(l\_price)$ column. In order to generate correct values, the *instantiateData* function needs to guess how many tuples are aggregated by the aggregate function; for instance, two tuples are aggregated for the second tuple of the $RTable\ R$ in Figure 4.2. The two tuples are generated by the $\chi^{-1}$ operator, but the $\pi^{-1}$ operator which only generates one output tuple per input tuple must be aware of this fact in order not to generate values that cannot be matched by the $\chi^{-1}$ operator. Unfortunately, today's publicly available model checkers have not been designed for aggregation so that this guessing must be carried out as part of the *instantiateData* function in a trial and error phase (Lines 6 to 11). The guessing iteratively tries different values of *n* (the number of tuples aggregated) and calls the decision procedure for each value until the decision procedure of the model checker was successful and able to instantiate data.

Continuing the example of Figure 4.2 for the second tuple of the $RTable\ R$ ($SUM(l\_price) = 120$), the following formula is generated for $n = 1$:[1]

```
sum_l_price=120 &
o_orderdate!=19900102 & avg_l_price<=100 &
sum_l_price=l_price1 & avg_l_price=sum_l_price/1
```

This formula is given to the decision procedure of the model checker and obviously, the model checker cannot find values for the variables $price1$ and $avg\_price$ that meet all constraints. In the second attempt for $n = 2$, the following formula is passed to the decision procedure:

```
sum_l_price=120 &
o_orderdate!=19900102 & avg_l_price<=100
sum_l_price=l_price1+l_price2 & avg_l_price=sum_l_price/2
```

This time, the decision procedure finds an instantiation:[2]

```
sum_l_price=120, avg_l_price=60,
l_price1=80, l_price2=40,
o_orderdate=20060731
```

From this instantiation, the values of $o\_orderdate$, $avg\_l\_price$, and $sum\_l\_price$ are used in order to generate the output tuple of the reverse project operator. In the SPQR prototype, the maximum number of attempts ($maxcount$ in Figure 7.2) can be constrained by the user in order to make sure that the whole process does not run for ever. Moreover, all the guessing is not necessary if the query involves a `COUNT` aggregation because the values (or constraints) of the corresponding $COUNT$ column in the tuple ($t$) can be used (Lines 2 and 3 of Figure 7.2). Furthermore, in order to avoid the guessing, several optimizations can be applied (Section 7.11). These optimization techniques work very well such that in practice not much guessing is required; in fact, the experimental results in Chapter 9 show that no guessing is required for the whole TPC-H benchmark.

[1]The constraint on `orderdate` is generated because $o\_orderdate$ is the *primary-key* attribute of the output schema and, thus, a different $o\_orderdate$ value must be generated for the tuple with $SUM(l\_price) = 120$ than for the tuple´with $SUM(l\_price) = 100$. 19900102 is the integer representation for the date January 2, 1990, the $o\_orderdate$ value of the tuple with $SUM(l\_price) = 100$.

[2]20060731 is the integer representation of the date July 7, 2006.

```
$\chi^{-1}$.pushNext(Tuple $t$)
 (1)  //Instantiate data
 (2)  ($I$,$count$)=instantiateData($t$,$S^{OUT}$)
 (3)  IF($I$=NULL) //no instantiation found
 (4)     RETURN error;
 (5)  ELSE
 (6)     FOR($n$=1 TO $count$)
 (7)        $t_{out}$ =createTuple($I$,$S^{OUT}$,$n$)
 (8)        nextOperator.pushNext($t_{out}$)
 (9)     END FOR
(10)  END IF
```

Figure 7.3: Method *pushNext* of $\chi^{-1}$

The pseudocode of Figure 7.2 is a simplification for the special case that there are no nested aggregations (e.g., `SUM(AVG(price))`) and no joins on aggregated values (e.g., aggregations in several subqueries). However, the code can easily be generalized for all cases. This generalization is not shown because it is fairly straightforward. SPQR indeed implements such a generalized version of the *instantiateData* function.

## 7.3 Reverse Aggregation

The reverse aggregation operator can be implemented in an analogous way to the reverse projection. The difference is that while the $\pi^{-1}$ operator only guesses how many tuples are potentially involved in an aggregation, the $\chi^{-1}$ operator actually generates these tuples. The key idea to use the decision procedure of a model checker, however, is the same.

Figure 7.3 shows the pseudo-code. The *instantiateData* function is called in the same way as for $\pi^{-1}$. The only difference is that the return parameter $count$ is now initialized (Line 2) which defines the number of output tuples. If the *instantiateData* function was successful, then $count$ tuples are generated (Lines 6 to 9) using the values returned by the *instantiateData* function. If not, then *error* is generated (Lines 3 and 4). Again, an example that shows this code in action can be seen in Figure 4.2 (c) (Tables (iii) and (iv)).

## 7.4 Reverse Join

The reverse join operator can be implemented in different ways, depending on the join predicate. The simplest (and cheapest) implementation is the implementation of an equi-join that involves a *primary-key* or an attribute with a *unique* constraint. Such joins are the most frequent joins in practice. They can be implemented as a simple projection with duplicate elimination. The

```
$\cup^{-1}$.pushNext(Tuple $t$)
 (1) //Create constraint formulas
 (2) $L_{left}$=createConstraint($t$,$S^{OUT}_{left}$)
 (3) $L_{right}$=createConstraint($t$,$S^{OUT}_{right}$)
 (4) //call model checker
 (5) IF(decisionProcedure($L_{left}$)!=NULL)
 (6)   left_operator.pushNext($t$)
 (7) //call model checker
 (8) ELSE IF(decisionProcedure($L_{right}$)!=NULL)
 (9)   right_operator.pushNext($t$)
(10) ELSE
(11)   return $error$
(12) END IF
```

Figure 7.4: Method *pushNext* of $\cup^{-1}$

implementation of general joins and Cartesian products is more complex; the full algorithms are given in Section 7.9.1. In any event, the implementation of reverse joins and Cartesian products do not involve calls to a model checker so that these operators are much cheaper than reverse projections and aggregations.

## 7.5 Reverse Selection

The simplest implementation of the $\sigma^{-1}$ operator would return its input (i.e., implement the identity function). For example in Figure 4.2 (c), the $\sigma^{-1}$ implements the identity function such that its output relation (Table (iii) in Figure 4.2 (c)) is identical to its input relation (Table (ii) in Figure 4.2 (c)). If any input tuple is not compliant with the output schema, then *error* is returned.

## 7.6 Reverse Union

Like the reverse join, the reverse union operator takes one relation as input and generates two output relations. According to the output schemata of the two output relations, the reverse union operator distributes the input tuples to the correct output relation.

Figure 7.4 shows a implementation of the reverse union. The implementation checks for each input tuple if it is complaint with the output schema of the left output relation by creating a constraint formula representing the input tuple and the constraints imposed by the output schema (Line 1 to 3); and pushes the tuple to the left output relation if they are compatible (Line 6). Otherwise, the reverse union checks the compatibility of the input tuple with the right output relation (Line 8). If an input tuple is not complaint with any output relations, then *error* is returned (Line 11). Obviously, the reverse union implementation is cheap: its complexity is Linear to the input size.

## 7.7 Reverse Minus

The implementation of the reverse minus operator is similar to the reverse union operator. It checks for each input tuple if it is compliant with the left output schema but not the right output schema; and pushes the input tuple to the left output if possible. Otherwise, it returns *error*. Again, the complexity of this implementation is Linear to the input size just like the reverse union operator.

## 7.8 Reverse Rename

Since the reverse rename operator does not have any data manipulation, its implementation is the same as the reverse selection: it returns identity.

## 7.9 Special Cases

The implementations of the operators discussed so far are all non-blocking. That is, whenever an operator takes in a tuple, the operator can push the result tuple(s) to the child output operator immediately after processing. However, in some very special cases, RQP needs to use blocking RRA operators in order to guarantee correctness and they are discussed in details in this section. These special cases, however, are very rare in practice. For example, the TPC-H benchmark used in the experiments does not have any of the special cases and all non-blocking operators described above were used in the experiments.

### 7.9.1 Reverse Join

As discussed before the reverse equi-join that involves a *primary-key* or an attribute with a *unique* constraint is trivial. However, all other reverse joins need more complex blocking implementations which are shown in the following.

**Case 1:** If the join predicate expresses the equality of two attributes ($a_i = a_j$) and both $a_i$ and $a_j$ are not the *primary-key* or an attribute with a *unique* constraint of the output schemata, then a blocking implementation of the reverse join operator is needed.

The blocking implementation is shown in Figure 7.5. First, the complete input relation is grouped by the attributes of the left output schema (Line 1). Afterwards each group is analyzed (Line 3 to 24). If the group does not fulfill the join predicate an *error* is returned (Line 5 to 7). Afterwards, the left and right output are created for that group (Line 10,11). If any of both outputs (in the algorithm we use the left output) of the previous group has the same value for the join attribute as the current group, then the current right output must be the same as the previous right output;

else an *error* is returned (Line 14 to 19). If the input is correct, then the current left and right outputs are propagated to the next operators and they are saved as previous outputs for the next loop execution (Line 21 to 24).

Moreover, if one of the output schemata allows duplicates, the reverse join operator has to find out the correct cardinality of the outputs out of different possibilities. In that case duplicate elimination is needed in Line 10 and Line 11. However, this extension is straightforward and not shown in this thesis.

**Example:** An example of that case can be shown by the following query:

```
SELECT c_id, c_age, s_id, s_age
FROM customer, supplier
WHERE c_age=s_age
```

Both relations ($customer$ and $supplier$) have a *primary-key* attribute $id$. The input is given by the following two tuples: `<1, 27, 1 , 27>` and `<2, 27, 2 , 27>`. Both tuples are in separate groups because the attributes of the left output $c\_id$ and $c\_age$ have different values. As the reverse join produces different supplier tuples for the right output of both groups, although they have the same attribute value for the join attribute $c\_age$, the input is incorrect. A correct input should have four tuples: `{<1, 27, 1 , 27>, <1, 27, 2 , 27>, <2, 27, 1 , 27>, <2, 27, 2 , 27>}`.

**Case 2:** If the join is not an equi-join and the join predicate is in the form of $a_i > a_j$ or in the form of $a_i \geq a_j$, then the blocking version of the reverse join operator is needed, too.

**Example:** Consider the following query and the given input:

```
SELECT c_id, c_age, s_id, s_age
FROM customer, supplier
WHERE c_age > s_age
```

| c_id | c_age | s_id | s_age | |
|---|---|---|---|---|
| 1 | 27 | 1 | 25 | /*1st group*/ |
| 1 | 27 | 2 | 26 | |
| 2 | 28 | 1 | 25 | /*2nd group*/ |
| 2 | 28 | 3 | 27 | |

```
$\bowtie^{-1}$.pushNext(Relation $r$)
$r\_groups$ = groupby($r$, $S_{left}^{OUT}.A$)
//analyze each group in $r\_groups$
FOREACH $r\_group$ in $r\_groups$
 //check join predicate
 IF($r\_group$ not fulfills $\bowtie^{-1}$.p)
  RETURN $error$
 END IF
 //projection (with dupl.  elimination)
 //to attributes of output schemata
 $left_{out}$ = $r\_group[S_{left}^{OUT}.A]$
 $right_{out}$ = $r\_group[S_{right}^{OUT}.A]$
 //if join values of previous group
 //are equal to current group
 IF($left_{pre}[\bowtie^{-1}.p.att()] == left_{out}[\bowtie^{-1}.p.att()]$)
 //then right outputs must be the same
  IF($right_{out} != right_{prev}$)
   RETURN $error$
  END IF
 END IF
 left_operator.pushNext($left_{out}$)
 right_operator.pushNext($right_{out}$)
 $left_{pre}$ = $left_{out}$
 $right_{pre}$ = $right_{out}$
END FOR
```

Figure 7.5: Case 1: Method *pushNext* of $\bowtie^{-1}$

```
$\ltimes^{-1}$.pushNext(Relation $r$)
 (1) $r\_groups$ = groupby($r$, $S^{OUT}_{left}.A$)
 (2) $r\_groups$ = sortbyatt($r\_groups$, $S^{OUT}_{left}.A \cap \ltimes^{-1}.p.atts()$, $\ltimes^{-1}.p$)
 (3) //analyze each group in $r\_groups$
 (4) FOREACH $r\_group$ in $r\_groups$
 (5)  //check join predicate
 (6)  IF($r\_group$ not fulfills $\ltimes^{-1}$.p)
 (7)   RETURN $error$
 (8)  END IF
 (9)  //projection (with dupl.  elimination)
(10)  //to attributes of output schemata
(11)  $left_{out}$ = $r\_group[S^{OUT}_{left}.A]$
(12)  $right_{out}$ = $r\_group[S^{OUT}_{right}.A]$
(13)  //right output of successor group
(14)  //must be contained in previous group
(15)   IF($right_{prev} - right_{out} != \emptyset$)
(16)    RETURN $error$
(17)   END IF
(18)  END IF
(19)  left_operator.pushNext($left_{out}$)
(20)  right_operator.pushNext($right_{out}$)
(21)  $right_{pre}$ = $right_{out}$
(22) END FOR
```

Figure 7.6: Case 2: Method *pushNext* of $\ltimes^{-1}$

With a careful look on the input, it can be seen that the input is not a valid input of the reverse join since a tuple `<2, 28, 2, 26>` is missing in the second group. As a result, the reverse join operator has to examine all the input before it produces the first result.

The implementation for that case is given in Figure 7.6. It is similar to the implementation of case 1 - the differences are marked bold. In particular, the reverse join also has to group the input by the attributes of the left output schema (e.g., $c\_id, c\_age$), and additionally has to sort the input by the join attribute (e.g., $c\_age$) in ascending order (descending order is used if the comparison operator is $<$ or $\leq$) (Line 1 and 2). This way, the set of output tuples which is produced for the right output (e.g. table $supplier$) of the first group must be contained completely in the set of output tuples which is produced for the second group (Line 15). If this condition holds among all adjacent groups, then the input is valid; otherwise *error* should be returned (Line 16).

**Case 3:** If the join is not an equi-join and the join predicate is in the form of $!(a_i = a_j)$, then a blocking version of the reverse join operator is needed. The blocking version is implemented similar to the first case: the input tuples are grouped by the left output schema and the join operator checks if each group produces the same set of output tuples for right output.

**Case 4:** In order to process more complex join predicates, the algorithms introduced before must be combined. For example, to check the input of a reverse join operator with a conjunctive predicate like $a_i > a_j \wedge a_k < a_l$, the input must be grouped by $a_i$ and $a_k$ and the groups must be sorted ascending by $a_i$ and descending by $a_k$. Moreover, to check the input of a reverse join operator with a disjunctive join predicate the tuples of the input must be divided into different input groups each fulfilling one predicate element. E.g. for a join predicate like $a_i > a_j \vee a_k < a_l$ we divide the input into two groups - one which fulfills the predicate $a_i > a_j$ and another which fulfills the predicate $a_k < a_l$. If a input tuple fulfills more than one predicate the tuple is added to all corresponding input groups. Afterwards, each input groups are checked separately by the algorithms introduced for the previous cases. As each join predicate can be transformed into disjunctive normal form, we are able to process arbitrary reverse join operators[3].

### 7.9.2 Reverse Projection and Reverse Aggregation

In two special cases, the top down phase of RQP needs the blocking implementation of the reverse projection operator and the reverse aggregation operator.

**Case 1:** If the output schema of a reverse projection (or reverse aggregation) operator contains a *check* constraint in the form of $a_j < a_i < a_k$ or in the form of $a_j < a_i < c$ or in the form of $c < a_i < a_j$ (alternatively the predicate could use the $\leq$ instead of the $<$ operator), where $a_j$ and $a_k$ are attributes in the input schema and $a_i$ is an attribute in the output schema but not in the input schema and is bound by a *unique* or *primary-key* constraint, the data instantiation phase should use the blocking implementations of the operators.

**Example:** An example of this special case is a query like the following one:

```
SELECT b
FROM R
WHERE b<a and a<10
```

The relation $R$ consists of attributes $a$ and $b$; and $a$ is a *primary-key* attribute. If there are two input tuples `<7>` and `<8>`, then the reverse projection may generate `<9, 7>` for the first input tuple `<7>`. If that is the case, the reverse projection could not find an instantiation for the second tuple `<8>` because `<9,8>` is the only possible instantiation (as $b < a < 10$) but this instantiation violates the *primary-key* constraints imposed by the first output tuple `<9, 7>` on the attribute $a$. As a result, a blocking implementation is needed such that the reverse projection and the reverse aggregation operator consider all input tuples and generates the output in one batch. For the

[3]This thesis does not discuss the details of this algorithm.

example above, the reverse projection has to buffer all the input in order to produce the output `<8, 7>` and `<9, 8>`.

Figure 7.7 shows a generalized version of the function *instantiateData* which is used by the blocking implementation of both operators. This version takes a complete relation as input and returns an instantiation of the output for the complete input, as well as an array of numbers which represent the number of tuples, which have to be used for each input tuple in order to dissolve aggregations ($n[i]$ is the number of output tuples which have to be created for the $i$-th input tuple). Therefore the function guesses the right number of output tuples for each input tuple by creating all possible combinations of count values for all input tuples (Line 1 to 13). Afterwards the function tries to find an instantiation of the output for each possible combination of count values (Line 14 to 20). In case that the function finds an instantiation, it returns this instantiation and the current combination of count values. If none of the combinations is satisfiable $(NULL, NULL)$ is returned.

This function is more expensive than the simple *instantiateData* function, because of several reasons: One is that the constraint formula is more complex for the complete input and thus the model checker needs more time; another reason is that the trial-and-error has to be carried out for the complete input and thus the size of combinations grows exponential with the number of input tuples.

**Case 2:** If a reverse projection (or reverse aggregation) operator generates tuples which are processed by a reverse join operator (implied by a join dependency in the output schema) and its join predicate does not express the equality on a *primary-key* attribute of one of the output schemata, then blocking versions of the operators are needed during the data instantiation phase. Otherwise these operators may generate incorrect values which do not satisfy the join properties. The implementation of the blocking versions for these two operators in this special case needs similar algorithms as the blocking version of the reverse join operators in Section 7.9.1 in order to check the input. The algorithms can be adapted easily from that section and are not shown here.

Additional algorithms are needed in order to produce the output. First, the input must be grouped as described for the different join predicates in Section 7.9.1. Afterwards, the output generation is carried out for each group of the input separately in order to generate values which respect the join properties.

In the following we explain the output generation for join predicates which equal to those of case 2 in Section 7.9.1. For illustration purposes we use the following example.

**Example:** Consider the following query and the given input. The query is similar to the example query of case 2 in Section 7.9.1. However the join attribute $s\_age$ is not given by the input:

**instantiateData(Relation $r$, Schema $S^{OUT}$)**

**Output:**

//data instantiation

-instantiation $I$

//number of tuples to ungroup each tuple

-int[] $n$

```
//number of tuples to ungroup $r$
int[] $count$, $maxcount$
$i = 1$
//analyze each tuple $t \in r$
FOREACH $t$ in $r$
 IF $t$ includes COUNT of aggregation
  $count[i] = maxcount[i]$=COUNT value in $t$
 ELSE //USER_THREHOLD=1 if no aggregation
  $count[i]$=1; $maxcount[i]$=USER_THREHOLD
 $i = i + 1$
END FOR
//create combinations of count domains
$comb$=createCombinations($count$,$maxcount$)
FOREACH $n$ in $comb$ //$n$ is a $k-$array; $k$ is the cardinality of $r$
   //Create constraint formula $L$
   $L$=createConstraint($r$,$S^{OUT}$,$n$)
   $I$=decisionProcedure($L$)
   IF($I$!=NULL) RETURN ($I$,$n$)
END FOR //Trial-and-error
RETURN (NULL,NULL)
```

Figure 7.7: Case 1: Function *instantiateData*

```
SELECT c_id, c_age, s_id
FROM customer, supplier s
WHERE c_age>s_age
```

| c_id | c_age | s_id | |
|---|---|---|---|
| 1 | 27 | 1 | /*1st group*/ |
| 1 | 27 | 2 | |
| 2 | 28 | 1 | /*2nd group*/ |
| 2 | 28 | 2 | |
| 2 | 28 | 3 | |

First, the operator analyzes which join attributes are given by the input. If at least one join attribute is given by the input (e.g. $c\_age$), the input is grouped by the attributes of that output schema of the corresponding reverse join operator which contains that join attribute (e.g. $c\_id, c\_age$). Afterwards, the input groups are sorted by that join attribute (e.g. $c\_age$) ascending or descending depending on the relational operator of the join predicate ($>$, $geq$ or $<$, $leq$). If both join attributes are not given by the input, then the input is grouped by the attributes of the left output schema of the corresponding reverse join and sorted ascending by the cardinality of each group. If the value for the join attribute of the output schema we grouped by is not given by the input (e.g. $c\_age$), then the operator has to generate one distinct value per group where the values for all groups are sorted ascending or descending depending on the join predicate (e.g. 27, 28). However, in our example the attribute $c\_age$ is given by the input and thus no values have to be generated. Other values which must be generated for that output schema must be distinct for each group, too. If the value for the join attribute of the other output schema is not given by the input (e.g. $s\_age$), then the attribute values generated for the first group must be reused by the second group (e.g. we generate 25, 26 for the tuples with $s\_id = 1$ and $s\_id = 2$). Values generated for other attributes of that output schema (not in the join predicate) must be reused, too. New values must be generated for those tuples which are in the second but not in the first group (e.g. we generate 27 for the tuple with $s\_id = 3$). The new values for the join attribute have to be greater than the maximum value of the join attribute (e.g. $s\_age$) used in the first group in case that the join operator is $>$ or $geq$ or smaller than the minimum value of the join attribute in case that the join operator is $<$ or $leq$. Moreover, all generated join attribute values have to fulfill the join predicate. These steps have to be carried out for all adjacent groups.

The algorithms for other join predicates are straightforward. As the previous algorithm, these algorithms generate values for the join attributes in a similar way such that these values fulfill the properties of the particular join predicate shown in the different cases of Section 7.9.1.

### 7.9.3 Reverse Union

If both output schemata of a reverse union operator have a *primary-key* or a *unique* constraint on the same attribute $a_i$ and there is a *check* constraint on another attribute $a_j$ in the output schema, then a blocking version of the reverse union is needed in the top down data instantiation phase.

**Example:** An example can be shown by the query in Figure 5.1 (left side). Assume attribute $b$ is the *primary-key* attribute of both relation $R$ and $S$ and the two input tuples are `<6, 6>` and `<2, 6>`. Using the non-blocking version of the reverse union operator, the first tuple `<6, 6>` might be distributed to the relation $R$. Then, the second tuple `<2, 6>` cannot be distributed to relation $S$ because $a = 2$ cannot not fulfill the selection predicate $a > 5$. Alike, this tuple also could not be distributed to relation $R$ because of the *primary-key* constraint. Therefore, a blocking implementation of the reverse union operator is needed which buffers all the input and distributes `<6, 6>` to $S$ and `<2, 6>` to $R$.

Figure 7.8 shows the implementation of the blocking version of the reverse union operator. First, the method analyzes which tuple must be distributed to the left, right, and which tuple can be distributed to both outputs in a similar way as the non-blocking reverse union implementation (Line 1 to 20). Afterwards those tuples which can by distributed to both outputs ($both_{out}$) must be divided into two relations, one for each output (by method call *distribute*) (Line 22). The method *distribute* (not shown as algorithm) analyzes possible combinations to distribute tuples in $both_{out}$ to $left_{out}$ and $right_{out}$. In order to check if a combination satisfies the output schemata, two constraint formulas have to be constructed (one for $left_{out}$ and one for $right_{out}$). These formulas have to be checked by the model checker if they are satisfiable. If not, the next combination is tried. If no combination is found, the *distribute* method returns an error (Line 24), else the output is propagated to the left and right branch (Line 26, 27) as specified in the combination.

## 7.10 Processing Nested Queries

As mentioned in Section 6, SPQR uses the concept of nested iterations (sometimes called apply operators) which are known from traditional query processing [GLJ01], in a reverse way: The inner subquery can be thought of as a reverse query tree whose input is parameterized on values generated for correlation variables of the outer query.

**Example 1:** Assume that the $lineitem$ table (from Figure 4.3a)) has an extra column $l\_shipdate$. Then, the following nested query is processed reversely as follows:

```
$\cup^{-1}$.pushNext(Relation $r$)
 (1)  $left_{out}$ = $\emptyset$
 (2)  $right_{out}$ = $\emptyset$
 (3)  $both_{out}$ = $\emptyset$
 (4)  FOREACH $t$ in $r$
 (5)   //Create constraint formulas
 (6)   $L_{left}$=createConstraint($t$,$S^{OUT}_{left}$)
 (7)   $L_{right}$=createConstraint($t$,$S^{OUT}_{right}$)
 (8)   //call model checker
 (9)   IF(decisionProcedure($L_{left} \wedge L_{right}$)!=NULL)
(10)    $both_{out}$.add($t$)
(11)   //call model checker
(12)   ELSE IF(decisionProcedure($L_{left}$)!=NULL)
(13)    $left_{out}$.add($t$)
(14)   //call model checker
(15)   ELSE IF(decisionProcedure($L_{right}$)!=NULL)
(16)    $right_{out}$.add($t$)
(17)   ELSE
(18)    return $error$
(19)   END IF
(20)  END FOR
(21)  ($left_{out}$, $right_{out}$) =
(22)   distribute($both_{out}$, $left_{out}$, $right_{out}$)
(23)  IF($left_{out}$, $right_{out}$=(NULL,NULL))
(24)   return $error$
(25)  END IF
(26)  left_operator.pushNext($left_{out}$)
(27)  right_operator.pushNext($right_{out}$)
```

Figure 7.8: Method *pushNext* of $\cup^{-1}$ in special case

```
SELECT o_id FROM orders
WHERE orderdate IN
 (SELECT l_shipdate FROM lineitem
  WHERE l_oid = o_id)
```

First, the reverse query plan of the outer query is executed given an $RTable\ R$. The values generated for the bind variable $o\_orderdate$ and the correlation attribute $o\_id$ are used to initialize the input for the reverse query tree of the inner subquery. Processing nested queries is, thus, expensive: it has quadratic complexity with the size of the $RTable\ R$. Section 8 shows how almost all nested queries can be unnested for reverse query processing in order to improve performance.

In those cases where the bind or the correlation predicate does not express the equality, the reverse apply operator has to be implemented as blocking operator. This is obvious, as each nested RRA expression can be unnested, e.g. by using reverse join operators (as shown in Section 8). In the case that the reverse join operator uses a inequality predicate, it also must use blocking implementation. Thus, the algorithms for the blocking reverse apply operators are similar to the reverse join and not shown in this technical report. The only difference is that the reverse apply generates new input values for the inner subquery.

## 7.11 Optimization of Data Instantiation

The previous subsections showed that reverse query processing heavily relies on calls to a model checker. Unfortunately, those calls are expensive. Furthermore, the cost of a call grows with the length of the formula; in the worst case, the cost is exponential to the size of the formula. The remainder of this section lists techniques in order to reduce the number of calls to the model checker and reduce the size of the formulae (in particular, the number of variables in the formulae). The optimizations are illustrated using the example of Figure 4.2.

**Definition 7.1** *(Independent attribute:) An attribute $a$ is* independent *with regard to an output schema $S^{OUT}$ of an operator iff $S^{OUT}$ has no integrity constraints limiting the domain of $a$ and $a$ is not correlated with another attribute $a'$ (e.g. by $a > a'$) which is not independent.*

**Definition 7.2** *(Constrictive independent attribute:) An attribute $a$ is* constrictive independent, *if it is independent with regard to an output schema $S^{OUT}$ disregarding certain optimization-dependent integrity constraints.*

The following optimizations use these definitions:

### OP 1: Default-value Optimization

This optimization assigns a default (fixed) value to an independent attribute $a$. The default value assigned to $a$ depends on the type of the attribute. Attributes which use this optimization are not included in the constraint formula. An example attribute which could use this optimization is the attribute $l\_name$ of $lineitem$. This attribute could use a default value; e.g.,‘product‘.

### OP 2: Unique-value Optimization

This optimization assigns a *unique* increment counter value to a constrictive independent attribute $a$ which is only bound by *unique* or *primary-key* constraints. Here, the optimization-dependent integrity constraints which are disregarded in the definition of constrictive independent attribute are *unique* and *primary-key* constraints. Attributes which use this optimization are not included in the constraint formula. In the running example, values for the $l\_id$ attribute could be generated using this optimization. If another attribute $a'$ of the same schema exists which is correlated by equality (e.g. $a = a'$ from an equi-join) and $a'$ is an independent or a constrictive independent attribute which is only bound by *unique* or *primary-key* constraints, then attribute $a'$ is set to the same *unique* value as $a$ and constraints involving $a'$ need not be included in calls to the model checker either.

### OP 3: Single-value Optimization

This optimization can be applied for a constrictive independent attribute $a$ which is only bound by *check* constraints. An example of such an attribute is the attribute $l\_discount$ of $lineitem$. Such attributes are only included in a constraint formula the first time the top-down phase needs to instantiate a value for them. Afterwards, the instantiated value is reused.

### OP 4: Aggregation-value Optimization

This optimization can be applied for constrictive independent attributes $a$ which are only bound by an aggregation constraint. If the attribute $a$ is used in an aggregation function, e.g., $SUM(a)$ and a result value for the aggregation function is given, then different techniques to instantiate values for $a$ can be used. Some possibilities are shown below:

1. If $SUM(a)$ is an attribute in the operator’s input schema, $MIN(a)$ and $MAX(a)$ are not in the operator’s input schema, and $a$ has type float: Instantiate a value for $a$ by solving $a = SUM(a)/n$ with $n$ the number of tuples used to solve the aggregation constraint in the *instantiateData* function. In this case, no variables $a_1, a_2, \ldots, a_n$ need to be generated and used in the constraint formula passed to the model checker.

2. Same as (1), but $MIN(a)$ or $MAX(a)$ are in the operator's input schema, and $n \geq 3$: Use values for $MIN(a)$ or $MAX(a)$ once to instantiate $a$. Instantiate the other values for $a$ by solving $a = (SUM(a) - MIN(a) - MAX(a))/(n - 2)$.

3. Same as (1), but $a$ is of data type integer: Again, we can directly compute $a$ by solving $SUM(a) = n_1 \times a_1 + n_2 \times a_2$, where $a_1 = \lfloor sum(a)/n \rfloor$, $a_2 = \lceil sum(a)/n \rceil$, $n_1 = n - n_2$ and $n_2 = (SUM(a)\%n)$.

4. If $COUNT(a)$ is in the operator's input schema, $a$ can be set using the Default-value optimization (OP 1) because $a$ is independent in this case.

**OP 5: Count Heuristic**

Unlike the previous four optimizations, this optimization does not find instantiations. Instead, this optimization reduces the number of attempts for guessing the number of tuples ($n$ in Figure 7.2) to reverse process an aggregation by constraining the value of $n$. The heuristics for this purpose are shown below. The theoretical foundations for these heuristics are given in [RSSS94].

1. If $SUM(a)$ and $AVG(a)$ are attributes of the operator's input schema,
   then $n = SUM(a)/AVG(a)$.

2. If $SUM(a)$ and $MAX(a)$ are attributes of the operator's input schema,
   then $n \geq SUM(a)/MAX(a)$ (if $SUM(a) \geq 0$ and $MAX(a) \geq 0$; if $SUM(a) \leq 0$ and $MAX(a) \leq 0$ use $n \leq SUM(a)/MAX(a)$).

3. If $SUM(a)$ and $MIN(a)$ are attributes of the operator's input schema,
   then $n \leq SUM(a)/MIN(a)$ (if $SUM(a) \geq 0$ and $MIN(a) \geq 0$; if $SUM(a) \leq 0$ and $MIN(a) \leq 0$ use $n \geq SUM(a)/MIN(a)$).

**OP 6: Tolerance on precision**

As mentioned in Section 4, tolerances can be exploited in order to speed up model checking. That is, rather than, say, specifying $a = 100$, a more flexible constraint $90 \leq a \leq 110$ can be used. Of course, this optimization is only legal for certain applications. Our prototype, SPQR has a user-defined tolerance range which is set to 0 percent by default.

**OP 7: Memoization**

Another general optimization technique is to cache calls to the model checker. For example, $\pi^{-1}$ and $\chi^{-1}$ often solve similar constraints and carry out the same kind of guessing. In Figure 4.2, for instance, the results of guessing for the $\pi^{-1}$ operator can be re-used by the $\chi^{-1}$ operator.

Memoization at run-time has been studied in [HN96] for traditional query processing; that work is directly applicable in the RQP context.

Chapter 8

# Reverse Query Optimization

*Efficiency is doing better what is already being done.*

*– Peter Drucker, 1909-2005 –*

The job of the reverse query optimizer is to transform a reverse query tree into a more *efficient* reverse query tree (Figure 4.1). As part of such a rewrite, the input and output schemes need to be adjusted (Chapter 6). Depending on the application, different optimization goals can be of interest (e.g., running time and or database size). The RQP framework allows the integration of different query optimizers for different goals. In this work we present some first ideas on a RQP optimizer that tries to minimize the running time of reverse query processing. E.g., designing optimizers with other optimization goals (e.g., minimizing the size of the generated database instances) are beyond the scope of this thesis.

Just as in traditional query optimization, the reverse query optimizer rewrites a reverse query tree into an *equivalent* reverse query tree that satisfies a certain optimization goal. There are several possible definitions of equivalence:

**Definition 8.1** *(General RQP-equivalence:) Two Reverse Query Trees $T_1$ and $T_2$ are generally RQP equivalent for a Query Q iff for all $RTables$ R: $Q(T_1(R)) = Q(T_2(R)) = R$.*

**Definition 8.2** *(Result-equivalence:) Reverse Query Trees $T_1$ and $T_2$ are result-equivalent iff for all $RTables$ R: $T_1(R) = T_2(R)$.*

Traditional query optimization is based on result-equivalence: after a rewrite the same results should be produced. Query optimization for RQP can be much more aggressive and thus allows more rewrites. A rewrite is correct if the new reverse query tree generates a different database instance (in fact, it might even be desired); the only thing that matters is that the overall RQP correctness criterion (Section 4.1) is met. That is why general RQP-equivalence is used in the optimizer of the SPQR prototype.

## 8.1 Optimizer Design

The most expensive operators of RQP are $\pi^{-1}$ and $\chi^{-1}$ because these operators call the decision procedure of the model checker. The exact cost of these operators is difficult to estimate for a specific query because there are no robust cost models for model checkers; defining such cost models is a research topic in its own right in that community. Nevertheless, it is clear that the simpler and shorter the constraints, the better. One consequence is that it is important to minimize the number of $\pi^{-1}$ and $\chi^{-1}$ operators in a reverse query tree. Therefore, the canonical translation of a SQL query into an expression of the relational algebra [GMUW01] is already good because it results in at most one $\pi^{-1}$ operator at the root of the reverse query tree. Optimizations that add projections and group-by operations as devised for traditional query processing need not be applied.

In addition to $\pi^{-1}$ and $\chi^{-1}$, the execution of nested queries is expensive because it is $\mathcal{O}(n^2)$, with $n$ the size of the input (i.e., $RTable$ or intermediate result). Therefore, it is important to unnest queries. Rules that make it possible to fully unnest almost all queries are given in the next subsection. Furthermore, $\cup^{-1}$ and $-^{-1}$ operators can be expensive because they potentially involve calls to the model checker. As for $\pi^{-1}$ and $\chi^{-1}$ operators, therefore, the goal is to minimize the number of $\cup^{-1}$ and $-^{-1}$ operators in a reverse query plan. Again, the canonical translation of SQL queries is good enough in practice for this purpose.

All other operators are cheap. They are linear in the size of their inputs and do not require any calls to the model checker. In particular, the reverse equi-join that involves a *primary-key* or an attribute with a *unique* constraint is cheap. As a result, it is not important to carry out cost-based join ordering or worry about different reverse join methods. Again, the canonical relational algebra expression can be used for simple rewrites that eliminate unnecessary operators (e.g., $\sigma^{-1}$'s in certain cases) and/or simplifies the expressions in the reverse query tree. Such rewrites are presented in the last subsection of this chapter.

## 8.2 Query Unnesting

There are three rewrite rules that can be used to fully unnest most SQL queries. Only some queries that involve the same table in the outer and in the inner query cannot be unnested for RQP. This very aggressive unnesting is possible because of the relaxed equivalence criterion presented at the beginning of this chapter.

**Rule 1:** A subquery $Q_{inner1}$ nested inside a `NOT IN` operator can be removed if (1) the inner and outer queries refer to different tables and (2) no other subquery $Q_{inner2}$ exists which refers to the same table as $Q_{inner2}$ and is not nested inside `NOT IN`.

As a result, in the following example query $Q_1$ can be rewritten to $Q_2$:

$Q_1$ :
```
SELECT l_name FROM lineitem WHERE l_oid NOT IN
 (SELECT MAX(o_id) FROM orders
  GROUP BY o_orderdate);
```
$Q_2$ :
```
SELECT l_name FROM lineitem;
```

To check the correctness, consider an $RTable\ R$ with only one tuple: `<'productA>'`. $Q_1$ and $Q_2$ are obviously not result-equivalent with respect to $R$. RQP for $Q_1$ would generate at least one $lineitem$ tuple and one $orders$ tuple; in contrast, RQP for $Q_2$ would only generate a $lineitem$ tuple. The queries are general RQP-equivalent, however, because applying $Q_1$ to both database instance would return the required result; i.e. a single row with value `'productA'`.

**Rule 2:** An inner query in a nested query can be removed if (1) the columns used in the `SELECT` clause of the inner query are also used in the `SELECT` clause of the outer query and (2) the two queries are correlated by an equality predicate or by an `IN` predicate.

For example, the following Query $Q_3$ can be rewritten to Query $Q_4$:

$Q_3$ :
```
SELECT l_name, l_price FROM lineitem
WHERE price=(SELECT MIN(l_price) FROM lineitem)
```
$Q_4$ :
```
SELECT l_name, l_price FROM lineitem
```

**Rule 3:** If Rule 1 and Rule 2 are not applicable, all methods proposed in [GW87] to unnest queries for traditional query processing can be applied to reverse query processing, too.

The proof is straightforward because result-equivalence for traditional query processing implies general RQP-equivalence for reverse query processing. However, the optimizer has to take care of the operator order mentioned in [Klu80] in order to preserve the general RQP-equivalence.

## 8.3 Other Rewrites

At the begin of this chapter, we would like to mention the following (somewhat surprising) rewrite rule:

**Rule 4:** Remove reverse select operators from the reverse query plan.

Chapter 7 showed that this operator can be implemented using the identity function at run-time. Only a reverse select at the root of the plan must not be removed in order to make sure that its predicate is checked.

There are several other rewrite rules that help to simplify expressions (e.g., eliminate `LIKE` and other SQL functions from predicates). One such rewrite rule is:

**Rule 5:** A `LIKE` predicate can be rewritten as a equality predicate without the wildcards (e.g. %) if (1) the attributes included in the `LIKE` predicate are not given by the input and (2) these attributes do not have a *unique* constraint.

It is obvious, that the instantiated values for the rewritten equality predicate also fulfills the `LIKE` predicate; e.g., all values which fulfill $name = $‘$A$‘ also fulfill $name$ `LIKE` $\%A\%$.

Chapter 9

# Experiments

*No amount of experimentation can ever prove me right; a single experiment can prove me wrong.*

*– Albert Einstein, 1879-1955 –*

This chapter presents the results of performance experiments with our prototype system SPQR and the TPC-H benchmark [TPCb]. These experiments show the running times of reverse query processing and the size of the generated databases.

## 9.1 Experimental Environment

The SPQR system was implemented in Java (Java 1.4) and installed on a Linux AMD Opteron 2.2 GHz Server with 4 GB of main memory. In all experiments reported here, SPQR was configured to allow 0 percent tolerance; that is, OP 6 of Section 7.11 was disabled. As a backend database system, PostgreSQL 7.4.8 was used and installed on the same machine. As a decision procedure, Cogent [CKS05] was used. Cogent is a decision procedure that is publicly available and has been used in several projects world-wide. Cogent was written using the C programming language. For our purposes, it was configured to generate *error* if numerical overflows occurred.

The TPC-H benchmark is a decision support benchmark and consists of 22 business oriented queries and a database schema with eight tables. The queries have a high degree of complexity: all of them include at least one aggregate function with a complex formula, and many queries involve subqueries. Some queries (e.g., Q11) are parametrized and their results and running times depend on random settings of the parameters. The experiments were carried out in the following

| | 100M | | 1G | | 10G | |
|---|---|---|---|---|---|---|
| Query | $RTable$ | Generated | $RTable$ | Generated | $RTable$ | Generated |
| 1 | 4 | 600,572 | 4 | 6,001,215 | 4 | 59,986,052 |
| 2 | 44 | 220 | 460 | 2,300 | 4,667 | 23,335 |
| 3 | 1216 | 3,648 | 11,620 | 34,860 | 114,003 | 342,009 |
| 4 | 5 | 10,186 | 5 | 105,046 | 5 | 1,052,080 |
| 5 | 5 | 30 | 5 | 30 | 5 | 30 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 4 | 24 | 4 | 24 | 4 | 24 |
| 8 | 2 | 32 | 2 | 32 | 2 | 32 |
| 9 | 175 | 1,050 | 175 | 1,050 | 175 | 1,050 |
| 10 | 3767 | 15,068 | 37,967 | 151,868 | 381,105 | 1,524,420 |
| 11 | 2541 | 7,623 | 1,048 | 3,144 | 289,022 | 867,066 |
| 12 | 2 | 6,310 | 2 | 61,976 | 2 | 621,606 |
| 13 | 38 | 162,576 | 42 | 1,629,964 | 46 | 16,298,997 |
| 14 | 1 | 4 | 1 | 4 | 1 | 4 |
| 15 | 1 | 2 | 1 | 2 | 1 | 2 |
| 16 | 2762 | 23,264 | 18,314 | 236,500 | 27,840 | 2,372,678 |
| 17 | 1 | 3 | 1 | 3 | 1 | 3 |
| 18 | 5 | 15 | 57 | 171 | 624 | 1,871 |
| 19 | 1 | 2 | 1 | 2 | 1 | 2 |
| 20 | 21 | 105 | 204 | 1,020 | 1,968 | 9,840 |
| 21 | 47 | 2,325 | 411 | 20,705 | 4,009 | 197,240 |
| 22 | 7 | 1,282 | 7 | 12,768 | 7 | 127,828 |

Table 9.1: Size of Generated Databases and $RTable$ (rows)

way: First, a benchmark database was generated using the *dbgen* function as specified in the TPC-H benchmark. As scaling factors, we used 0.1 (100 MB database; 860K rows), 1 (1 GB; 8.6 million rows), and 10 (10 GB; 86 million rows). Then, the 22 queries were run, again as specified in the original TPC-H benchmark. The query results were then used as inputs (i.e., as $RTable$s) for reverse query processing of each of the 22 queries. We measured the size of the resulting database instance (as compared to the size of the original TPC-H database instance) and the running time of reverse query processing.

## 9.2 Size of Generated Databases

Table 9.1 shows the size of the databases generated by SPQR for all queries on the three scaling factors. For queries which include an explicit or implicit[1] `COUNT` value in $R$, the size of the generated database for different scaling factors depends on that `COUNT` value. For example, Q1 generates many tuples (600,572 tuples for SF=0.1) from a small $RTable$ $R$ because Q1 is an aggregate query where $R$ explicitly defines big `COUNT` values for each input tuple. For those queries which do not define a `COUNT` value, only a handful of tuples is generated because the trial-and-error phase starts from creating one output tuple per input tuple (e.g., Q6). In that case, the size of the generated database is independent from the scaling factor. As a summary, we see

[1]Implicit means that the `COUNT` value can be calculated by the optimization rule *OP 5* of Section 7.11.

| Query | RQP | QP | DB | MC | M-Invoke |
|---|---|---|---|---|---|
| 1 | 26:51 | 12:01 | 8:42 | 6:06 | 4 |
| 2 | 0:24 | $<$ 1ms | 0:21 | 0:02 | 44 |
| 3 | 19:20 | 0:14 | 0:11 | 18:55 | 1216 |
| 4 | 0:20 | 0:05 | 0:14 | $<$ 1ms | 5 |
| 5 | 0:12 | $<$ 1ms | $<$ 1ms | 0:11 | 10 |
| 6 | 0:02 | $<$ 1ms | $<$ 1ms | 0:1 | 2 |
| 7 | 0:10 | $<$ 1ms | 0:01 | 0:9 | 8 |
| 8 | 0:15 | $<$ 1ms | 0:02 | 0:13 | 12 |
| 9 | 4:23 | 0:02 | 0:03 | 4:17 | 175 |
| 10 | 56:33 | 0:42 | 0:37 | 55:13 | 3767 |
| 11 | 42:11 | 0:13 | 0:14 | 41:43 | 2541 |
| 12 | 7:25 | 0:16 | 0:11 | 6:57 | 3155 |
| 13 | 2:56 | 1:38 | 1:16 | $<$ 1ms | 21 |
| 14 | 0:08 | $<$ 1ms | 0:01 | 0:07 | 6 |
| 15 | 0:03 | $<$ 1ms | $<$ 1ms | 0:03 | 3 |
| 16 | 0:29 | 0:15 | 0:14 | $<$ 1ms | 0 |
| 17 | 0:02 | $<$ 1ms | $<$ 1ms | 0:01 | 2 |
| 18 | 0:01 | $<$ 1ms | $<$ 1ms | $<$ 1ms | 15 |
| 19 | 0:02 | $<$ 1ms | $<$ 1ms | 0:01 | 2 |
| 20 | 0:21 | $<$ 1ms | $<$ 1ms | 0:20 | 42 |
| 21 | 1:43 | 0:04 | 0:05 | 1:34 | 465 |
| 22 | 0:26 | 0:01 | 0:01 | 0:23 | 641 |

Table 9.2: Running Time (min:sec): SF=0.1

that the generated databases are already as small as possible. Huge databases are only generated by SPQR if the query result explicitly states the size.

## 9.3 Running Time (SF=0.1)

Table 9.2 shows the running times of RQP for the TPC-H benchmark with scaling factor 0.1. In the worst case, the running time is up to one hour (Query 10). However, most queries can be reverse processed in a few seconds. Table 9.2 also shows the cost break-down of reverse query processing. `QP` is the time spent processing tuples in SPQR (e.g., constructing constraint formulae and calls to the *pushNext* function). For all queries (except Q1), this time is below a minute. Q1 is an exception because it generates many tuples and a great deal of work is necessary in order to carry out the optimizations of Section 7.11 for each tuple. `DB` shows the time that is spent by PostgreSQL in order to generate new tuples (processing SQL `INSERT` statements through JDBC). Obviously, this time is proportional to the size of the database instance generated as part of RQP. The `MC` column shows the time spent by the decision procedure of the model checker. It can be seen that this time dominates the overall cost of RQP in most cases; in particular, it dominates the cost for the expensive queries (Q10 and Q11). This observation justifies the decision to focus all optimization efforts on calls to the decision procedure (Sections 7 and 8). `M-Invoke` shows the number of times the decision procedure is called. Comparing the `MC` and `M-Invoke` columns, it can be seen that the cost per call varies significantly. Obviously, the decision procedure needs more time for long constraints (e.g., Q10) than for simple constraints (e.g., Q22). We still have

| Query | 100M | 1G | 10G |
|---|---|---|---|
| 1 | 26:51 | 207:11 | 2054:19 |
| 2 | 0:24 | 0:47 | 4:02 |
| 3 | 19:20 | 183:49 | 1819:48 |
| 4 | 0:20 | 2:26 | 24:15 |
| 5 | 0:12 | 0:12 | 0:12 |
| 6 | 0:02 | 0:01 | 0:01 |
| 7 | 0:10 | 0:10 | 0:09 |
| 8 | 0:15 | 0:17 | 0:14 |
| 9 | 4:23 | 4:33 | 10:20 |
| 10 | 56:33 | 566:45 | 5639:13 |
| 11 | 42:11 | 18:15 | 4472:00 |
| 12 | 7:25 | 83:09 | 719:56 |
| 13 | 2:56 | 27:47 | 276:05 |
| 14 | 0:08 | 0:08 | 0:15 |
| 15 | 0:03 | 0:03 | 0:04 |
| 16 | 0:29 | 4:04 | 36:37 |
| 17 | 0:02 | 0:02 | 0:08 |
| 18 | 0:01 | 0:10 | 1:54 |
| 19 | 0:02 | 0:02 | 0:02 |
| 20 | 0:21 | 3:24 | 32:27 |
| 21 | 1:43 | 14:44 | 140:47 |
| 22 | 0:26 | 4:08 | 42:00 |

Table 9.3: Running (min:sec): Vary SF

not found a way to predict the cost per call and we are hoping for progress in this matter from the model checking research community.

We also measured the number of attempts each TPC-H query needed for guessing the number of tuples in aggregations (Section 7). These results are not shown in Table 9.2, but the results are encouraging: in fact, none of the 22 required any trial-and-error. The reason is that the optimizations proposed in Section 7.11 effectively made it possible to pre-compute the right number of tuples for all TPC-H queries.

## 9.4 Running Time: Varying SF

Table 9.3 shows the running times of reverse processing the 22 TPC-H queries for the three different scaling factors. In some cases, due to the nature of the queries, the running times (as the size of the generated databases, Table 9.1) is independent of the scaling factor; example queries are Q5 and Q6. For all those queries, for which the running times were higher for a larger scaling factor, the running time increased linearly. Examples are queries Q10 and Q21. Again, these results are encouraging because they show that RQP potentially scales linearly and that even large test databases can be generated using RQP. Note that Q11 has a parameter that is set randomly; this observation explains the anomaly that the running time for SF=0.1 is higher than for SF=1 for that query.

# Chapter 10

# Related Work

*The work of the individual still remains the spark that moves mankind ahead even more than teamwork.*

*– Igor Sikorsky, 1889-1972 –*

To the best of our knowledge, there has not been any previous work on reverse query processing. The closest related work is the work on model checking which has a similar goal: find instantiations of logical expressions. Consequently, we use the results of that research community in our design. However, the model checking community has not addressed issues involving SQL or database applications. In addition, that community has not addressed any scalability issues that arise if millions of tuples need to be generated as for the TPC-H benchmark. In order to provide scalability, our design adopted techniques from traditional query processing; e.g., [HFLP89; Gra93]. All that work is orthogonal to our work.

As mentioned in Chapter 2.2 there has been significant related work in the area of generating test databases. [MR86] shows how functional dependencies can be processed for generating test databases. The bottom-up phase of RQP (Section 6) makes use of the findings of the work in [Klu80] and extends it for the complete SQL specification. Likewise, other work on the generation of test databases (e.g., [NML93; CDF$^+$04]) focuses on one aspect only and falls short on most other aspects of RQP. [IWL83] discusses a similar problem statement as RQP but only applicable to a very restricted set of relational expressions. There has also been work on efficient algorithms and frameworks to produce large amounts of test data for a given statistical distribution [GSE$^+$94; BC05]. In the other potential application areas of RQP (e.g., sampling), to the best of our knowledge, nobody has tried yet to apply techniques such as RQP.

# Part III

# Applications of Reverse Query Processing

The first two chapters of this part discuss the extensions of RQP to support two further applications: Chapter 11 presents the extensions that are necessary to support the testing of OLTP applications where RQP gets set of queries and results as input to generate a test database. Chapter 12 then describes the extensions of RQP to support the testing of a query language where we need to be able to verify the actual query result that is returned by executing a test query on a particular test database. These techniques are currently used in an industrial environment for the testing of the Query Processing Functionality of the new ADO.Net Entity Framework of Microsoft (Redmond, USA). Finally, in the last chapter of this part we sketch some other applications of RQP which include the debugging of SQL queries and the testing of the confidentiality of data that comes from different views.

Chapter 11

# Functional Testing of OLTP Applications

*If I have seen further, it is by standing on the shoulders of giants.*

*– Isaac Newton, 1643-1727 –*

In contrast to OLAP applications which implement reports that read a huge amount of correlated data from the database, OLTP applications implement use cases which execute a sequence of actions whereas each action usually reads or updates only a small set of tuples in the database. As an example, think of an online library. One potential use case of such an application is that a user wants to borrow a book. The sequence of actions which is implemented by that use case could be as follows:

1. The user enters the ISBN of the book (where the ISBN is unique for each book of the library).
2. The system shows the details of that book.
    - Exception 1: The book is borrowed by another user. The system denies the request.
    - Exception 2: The book belongs to the closed stack of the library. The system denies the request.
3. The user enters personal data (username, password) and confirms that she wants to borrow the book.
4. The system checks the user data and updates the database.
    - Exception 3: The user has entered an incorrect username or password. The system denies the request.
    - Exception 4: There are charges on the user account that exceed a certain limit. The system denies the request.

Functional testing the implementation of such a use case means that we have to check the conformance of the implementation with the specification of the functionality [Bin99] (i.e., the use case). Consequently, we need to create a set of test cases to test the correctness of the different execution paths of a use case. In the following we show some test cases which can be used for the functional testing of the implementation of the use case shown above:

- *Test Case 1:* The user wants to borrow a book with a particular ISBN that is already borrowed by another user.
- *Test Case 2:* The user wants to borrow a book with a particular ISBN but the book belongs to the closed stack.
- *Test Case 3:* The user wants to borrow a book with a particular ISBN and enters an incorrect username or password.
- *Test Case 4:* The user wants to borrow a book with a particular ISBN but there are charges on her account that exceed a certain limit.
- *Test Case 5:* The user borrows a book with a particular ISBN successfully.

In order to execute all these test cases, one or more test databases need to be created which comprise different types of books (i.e., books which are already borrowed by another user or not, and books which belong to the closed stack and other books which do not) and different user accounts (i.e., user accounts with and without charges which exceed a certain limit). For example, in order to execute *Test Case 2* the database should include a book which belongs to the closed stack.

Currently, there are a number of commercial and academic tools available (e.g., [IBM; DTM; dbM; BC05; SP04; HTW06; NML93; CDF$^+$04]) which generate test databases for a given database schema. Beside the database schema, some tools also support the input of the table sizes, data repositories and additional constraints used for data instantiation (e.g., statistical distributions of individual attributes, value ranges). Unfortunately, all the aforementioned tools suffer from the problem that the generated test databases often do not comprise the data characteristics sufficient to execute a given set of test cases. The reason is that these tools take constraints on the complete database state as input (e.g., table sizes and value distributions of individual attributes) which are not suitable to express the needs of the individual test cases. Consequently, the generated test databases are usually inadequate to support the execution of all given test cases.

A solution to tackle this problem was shown in Part II which discusses a new technique called Reverse Query Processing (or RQP for short). The idea of RQP is to let the user constrain the database state by using one SQL query $Q$ and a expected result $R$ of that query. The RQP processor SPQR then generates a set of `INSERT` statements which create a test database $D$ for a given schema $S$ (including integrity constraints) such that $D$ returns the expected result $R$ for that query; i.e., $Q(D) = R$. The main application of RQP is the testing of the reporting functionality of an

OLAP application. In order to create a set of test cases and the corresponding test databases, we suggested to extract the SQL queries which are defined by the individual reports and to manually create one or several expected results for each of these reports. A SQL query which implements a report and a sample result of that report together represent a test case which can directly be used as input of the RQP processor to generate a corresponding test database for that test case. For the testing of the OLAP application, the SQL query (i.e., the report) which is defined by a test case is executed on the generated test database and the actual result is compared the expected result for verification.

However, one SQL `SELECT` query and one expected result are usually not sufficient to specify a test database that is adequate to execute a test case of an OLTP application. The reason is that most test cases of an OLTP application need to *read* or *update* different tuples in the database that are not necessarily correlated. Therefore, in order to specify the relevant values of the tuples that are read or updated by a particular test case, in this chapter we suggest that a tester uses SQL as a database generation language: i.e., the tester specifies the test database for one test case by *manually* creating a set of SQL `SELECT` queries and their expected results (called test database specification). A test database which returns these expected results for all the given SQL `SELECT` queries enables the execution of a particular test case of an OLTP application. Compared to the approach discussed in Part II, we do not provide a formal method to derive the SQL `SELECT` queries for the test database specification from the code of the OLTP application or from the test cases because we think that using SQL as a database generation language is intuitive. Consequently, the SQL `SELECT` queries in the test database specification are independent from the SQL statements implemented by the OLTP application (i.e., the `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements).

For example, if we want to generate a test database for *Test Case 4* above, the test database needs to comprise a book with a particular ISBN which does not belong to the closed stack (i.e., the attribute $b\_closedstack$ must have the value '$false$') and a user whose charges exceed a certain limit (e.g. \$20)[1]. The desirable database state, can be specified by multiple queries and the corresponding expected query results (e.g., the queries and expected results shown in the following example). By doing so, the tester can focus on the data that is relevant for *Test Case 4* (e.g., the values for $b\_isbn$ and $b\_closedstack$ specified by $Q_1$ and $R_1$) and she does not have to take care of the irrelevant data (e.g., the values for $b\_price$ and $b\_title$). Unfortunately, RQP is not capable to support multiple queries and the corresponding expected results as input.

[1]The database schema for all examples in this chapter is shown in Figure 11.3 (a) on Page 96.

$Q_1$ : `SELECT b_closedstack FROM book`
`WHERE b_isbn='0130402648'`
$R_1$ : `{<'false'>}`
$Q_2$ : `SELECT u_pasword, u_charges FROM user`
`WHERE u_name='test'`
$R_2$ : `{<test, 20.0>}`

Consequently, in this chapter we study the problem of Multi-RQP (or MRQP for short). Unlike RQP, MRQP gets a *set* of SQL `SELECT` queries and the *corresponding expected query results* as input and tries to generate one test database that returns the expected results for all the given queries. However, we can show that MRQP is undecidable for arbitrary SQL `SELECT` queries. Thus, as we suggest that the tester creates the queries manually to specify the test database, we can restrict the classes of queries to be supported by MRQP so that MRQP becomes decidable. Moreover, when defining the restricted classes of input queries that are to be supported by MRQP, we have to make sure that the tester can still specify any database instance any test database for a given OLTP application by only using these restricted query classes.

**Contributions:** The contributions of this chapter can be summarized as follows: (1) We formulate the problem statement of MRQP and prove that it is undecidable for arbitrary SQL `SELECT` queries. (2) In order to generate test databases for a test case of an OTLP application, we propose a new database generation language called MSQL. MSQL is a pure subset of SQL. Using MSQL a tester can manually create a set of queries and the corresponding expected results to specify the test database state for one test case. MSQL is carefully designed: Using MSQL the tester can easily formulate queries that satisfy certain restrictions so that MRQP on these queries is decidable and can be solved efficiently while the tester can still specify any test database for a given schema. (3) Using the specified queries and expected results, we discuss how a test database can be automatically generated by MRQP which is adequate to support the execution of a particular test case. (4) As a last contribution we present an algorithm which reduces the number of test databases for all test cases (MRQP initially generates one test database per test case). As a result many test cases can use the same test database. Consequently, the test cases can be executed more efficiently [CAA$^+$04; HKL07] and the management of the test databases becomes easier.

**Outline:** The remainder of this chapter is organized as follows: Section 11.1 discusses the problem statement of MRQP and define some general restrictions on the input queries of MRQP such that MRQP becomes decidable under the assumption that RQP is decidable for each single query. Section 11.2 then introduces the new test database generation language MSQL for which we can easily check whether a set of given queries fulfills the aforementioned restrictions or not. Moreover, we also show a complete example of MRQP using MSQL and discuss some further extensions of MSQL. Section 11.3 describes the algorithm which reduces the number of test databases for all test cases. Finally, Section 11.4 discusses related work.

## 11.1 MRQP Overview

In this section we first study the decidability of MRQP. Therefore, we present the general problem statement of MRQP and show that MRQP is undecidable for arbitrary SQL queries. Afterwards, we introduce some restrictions on the input queries of MRQP such that MRQP becomes decidable (under the assumption that RQP is decidable for each individual query and its expected query result). Finally, we illustrate a procedure which solves MRQP under these restrictions.

### 11.1.1 Problem Statement and Decidability

As mentioned before, this chapter addresses the following problem: Given a set of arbitrary SQL `SELECT` queries $Q = \{Q_1, ..., Q_n\}$, a set of expected results $R = \{R_1, ..., R_n\}$ of these queries, and the database schema $S$ of a relational database (including integrity constraints), find a database instance $D$ so that

$$R_i = Q_i(D)$$

for all $1 \leq i \leq n$ and $D$ is compliant with $S$ and its integrity constraints. There may exist many different database instances $D$ that satisfy these criteria. In this chapter, it is the goal to find one viable database instance.

The decision problem (based on the problem statement above) which asks whether a database instance $D$ exists or not that satisfies the schema $S$ and returns $R_i = Q_i(D)$ for all $1 \leq i \leq n$ is thus called the *MRQP decision problem*. Obviously, the MRQP decision problem cannot be decidable because RQP is not decidable for arbitrary SQL queries either (see Section 4.1).

### 11.1.2 MRQP Restrictions

As already mentioned in the introduction of this chapter, we suggest that a user manually creates a set of `SELECT` queries $Q = \{Q_1, Q_2, \ldots, Q_n\}$ and the expected results of these queries $R = \{R_1, R_2, \ldots, R_n\}$ in order to specify the test database for one test case. By doing so, we can restrict the input queries in $Q$ to be able generate a test database for many practical situations.

Consequently, in this section we first introduce a restriction on the query set $Q$ which requires that $Q$ must be *RQP-disjoint*. Under that restriction and the assumption that RQP is decidable for each individual SQL query $Q_i \in Q$, MRQP can be solved efficiently by first generating one individual test database for each query $Q_i \in Q$ using RQP and then taking the union over all these individual test databases to create the final test database that returns the expected result defined in $R$ for all queries in $Q$.

Despite that restriction, a tester can still specify any test database instance for a given database schema of an OLTP application. However, in some cases it is cumbersome for the tester to define such an RQP-disjoint query set $Q$. Therefore, we introduce a relaxation of that restriction which enables a tester to specify the test database in a more elegant way by creating *query refinements* for the individual queries $Q_i \in Q$. Moreover, in this section we also show how to generate a test database under that relaxation.

As already mentioned, we do not present a method how to derive the test database specification for one test case (i.e., an RQP-disjoint query set and some query refinements) from the code of the application or the test cases because we believe that it is intuitive to use SQL as a database generation language. Moreover, compared to the manual creation of a test database, using our approach the tester only needs to specify the relevant data for a test case and she does not have to take care of the irrelevant data.

### RQP-disjoint Queries

In order to solve MRQP we require that the input query set $Q$ is *RQP-disjoint*.

**Definition 11.1** *(RQP-disjoint Queries:) A set of queries $Q$ is RQP-disjoint iff all possible pairs ($Q_j$, $Q_k$) with $j \neq k$ are RQP-disjoint. Two queries $Q_j$ and $Q_k$ in $Q$ with $j \neq k$ are RQP-disjoint, iff the view specified by query $Q_j$ is update independent from any update (i.e.,* `INSERT` *statement) that could be generated by RQP for the query $Q_k$ and any possible expected result $R_k$ of that query and vice versa.*

As an example, look at the following two SQL queries $Q_1$ and $Q_2$ and the corresponding expected results $R_1$ and $R_2$ which specify the test database for *Test Case 3* in Section 1. This test case requires a test database which comprises a book with a particular ISBN that does not belong to the closed stack and a user with a distinct user name and a password which is different from a given password (that is used as input value for the test case). The two queries $Q_1$ and $Q_2$ are RQP-disjoint because $Q_2$ is update independent from any `INSERT` statement that could be generated by an RQP processor for $Q_1$ any expected result of that query (e.g., $Q_1$ is update independent from the `INSERT` statement $I_1$ which is generated for $Q_1$ and $R_1$ by an RQP processor) and vice versa (e.g., the view defined by $Q_1$ is update independent from $I_2$).

$Q_1$ :
```
SELECT b_closedstack FROM book
WHERE b_isbn='0201485419'
```
$R_1$ :
```
{<'false'>}
```
$I_1$ :
```
INSERT INTO book
(b_id, b_title, b_price, b_isbn, b_closedstack)
VALUES (1, 'TitleB', 100.0, '0201485419', 'false')
```
$Q_2$ :
```
SELECT COUNT(*) FROM user
WHERE u_name='test' AND u_password!='test'
```
$R_2$ :
```
{<1>}
```
$I_2$ :
```
INSERT INTO user
(u_id, u_name, u_password, u_charges)
VALUES (1, 'test', 'test1', 0.0)
```

If the queries in $Q$ are RQP-disjoint, then we can generate a test database by calling the RQP processor separately for each query and the corresponding expected result (i.e., $RQP(Q_1, R_1, S) = D_1$, ..., $RQP(Q_n, R_n, S) = D_n$)[2]. Afterwards, we take the union of all the individual test databases to create the final test database (i.e., $D = D_1 \cup ... \cup D_n$)[3]. Continuing the example above: In order to generate a test database for the two RQP-disjoint queries $Q_1$ and $Q_2$ and the two expected results $R_1$ and $R_2$, we first generate two individual databases $D_1$ and $D_2$. Consequently, the test database $D_1$ comprises one book with the given ISBN and the value specified for the attribute $b\_closedstack$ (i.e., $D_1$ is created by $I_1$) and test database $D_2$ comprises the user account with the given username and a password which is not equal to the input value '$test$' (i.e., $D_2$ is created by $I_2$). Subsequently, the final test database $D$ is $D = D_1 \cup D_2$.

Using an RQP-disjoint query set as input of MRQP, the user can specify any database instance for a given schema $S$. In order to show that this is possible, we assume that a tester creates one query per table which reads all tuples (e.g., `SELECT * FROM orders`) and the expected results of these queries. Using these queries and the expected results the tester can obviously control all attribute values individually for each tuple in every table of the database schema $S$ and thus specify any database instance.

In order to make sure that the final database $D$ which is generated for an RQP-disjoint query set fulfills the *primary-key* and *unique* constraints in the database schema $S$, MRQP has to make sure that the individual RQP calls assign unique values to the attributes in $S$ that are bound by such a constraint for all queries in $Q$ and the corresponding expected results in $R$. However, if some expected results in $R$ define values that violate the *primary-key* or *unique* constraint of an attribute in $S$, then MRQP will return an *error* if the union of the individual databases (i.e., $D = D_1 \cup D_2$) violates such a constraint. This error handling can be implemented using standard database techniques for checking integrity constraints. For example, assume that the tester specifies two queries and expected results where each query and its expected query result defines a user tuple

[2] In this chapter we call the RQP processor as an external function which takes a query $Q$, an expected result $R$, and a database schema $S$ as input and generates a database $D$ which satisfies $S$ and returns $Q(D) = R$.

[3] The $\cup$ operator here creates the union over all tables of the database schema $S$.

with the same username (i.e., the attribute $u\_name$) but with different passwords (i.e., the attribute $u\_password$). Creating the union of the two test databases that are generated for these two queries and expected results would return an error because the attribute $u\_name$ has a *unique* constraint in the database schema $S$ (see Figure 11.3 (a)). One way for the user to avoid these kinds of errors will be discussed in Section 11.2.4.

### Query Refinements

Using only an RQP-disjoint query set to specify the intended test database for a test case can sometimes be cumbersome for the tester. For example, assume the tester wants to specify a test database (for a test case not shown in Section 1) which should comprise five books with a total sum of prices which is $\$1000$ while one of these books should have the price $\$100$ and the title '$TitleA$'.

Unfortunately, there is no elegant way to specify such a test database by using only an RQP-disjoint query set: (1) The first possibility is that the tester specifies one query (i.e, `SELECT b_price, b_title FROM book`) and defines an expected result which holds the values for the attributes $b\_price$ and $b\_title$ of all books (while the tester has to manually take care that the total sum is $\$1000$ and she also has to define the titles for four out of five books that are not relevant for the test case). (2) Another possibility is that the tester specifies two individual SQL queries while one query specifies the one book which has the price of $\$100$ and the title '$TitleA$' (as shown by the query $Q_1$ and the expected result $R_1$ in the following example) and the other query specifies the remaining four books (as shown by query $Q_2$ and the expected result $R_2$ in the following example). However, in that case the tester has to manually adjust the query $Q_2$ and the expected result $R_2$ so that the total sum for the four remaining books is $\$900$ and none of these books uses the same ISBN as the book with the price of $\$100$ (i.e., the selection predicate of $Q_2$ must be $b\_isbn! =$ '0130402648').

```
Q1 : SELECT b_price, b_title FROM book
     WHERE b_isbn='0130402648'
R1 : {<100.0, 'TitleA'>}
Q2 : SELECT SUM(b_price), COUNT(*) FROM book
     WHERE b_isbn!='0130402648'
R2 : {<900.0, 4>}
```

A more elegant solution to that problem is that in addition to the RQP-disjoint set of queries $Q$ and the expected results $R$ we allow the user to define at maximum one *query refinement* for each query $Q_i \in Q$. The intuition is that a query refinement $F_i$ for a query $Q_i$ gives more information (attribute values) about a subset of tuples that are read by $Q_i$. That is, a query refinement $F_i$ refines a query $Q_i$. In the following we give a more formal definition and show how MRQP can generate the test database if some queries $Q_i \in Q$ are refined by a query refinement.

**Definition 11.2** *(Query Refinement:) A query refinement $F_i$ for a query $Q_i \in Q$ is a set of RQP-disjoint queries $F_i = \{F_{i1}, \ldots, F_{in}\}$ plus the expected results $RF_i$ for each query in $F_i$; i.e., $RF_i = \{RF_{i1}, \ldots, RF_{in}\}$ where $Q_i$ is update dependent (=opposite of update independent) of all `INSERT` statements that could be generated by RQP for any query $F_{ij} \in F_i$ and an arbitrary expected result $RF_{ij} \in RF_i$ of that query. Moreover, $baseAttr(R_i) \subseteq baseAttr(RF_{ij})$ must hold for all $RF_{ij} \in RF_i$ ($R_i$ is the expected result of $Q_i$ and $baseAttr(R_i)$ is a function that extracts the names of the attributes in the database schema $S$ that participate in the expected result $R_i$). Furthermore, for each query $F_{ij} \in F_i$ the user can recursively specify further query refinements.*

A simple query refinement for the example above is shown by the following query $Q_1$ and the refinement given by $F_1 = \{F_{11}\}$. While $Q_1$ specifies the total sum of prices for all books, $F_1$ specifies the price and the title of one book with a particular ISBN number. Obviously, $Q_1$ is update dependent from any `INSERT` statement that could be generated by RQP for each query in $F_1$ and some arbitrary expected results (e.g., $Q_1$ is update dependent from the `INSERT` statement $IF_{11}$ which is generated by RQP for the expected result $RF_{11}$ and the query $F_{11}$). Moreover, $baseAttr(R_1) = \{b\_price\}$ is a subset of $baseAttr(RF_{11}) = \{b\_price, b\_title\}$. Thus, $F_1 = \{F_{11}\}$ is a query refinement for query $Q_1$.

$Q_1$ :
```
SELECT SUM(b_price), COUNT(*) FROM book
```
$R_1$ :
```
{<1000.0, 5>}
```
$F_{11}$ :
```
SELECT b_price, b_title FROM book
WHERE b_isbn='0130402648'
```
$RF_{11}$ :
```
{<100, 'TitleA'>}
```
$IF_{11}$ :
```
INSERT INTO book
(b_id, b_title, b_price, b_isbn, b_closedstack)
VALUES (1, 'TitleA', 100.0, '0130402648', 'false')
```

In the following we illustrate how MRQP can generate a test database for a query $Q_i$ of an RQP-disjoint query set $Q$ which is refined by a query refinement $F_i$ and its expected results $RF_i$. A general solution how to generate a test database for a RQP-disjoint query set $Q$ where some queries $Q_i \in Q$ can be recursively refined by a query refinement is shown in the next Section 11.1.3. The idea presented here is similar to the one shown for an RQP-disjoint query set. MRQP first generates one test database for $Q_i$ and another one for $F_i$ by calling an RQP processor individually for $Q_i$ and $F_i$ and taking the union of both test databases. However, before the test database for the query $Q_i$ and its expected result $R_i$ can be generated by an RQP processor, MRQP has to adjust $Q_i$ and $R_i$ w.r.t. the query refinement $F_i$ and its expected results $RF_i$. The details of this process are described in the sequel.

Firstly, MRQP generates a test database $DF_i$ for the query refinement $F_i$ of query $Q_i$ and the expected results $RF_i$ of the refinement $F_i$ as described in Section 11.1.2 for any RQP-disjoint set of queries. Afterwards, MRQP *adjusts* the expected result $R_i$ of the query $Q_i$ which is refined by $F_i$ with respect to the generated test database $DF_i$ by executing $R'_i = R_i \ominus Q_i(DF_i)$. The operator $\ominus$ is called the *Adjust* operator and its implementation depends on the type of query $Q_i$. In general, the $\ominus$ operator "removes" those tuples from the expected result $R_i$ that are already specified by the queries in the refinement $F_i$ and the expected results $RF_i$ and thus do not have to be generated for the query $Q_i$ and the expected result $R_i$ anymore. Consequently, in some cases the *Adjust* operator $\ominus$ can be implemented by the relational minus operator for bags (i.e., $-$). If an expected result $R_i$ is not *adjustable* then this operator returns an error. Moreover, in addition to the expected result $R_i$, we also have to adjust the query $Q_i$ (which results in $Q'_i$) so that RQP generates no tuples for $Q'_i$ and the adjusted expected result $R'_i$ that would be returned by any query in the refinement $F_i$. A detailed description of the implementation of the *Adjust* operator and the function which adjusts the query $Q_i$ will be given in Section 11.2 for all query classes supported in in the database generation language MSQL. Subsequently, MRQP generates a test database $D'_i$ for the adjusted query $Q'_i$ and the adjusted expected result $R'_i$ by calling the RQP processor. The final test database $D_i$ for the query $Q_i$ and the query refinement $F_i$ is created by taking the union of $D'_i$ and $DF_i$; i.e., $D_i = D'_i \cup DF_i$.

For instance, in order to generate a test database $DF_1$ for $Q_1$ and $F_1$ in the example above, MRQP first generates a test database $D_F$ for the query refinement $F_1 = \{F_{11}\}$ and the expected results $RF_1 = \{RF_{11}\}$ as discussed in Section 11.1.2; e.g., a minimal test database $DF_1$ comprises one book with the given values (i.e., one book with the values specified for the attributes $b\_price$, $b\_title$ and $b\_isbn$ by $F_{11}$ and $RF_{11}$). Afterwards, the expected result $R_1$ is adjusted by executing $R'_1 = R_1 \ominus Q_1(DF_1) = \{< 900.0, 4 >\}$ and the query $Q_1$ is adjusted, too, which returns the adjusted query $Q'_1$:

$Q'_1$ :
```
SELECT SUM(b_price), COUNT(*) FROM book
WHERE b_isbn!='0130402648'
```

Subsequently, we generate the test database $D'_1$ for the adjusted query $Q'_1$ and the adjusted expected result $R'_1$ (i.e., four books with the total sum \$900 that have an ISBN value other than '0130402648'). The final test database $D_1$ that returns $R_1$ for $Q_1$ and $RF_{11}$ for $F_{11}$ is created by taking the union of $D'_1$ and $DF_1$; i.e., $D_1 = D'_1 \cup DF_1$.

### 11.1.3 MRQP Solution

The function *MRQP* which is shown in Figure 11.1 implements a general procedure for MRQP which generates a test database for a RQP-disjoint query set $Q$ where some queries $Q_i \in Q$ can be recursively refined by a query refinement.

The function *MRQP* first creates an empty database $D$ for the query set $Q$ (Line 1). Afterwards, the function checks for each query $Q_i \in Q$ if there exists a query refinement $F_i$ for that query

```
**MRQP**(**Queries** $Q$, **Results** $R$, **Schema** $S$, **Query Refinements** ($F_i$, $RF_i$))
**Output:** database $D$
(1)  $D$=$\emptyset$ //Generate an empty DB
(2)  FOR EACH Query $Q_i$ in $Q$
(3)   $R_i$= $R$.get($i$) //Extract expected result
(4)   $DF_i$=$\emptyset$
(5)   IF($Q_i$ has a Query Refinement ($F_i$, $RF_i$))
(6)    $DF_i$=MRQP($F_i, RF_i, S$) //Generate DB for $F_i$
(7)    $R_i$=$R_i \ominus Q_i(DF_i)$ //Adjust result
(8)    $Q_i$=$AdjustQuery(Q_i, F_i)$ //Adjust query
(9)   END IF
(10)  //Try to create union
(11)  IF($D$=$D\cup$RQP($Q_i$,$R_i$,$S$)$\cup DF_i$ returns ERROR)
(12)   RETURN ERROR
(13)  END IF
(14) END FOR
(15) RETURN $D$
```

Figure 11.1: Function *MRQP*

(Line 5). If yes, then this function generates a test database $DF_i$ for that query refinement and the expected results $RF_i$ by calling *MRQP* recursively (Line 6). Subsequently, the function adjusts the expected result $R_i$ and the query $Q_i$ w.r.t. $DF_i$ (Line 7-8). Afterwards, the function *MRQP* creates the new test database $D$ as a union of the existing test database $D$, the test database that is created for the adjusted query $Q_i$ and the adjusted expected result $R_i$, and the test database $DF_i$ generated for a potential query refinement $F_i$ (Line 11). If the union does not satisfy the database schema $S$ because some *primary-key* or *unique* constraints in $S$ are violated, then an error is returned (Line 12). If all queries in $Q$ are processed the final test database $D$ for $Q$ and $R$ is returned.

## 11.2 The DB Generation Language MSQL

As discussed in the Section 11.1.2, we allow a tester to specify a test database which is adequate to execute a particular test case by manually creating a set of RQP-disjoint queries $Q$ and at maximum one query refinement $F_i$ for each query $Q_i \in Q$. In order to support the tester in formulating an RQP-disjoint query set $Q$ and some query refinements, we have to decide whether $Q$ is RQP-disjoint or not and whether a query refinement $F_i$ refines a query $Q_i \in Q$ or not. However, update independence in general is undecidable [LS93], which means that it is also undecidable whether a set of arbitrary queries $Q$ is RQP-disjoint or not and it is also undecidable whether a given set of arbitrary queries in $F_i$ refines a query $Q_i \in Q$ or not.

Consequently, in this section we define a database generation language called MSQL (based on SQL) and a Reverse Relational Algebra called MRRA which is used in MRQP to generate the test database (based on the Reverse Relational Algebra RRA of RQP in Section 5)[4]. For a query set $Q$ and some query refinements for the queries $Q_i \in Q$ that are formulated in MSQL, we can easily check whether the query set $Q$ is RQP-disjoint and if a query refinement $F_i$ refines a query $Q_i \in Q$ (if the MRRA is used to reverse process these queries). Moreover, in order to enable the generation of a test database using the function illustrated in Section 11.1.3, we designed MSQL in such a way that RQP is decidable for individual queries. However, we did not prove that there does not exist a more expressive language than MSQL that has the same properties.

Additionally, in this section we also illustrate an efficient solution for the *AdjustQuery* function and the *Adjust* operator $\ominus$. Both are necessary to reverse process MSQL queries that are refined by a query refinement (as discussed in the section before). Finally, we present some extensions (query and result variables) as well as some query rewrites to enhance the usability of MSQL.

### 11.2.1 Query Classes and Algebra

In MSQL, a tester can formulate SQL `SELECT` queries with and without aggregations in the `SELECT` clause. Moreover, the queries supported by MSQL are not allowed to include join statements or subqueries and the predicate in the `WHERE` clause must be a conjunctive predicate in propositional logic that satisfies certain restrictions[5]. More precisely, the supported query classes in MSQL are:

(1) Non-Aggregation queries which can be mapped to the following relational algebra expression:

$$\pi_A(\sigma_p(T))$$

where $A$ represents the attributes and arithmetic functions in the `SELECT` clause, $p$ is the selection predicate in the `WHERE` clause, and $T$ is an arbitrary relation of the schema $S$.

(2) Aggregation queries which can be mapped to the following relational algebra expression:

$$\sigma_q(\chi_{B,COUNT(*)\ as\ c,AGG(D)}(\sigma_p(T)))$$

where $q$ is the selection predicate in the `HAVING` clause, $B$ represents the `GROUP-BY` attributes, $COUNT(*)$ is the non-distinct count function, $AGG(D)$ are the aggregation functions ($AVG$, $MIN$, $MAX$, $SUM$) in the `SELECT` clause on the attributes and arithmetic functions $D$, $p$ is the selection predicate in the `WHERE` clause, and $T$ is an arbitrary relation of the schema $S$.

[4]The RRA of RQP is the reverse variant of the relational algebra which pushes the expected query result from the root of a query tree down to the leaves in order to generate the test database.

[5]All example queries shown in the previous sections are already supported by MSQL.

The $COUNT(*)$ function is obligatory for aggregation queries (query class (2) above) because the *Adjust* operator $\ominus$ which is used in the *MRQP* function to process query refinements relies on that value (see Section 11.2.2). Moreover, for both query classes the selection predicate $p$ must be a conjunctive predicate formulated in propositional logic. If a clause $p_i$ in the conjunctive selection predicate $p$ comprises an attribute $a$ with a *primary-key* constraint, or a *unique* constraint, or a *foreign-key* constraint in the database schema $S$ then $p_i$ is only allowed to be a simple predicate expressing the equality of the attribute $a$ and a constant value $v$ (i.e., $a = v$).

The reverse relational algebra which is used to reverse process these query classes in MRQP is called MRRA. MRRA is similar to the RRA defined in Section 5. The only difference is that the reverse selection operator (i.e., $\sigma^{-1}$) and the reverse join operator (i.e., $\bowtie^{-1}$) are not allowed to generate additional tuples that satisfy the negation of the selection predicate or the negation of the join predicate. Provided, that we use MRRA to generate a test database, then the following two theorems hold.

**Theorem 11.3** *Two arbitrary MSQL queries $Q_j$ and $Q_k$ are RQP-disjoint iff $Q_j$ and $Q_k$ specify tuples for different relations or $p_j \wedge p_k$ is not satisfiable which is decidable for the selection predicates in MSQL ($p_j$ is the selection predicate representing the `WHERE` clause of $Q_j$ and $p_k$ is the selection predicate representing the `WHERE` clause of $Q_k$).*

**Proof (Sketch) 11.4** *It is obvious that $Q_j$ and $Q_k$ are RQP-disjoint if $Q_j$ and $Q_k$ specify tuples in different relations because $Q_k$ will be update independent from any `INSERT` statement which is generated by RQP for $Q_j$ and an arbitrary expected result $R_j$ of that query and vice versa. It immediately follows from [BCL86] that $Q_j$ and $Q_k$ are RQP-disjoint if $Q_j$ and $Q_k$ read tuples from the same relation $T$ and $p_j \wedge p_k$ is not satisfiable because all `INSERT` statements that could be generated for $Q_j$ and an arbitrary expected result $R_j$ by RQP satisfy $p_j$ and thus will not be returned by $Q_k$ which has the selection predicate $p_k$ and vice versa.*

**Theorem 11.5** *An arbitrary MSQL query $Q_j$ refines another arbitrary MSQL query $Q_k$ iff the queries $Q_j$ and $Q_k$ read tuples from the same relation $T$ and $(p_j \Rightarrow p_k)$ is valid which means that we have to show that $(!pj \vee pk)$ is valid or the negation $(pj \wedge !pk)$ is not satisfiable which is decidable for the selection predicates in MSQL (again, $p_j$ is the selection predicate representing the `WHERE` clause of $Q_j$ and $p_k$ is the selection predicate representing the `WHERE` clause of $Q_k$).*

**Proof (Sketch) 11.6** *It immediately follows from [BCL86] that $Q_j$ refines $Q_k$ iff $(p_j \Rightarrow p_k)$ is valid, because all `INSERT` statements that could be generated for $Q_j$ and an arbitrary expected result $R_j$ by RQP satisfy $p_j$ and thus will be returned by $Q_k$ which has the selection predicate $p_k$.*

```
 ⊖(Relation $R$, Relation $S$)
 Output: Relation $R'$
 (1) $R' = \emptyset$ //Create an empty result
 (2) FOR EACH tuple $r$ in $R$
 (3)  $r'$ = $\emptyset$ //Create empty tuple $r'$
 (4)  //Extract tuple $s$ from $S$
 (5)  $s$ = $S.get(B, R.B)$
 (6)  IF($s$==$\emptyset$) $r'$=$r$
 (7)  ELSE
 (8)   if($B$!=$\emptyset$) $r'(B)$ = $r(B)$ //set $B$
 (9)   $r'(c)$ = $r(c)$-$s(c)$ //set $c$
(10)   //Init results of agg. functions
(11)   FOR EACH attribute $agg(D)$ in $AGG(D)$
(12)    IF($agg$==SUM)
(13)     $r'(agg(D))$=$r(agg(D))$-$s(agg(D))$
(14)    ELSE IF($agg$==AVG)
(15)     $r'(agg(D))$=($r(agg(D))$*$r(c)$-$s(agg(D))$*$s(c)$)/$r'(c)$
(16)    ELSE IF($agg$==MIN || $agg$==MAX)
(17)     IF($r(agg(D))$!=$s(agg(D))$) $r'(agg(D))$=$r(agg(D))$
(18)   END FOR
(19)  END IF
(20)  $R'$.add($r'$) //add new tuple $r'$ to $R'$
(21) END FOR
(22) RETURN $R'$ //return result
```

Figure 11.2: Adjust operator ⊖

### 11.2.2 Adjust Operations for Query Refinements

The *AdjustQuery* function is used in the *MRQP* function (see Figure 11.1) to adjust the query $Q_i$ so that calling RQP for $Q_i$ does not generate any data for the expected result $R_i$ which is returned by any query in the query refinement $F_i = \{F_{i1}, \ldots, F_{in}\}$ for the query $Q_i$. The implementation of this function is the same for both query classes of MSQL. We simply extract the selection predicate $p_i$ of the query $Q_i$ and the selection predicates $p_{Fij}$ of each query $F_{ij} \in F_i$ and create a new selection predicate $p'_i$ for the adjusted query $Q'_i$ as $p'_i = p_i \wedge \neg p_{Fi1} \wedge \cdots \wedge \neg p_{Fin}$. An example for the *AdjustQuery* function was shown at the end of Section 11.1.2.

The *Adjust* operator ⊖ is used in the *MRQP* function (see Figure 11.1) to adjust the expected result $R_i$ of a query $Q_i$ w.r.t. the database $DF_i$ generated for the query refinement $F_i$. The implementation of ⊖ for a non-aggregation query $Q_i$ (query class (1) of MSQL) is a standard relational minus operator for bags as described in any database textbook. Additionally, the *Adjust* operator for non-aggregation queries checks if $Q_i(DF_i) - R_i = \emptyset$ holds. Otherwise the expected result is not adjustable because the queries in the refinement $F_i$ specify more tuples than the query $Q_i$ which is not allowed by the definition. In that case the *Adjust* operator returns an error.

For an aggregation query $Q_i$ (query class (2) of MSQL) the implementation of the *Adjust* operator is given in Figure 11.2. In that algorithm we refer to the group-by attributes $B$, the count value $c$, and the aggregation functions $AGG(D)$ that are defined by the expected result of an aggregation query. An example for that operator will be discussed in the next Section 11.2.3. The *Adjust* operator in Figure 11.2 is a binary operator which takes two relations $R$ and $S$ as input: $R$ is the expected result of the query $Q_i$ and $S$ is the actual result of executing the query $Q_i$ over the test database $DF_i$ that was generated for the query refinement $F_i$ (i.e., $S = Q_i(DF_i)$). The output of this operator is the adjusted result $R'$.

The implementation of the *Adjust* operator is as follows: The operator first creates an empty result $R'$ (Line 1) and then iterates over all tuples in the expected result $R$ (Line 2-21). For each tuple $r$ in $R$ an empty result tuple $r'$ is created that should hold the adjusted values from $r$ (Line 3). Afterwards, the tuple $s$ is extracted from $S$ that has the same values for the group-by attributes $B$ in $r$. If query $Q_i$ does not define a group-by attribute (i.e., $B = \emptyset$) then the only tuple in result $S$ is returned (Line 5). If there does not exist such a tuple $s$, then the $\ominus$ operator uses $r$ as the adjusted tuple $r'$ and adds $r'$ to $R'$. Otherwise, the $\ominus$ operator adjusts the expected query result (Line 7-19) as follows: First, the group by attributes $B$ of $r'$ are initialized with the attribute values $r(B)$ (Line 8). Then, the new count value is calculated as the difference of the original count value $r(c)$ and the count value $s(c)$ (Line 9). Finally, the adjusted expected results $r'(agg(D))$ for each aggregation function $agg(D) \in AGG(D)$ is created according to the type of the aggregation function (Line 10-18):

- Line 12-13: The adjusted expected result of a `SUM` function is calculated as the difference of the original expected `SUM` value $r(agg(D))$ and the one in $s$ (i.e., $s(agg(D))$).

- Line 14-15: The adjusted expected result of a `AVG` function is calculated as the difference of the original expected `AVG` value $r(agg(D))$ multiplied with the original expected count value $r(c)$ (which results in the original expected `SUM` value) minus the `AVG` value $s(agg(D))$ multiplied with the count value $s(c)$ divided by the adjusted expected count value $r'(c)$.

- Line 16-17: The adjusted expected result of a `MIN`/`MAX` function has the same value as $r(agg(D))$ if $r(agg(D))$ is different from $s(agg(D))$. Else, the `MIN`/`MAX` value is not added to $r'$ (which means that $agg(D)$ has to be removed from the adjusted query $Q_i'$ as well).

Finally, the adjusted tuple $r'$ is added to the result $R'$ (Line 20). If all tuples $r$ in $R$ are processed the adjusted result $R'$ is returned (Line 22). An example of that algorithm is given in the next subsection.

```
CREATE TABLE user (
u_id INTEGER PRIMARY KEY,
u_name VARCHAR(20) UNIQUE,
u_password VARCHAR(20),
u_charges FLOAT NOT NULL
          CHECK(u_charges>=0));


CREATE TABLE book (
b_id INTEGER PRIMARY KEY,
b_isbn VARCHAR(20) UNIQUE,
b_closedstack BOOLEAN NOT NULL
b_aid INTEGER FOREIGN KEY
      REFERENCES author(a_id));


CREATE TABLE author (
a_id INTEGER PRIMARY KEY,
a_name VARCHAR(20) UNIQUE,
a_fname VARCHAR(20));
```

**(a) Database Schema**

$Q_1$:
```
SELECT COUNT(*)
FROM book
WHERE b_aid=1
```
$R_1$: `<5>`

$Q_2$:
```
SELECT u_password
FROM user
WHERE u_name = 'test'
AND u_charges<=20
```
$R_2$: `<'test'>`

$F_1$:
```
SELECT b_closedstack
FROM book
WHERE b_isbn='0130402648'
AND b_aid=1
```
$R_{F1}$: `<false>`

**(b) Example Query Set Q={$Q_1$, $Q_2$} and Query Refinement F={$F_1$}**

$D_1$:

| b_id | b_isbn | b_closedstack | b_aid |
|---|---|---|---|
| 1 | 0130402648 | false | 1 |
| 2 | 0130402649 | true | 1 |
| 3 | 0130402650 | false | 1 |
| 4 | 0130402651 | true | 1 |
| 5 | 0130402652 | true | 1 |

*book*

| a_id | a_name | a_fname |
|---|---|---|
| 1 | a_name1 | a_fname1 |

*author*

$D_2$:

| u_id | u_name | u_password | u_charges |
|---|---|---|---|
| 1 | test | test | 0.0 |

*user*

$D_F$:

| b_id | b_isbn | b_closedstack | b_aid |
|---|---|---|---|
| 1 | 0130402648 | false | 1 |

*book*

| a_id | a_name | a_fname |
|---|---|---|
| 1 | a_name1 | a_fname1 |

*author*

**(c) Generated Test Database D = $D_1 \cup D_2$**

Figure 11.3: MSQL Example

The *Adjust* operator $\ominus$ for aggregation queries returns an error if the expected result $R$ is not adjustable w.r.t. the result $S$ , if one of the following cases occurs (this error handling is not implemented in Figure 11.2):

- The expected count value $R(c)$ is less than $S(c)$
- $SUM(D)$ in $R$ is less than $SUM(D)$ in $S$ and $D$ can only have positive values or $SUM(D)$ in $R$ is greater than $SUM(D)$ in $S$ and $D$ can only have negative values
- $AVG(D)$ in $R$ is less than $AVG(D)$ in $S$ and $D$ can only values greater than $AVG(D)$ in $R$ or $AVG(D)$ in $R$ is greater than $AVG(D)$ in $S$ and $D$ can only values less than $AVG(D)$ in $R$
- $MIN(D)$ in $R$ is greater than $MIN(D)$ in $S$ or $MAX(D)$ in $R$ is less than $MAX(D)$ in $S$
- $S$ has a tuple $s$ that has values for the group-by attributes $B$ and $R$ does not have a tuple with the same values for the attributes $B$.

### 11.2.3 MSQL Example

Figure 11.3 gives a complete example of MRQP: Figure 11.3 (a) shows the database schema and Figure 11.3 (b) shows the RQP-disjoint query set $Q$ and a query refinement $F_1$ which specify the test database for Test Case 5 of the online library (see Section 1). The query $Q_1 \in Q$ specifies the total number of all books in the test database. The other query $Q_2 \in Q$ specifies the password and the charges of a particular user with a certain password. The queries $Q_1$ and $Q_2$ are RQP-disjoint because they specify tuples for different tables. The query $Q_1$ is refined by a query refinement $F_1 = \{F_{11}\}$ and its expected results $RF_1 = \{RF_{11}\}$. The query $F_{11}$ and the result $R_{11}$ specify the value of the attribute $b\_closedstack$ of one book in the test database with a particular ISBN (i.e., $b\_isbn =$ '0130402648').

If we call the function *MRQP* in Figure 11.1 for the RQP-disjoint query set $Q$, then the test database $DF_1$ for the query refinement $F_1$ is generated first ($DF_1$ is shown in Figure 11.3 (c)). Due to the foreign-key handling in RQP a tuple for the author with $a\_id = 1$ is created, too. As a next step in MRQP, the adjusted expected result for query $Q_1$ is calculated as $R'_1 = R_1 \ominus Q_1(DF_1)$: The query $Q_1$ is an aggregation query and $Q_1(DF_1)$ returns a count value of $1$. Following the algorithm of $\ominus$ for aggregation queries (see Figure 11.2) the adjusted expected result $R'_1$ has one tuple with the count value of $4$. Afterwards, the query $Q_1$ is adjusted which results in $Q'_1$:

$Q'_1$ :
```
SELECT COUNT(*) FROM book
WHERE b_isbn!='0130402648'
```

Afterwards, the test database $D'_1$ for the adjusted query $Q'_1$ and its adjusted expected result $R'_1$ is generated which means that four tuples in the table $book$ are created which satisfy $b\_isbn! =$ '0130402648' ($D'_1$ is not shown separately in Figure 11.3 (c)). The database $D_1$ which is shown in Figure 11.3 (c) is the union of $D'_1$ and the test database $DF_1$ generated for the query refinement $F_1$.

Finally, the test database $D_2$ is generated for $Q_2$ and $R_2$ ($Q_2$ is not refined by a query refinement). As a last step, the final test database $D$ is created as the union of $D_1$ and $D_2$ (i.e., $D = D_1 \cup D_2$).

### 11.2.4 Queries and Result Variables

For a test case of an OLTP application it is common that a tester needs to specify individual tuples in the test database that have certain values (e.g., an author with a certain name or a book with a certain ISBN) in order to enable the execution of that test case. Thus, a tester often needs to formulate queries that specify single tuples using a *unique* value for an attribute in the selection predicate with a *primary-key* constraint or a with a *unique* constraint in the database schema (e.g., the predicate $b\_id = 1$ on the table $book$). However, defining unique values for such an attribute in the selection predicates or in the expected query results over different queries (of one test case or even over different test cases) that are used as input of MRQP is not trivial for the tester.

Consequently, we extend MRQP so that the user can define variables as a placeholders for unique values and let MRQP instantiate the variables so that the constraints of the database schema $S$ are satisfied. A variable has a name that starts with a \$ sign (e.g., $\$b\_id1$ could be use as a variable in the selection predicate above which results in $b\_id = \$b\_id1$). In general a variable can be used as a placeholder in the following cases:

(1) As a placeholder for a constant value $v$ of a *primary-key* attribute, an attribute with a *unique* constraint, or a *foreign-key* attribute in the expected result of a MSQL query.

(2) As a placeholder for a constant value $v$ in a clause $p_i$ (i.e., a simple predicate) of the conjunctive selection predicate $p$ of a MSQL query $Q_i$ where the clause $p_i$ expresses the equality of a *primary-key* attribute, or an attribute with a *unique* constraint, or a *foreign-key* attribute and a constant value $v$ (e.g., $b\_id = \$b\_id1$).

However, RQP does not support variables as placeholders for unique values. Consequently, in order to instantiate these variables, we add a pre-processing phase to MRQP which creates unique values for these variables defined by an MSQL query and its expected result. This pre-processing phase instantiates the variables using the following rules:

- For variables specified by the queries and expected results of one test database specification for one test case (i.e., an RQP-disjoint query, some query refinements, and the expected query results) which have the same name and which are assigned to the same attribute, the pre-processing phase instantiates the same value.
- For variables specified by the queries and expected results of one test database specification for one test case which have different names and which are assigned to the same attribute, the pre-processing phase instantiates different values.
- For variables specified by the queries and expected results of different test database specification for different test cases which are assigned to the same attribute, the pre-processing phase instantiates different values (This is useful if we want to merge two test databases of different test cases; see Section 11.3).

An example of the pre-processing phase is given for the following RQP-disjoint query set $Q = \{Q_1, Q_2\}$ which is used in a test database specification for one test case. A valid instantiation of the variables that could be produced by the pre-processing phase of MRQP is: $\$b\_id1 = 1$ and $\$b\_id2 = 2$.

$Q_1$ :
```
SELECT b_title
FROM books
WHERE b_id=$b_id1
```

$Q_2$ :
```
SELECT b_title
FROM books
WHERE b_id=$b_id2
```

As the pre-processing phase does not analyse all queries and expected results of one or even all test database specifications before it instantiates the unique values for the variables, the pre-processing phase requires that either all test database specifications define the values of a particular attribute as variables or as constants. Otherwise, this phase may generate values that violate the constraints of the database schema $S$ (e.g., different tuples with the same value for a *primary-key* attribute might be generated). For example, assume that one test database specification for on test case defines an expected result $R_1$ of a query $Q_1$ holding a variable for a result attribute (e.g., the variable $\$b\_id1$ for the attribute $b\_id$) and another expected result $R_2$ of a query $Q_2$ holding a constant value (e.g., $1$ for the attribute $b\_id$). If MRQP first generates a test database $D_1$ for $Q_1$ and $R_1$, then the pre-processing phase could instantiate the constant value 1 for the variable $\$b\_id1$ in $R_1$. If MRQP subsequently generates the test database $D_2$ for the $Q_2$ and $R_2$ which defines the constant value 1 for the attribute $b\_id$, then the union of $D_1$ and $D_2$ will return an error because the *primary-key* constraint on the attribute $b\_id$ in the database schema $S$ is violated.

### 11.2.5 Query Rewrites

The query classes of MSQL that can be used by the tester to specify the test database (i.e., non-aggregation and aggregation queries on one relation) are limited because join operations or nested queries are not supported. However, some of these queries can be rewritten so that they are supported by MSQL. A rewrite of a query $Q_i \in Q$ ($Q$ is a RQP-disjoint set of MSQL queries which specifies the test database in MRQP) into one or more queries $P$ is valid if (1) all queries in $P$ are supported by MSQL and (2) the query set consisting of $\{Q - Q_i \cup P\}$ is still RQP-disjoint. In this section we discuss rewrites of queries using equi-joins, nested queries, and queries using view definitions.

#### Rewrites for Equi-Join Queries

A non-aggregation query $Q_i$ (i.e., query class (1) of MSQL) where $T$ is the result of a 2-way equi-join on the relations $T_1$ and $T_2$ can be rewritten as follows if the selection predicate $p$ is a conjunctive predicate where each clause uses attributes from either $T_1$ or $T_2$:

(1) If $T$ is the result of an equi-join on the primary attributes of the two input relations $T_1$ and $T_2$[6], then we split $Q_i$ into two queries $Q_{i1}$ and $Q_{i2}$: $Q_{i1}$ is a new non-aggregation query on the relation $T_1$ and $Q_{i2}$ is a new non-aggregation query on the relation $T_2$. The projection attributes for $Q_{i1}$ are $A_1 = A \cap attr(T_1)$ ($A$ are the attributes in the `SELECT` clause of $Q_i$, $attr(T_1)$ is a function that returns all attributes of the relation $T_1$). The selection predicate $p_1$ of $Q_{i1}$ is a conjunction of all clauses in the selection $p$ of $Q_i$ that uses attributes in $attr(T_1)$. The expected result of $Q_{i1}$ is $R_{i1} = \pi_{A1}(R_i)$ (where $R_i$ is the expected result of $Q_i$ and the $\pi$ operator deletes duplicates). $Q_{i2}$ and $R_{i2}$ can be created analogously.

[6] $T_1$ and $T_2$ could represent one entity that was vertically split into two relations for performance reasons.

(2) If $T$ is the result of an equi-join along the *foreign-key* relationship from $T_2$ to $T_1$ (e.g., an equi-join on the *foreign-key* attribute $b\_aid$ in the table $book$ and the *primary-key* attribute $a\_id$ in the table $author$), then we create $Q_{i1}$ and $Q_{i2}$ as well as the expected results $R_{i1}$ and $R_{i2}$ as described in (1). Additionally, we add the *primary-key* attribute of $T_1$ to the projection attributes $A_1$ (e.g., the attribute $a\_id$ of the table $author$) and the *foreign-key* attribute of $T_2$ to the projection attributes $A_2$ (e.g., the attribute $b\_aid$ of the table $books$). We also have to add these attributes and the corresponding values to the expected results $R_{i1}$ and $R_{i2}$ to implement the *foreign-key* relationship explicitly (e.g., by using the same variables or constants for the *primary-key* and *foreign-key* attributes that are to be joined). Moreover, if there is an equality-predicate on the *primary-key* attribute of $T_1$ in $p_1$ of $Q_{i1}$ (e.g., $a\_id = \$a\_id1$, where $\$a\_id1$ is a variable), then we add the same simple equi-predicate to $p_2$ of $Q_{i2}$ replacing the *primary-key* attribute in $T_1$ by the *foreign-key* attribute of $T_2$ (e.g., $b\_aid = \$a\_id1$). In that case we do not need to add the *primary-key* attribute and the *foreign-key* attribute to the projection attributes (i.e., to $A_1$ and $A_2$) of the queries $Q_{i1}$ and $Q_{i2}$ as well as to the expected results $R_{i1}$ and $R_{i2}$. Alike, if there is an equality-predicate on the *foreign-key* attribute of $T_2$ in $p_2$ of $Q_{i2}$ (e.g., $b\_aid = \$b\_aid1$, where $\$b\_aid1$ is a variable) then we can add a simple equality-predicate on the *primary-key* of $T_1$ to $p_1$ that uses the value of the *foreign-key* attribute from $T_2$ defined in $p_2$ (e.g., $a\_id = \$b\_aid1$) .

A complete example of the rewrite (2) is given by the following 2-way join query $Q_1$ which selects all book titles, prices and the author name of one particular author:

$Q_1$ :
```
SELECT b_title, b_price, a_name
FROM book JOIN author ON b_aid=a_id
WHERE a_id=$a_id1
```
$R_1$ :
```
{<'TitleB',64.80, 'Hector Garcia-Molina'>}
```

This query can be rewritten into the two queries $Q_{11}$ and $Q_{12}$ below. The selection predicate of $Q_{11}$ (i.e., $a\_id = \$a\_id1$) is used directly as selection predicate for $Q_{12}$ on the *foreign-key* attribute $b\_aid$ (i.e., $b\_aid = \$a\_id1$). Following the rule (2) above, the *primary-key* attribute $a\_id$ is not added to the expected result $R_{11}$ of query $Q_{11}$ and the *foreign-key* attribute $b\_aid$ is not added to the expected result $R_{12}$ of query $Q_{12}$:

$Q_{11}$ :
```
SELECT a_name FROM author
WHERE a_id=$a_id1
```
$R_{11}$ :
```
{<'Hector Garcia-Molina'>}
```
$Q_{12}$ :
```
SELECT b_title, b_price FROM book
WHERE b_aid=$a_id1
```
$R_{12}$ :
```
{<'TitleB',64.80>}
```

N-way equi-joins can be rewritten by applying the above rules recursively.

```
$Reduce$(List of Test Cases $T$, Database Schema $S$)
Output: Map $D\_T$
//Generate one Test DB per Test Case $T_i$:
//$T_i$ specifies Queries $T_i.Q$ + Results $T_i.R$
$D$ = Empty List of Test DBs
FOR EACH $T_i$ in $T$
 $D_i$=MRQP($T_i.Q$, $T_i.R$, $S$)
 $D$.add($D_i$) //add $D_i$ to $D$
END FOR
//Reduce the number of Test DB
$D\_T$=Empty Map(Key:DB,Value:Test Cases)
FOR(i=1...$D$.length())
 $D_i$=$D$.get($i$) //Test DB $D_i$
 $T_i$=$T$.get($i$) //Test Case $T_i$
 FOR EACH $D_j$ in $D\_T$.keys()
  IF($D_{ij} = D_i \cup D_j$ returns error) continue
  $T_J$=$D\_T$.get($D_j$) //Test Cases for $D_j$
  //Test if merge was successful
  IF($Q(D_{ij}) = R$ for all $T_J$ and $T_i$)
   $D\_T$.remove($D_j$) //Remove $D_j$ and $T_J$
   //Add $D_{ij}$ and all Test Cases to $D\_T$
   $D\_T$.add($D_{ij}$, $T_J \cup T_i$)
   BREAK //Continue with next $D_i \in D$
  END IF
 END FOR
 $D\_T$.add($D_i$, $T_i$)
END FOR
RETURN $D\_T$
```

Figure 11.4: *Reduce* Function

**Other Rewrites**

A nested query or a query that involves a view definition is supported by MSQL if the query can be rewritten by unnesting techniques discussed in [GW87] or view unfolding if the rewrite is valid as discussed at the beginning of this subsection.

## 11.3 Reducing the Test Databases

In this section, we present a greedy algorithm which first generates an individual test database for each test database specification of test case (of a given set of test cases) and then tries to reduce the number of test databases that are necessary to execute all test cases. The implementation of this algorithm is given by the function *Reduce* in Figure 11.4. This function takes a set of test cases $T$ (where each test case $T_i \in T$ defines a set of queries $T_i.Q$ and the corresponding expected results $T_i.R$ as test database specification) and a database schema $S$ as input and generates a

| | |
|---|---|
| Input: | $T = \{T_1, T_2, T_3, T_4\}$ |
| Output: | $D\_T = \{D_{124} \rightarrow \{T_1, T_2, T_4\}, D_3 \rightarrow \{T_3\}\}$ |
| Line 7: | $D = \{D_1, D_2, D_3, D_4\}$ |
| Line 25, $D_1$: | $D\_T = \{D_1 \rightarrow \{T_1\}\}$ |
| Line 25, $D_2$: | $D\_T = \{D_{12} \rightarrow \{T_1, T_2\}\}$ |
| Line 25, $D_3$: | $D\_T = \{D_{12} \rightarrow \{T_1, T_2\}, D_3 \rightarrow \{T_3\}\}$ |
| Line 25, $D_4$: | $D\_T = \{D_{124} \rightarrow \{T_1, T_2, T_4\}, D_2 \rightarrow \{T_3\}\}$ |

Figure 11.5: Example of the *Reduce* Function

map $D\_T$ that assigns test cases to a test database that could be used to execute these test cases. This algorithm does not guarantee to find the minimal number of test databases for a set of test cases. This would be an avenue for future work to find an efficient algorithm which guarantees minimality.

As discussed before, the function *Reduce* first generates a list of test databases $D$ which contains one test database $D_i$ for each test case $T_i \in T$ using MRQP (Line 1-7). Afterwards, the function reduces the number of test databases required to execute all test cases in $T$ (Line 8-26): Therefore, a map $D\_T$ is created which assigns a list of test cases to individual test databases (Line 9). Then, the function iterates over all generated test databases $D_i$ in $D$ and tries to merge each test database $D_i$ with the test databases saved as keys in the map $D\_T$. Obviously, for the first test database in $D$ the lines 13-23 are skipped because $D\_T$ is empty and the test database $D_i$ and the corresponding test case $T_i$ are just added as a new entry to the map $D\_T$ (Line 24). For all other test cases, the function tries to create a merged database $D_{ij}$ iteratively for each key in $D_T$ and checks if the following conditions hold for the merged database $D_{ij}$: (1) the union does not return an error because it violates the schema $S$ (Line 14) and (2) the queries of all test cases that should be executed on the merged test database $D$ (i.e., the list of test cases $T_J$ and the test case $T_i$) return the expected results for all queries of those test cases (Line 17). In the current implementation of the *Reduce* function we use DBMS to merge the databases and to check whether the merged database satisfies the given database schema $S$. If these conditions hold, then the database $D_j$ and the corresponding test cases $T_J$ are removed from $D\_T$ and the list of test cases $T_J$ for $D_j$ together with the test case $T_i$ for $D_i$ are assigned to the merged test database $D_{ij}$ (Line 18-21). At the end of the function, the map $D\_T$ is returned (Line 26).

Figure 11.5 shows an example for the function *Reduce*. The input is a list of four test cases $T = \{T_1, T_2, T_3, T_4\}$ and the output is a map that assigns the list of test cases $\{T_1, T_2, T_4\}$ to a test database $D_{124}$ and the test case $T_3$ to a test database $D_3$. The example shows the list $D$ and the map $D\_T$ in different stages of the algorithm. At the beginning (function *Reduce*, Line 7) a list of test databases $D$ is created that contains one test database for each test case in $T$ (The queries and expected results of each test case are not shown). Afterwards, the function *Reduce* merges these test databases:

- *Function Reduce, Line 25:* In the first iteration the test database $D_1$ and the test case $T_1$ is added to the empty map $D\_T$ without merging.
- *Function* Reduce, *Line 25:* In the next iteration, the test database $D_2$ is successfully merged with $D_1$ which results in a new test database $D_{12}$ that can be used to execute the test cases $T_1$ and $T_2$.
- *Function Reduce, Line 25:* In the third iteration the merge of the test database $D_3$ and $D_{12}$ fails, e.g. because the queries of one test case in $\{T_1, T_2, T_4\}$ do not return the expected results for the merged test database $D_{123}$. Thus, the test database $D_3$ and test case $T_3$ are added as a new entry to $D\_T$.
- *Function Reduce, Line 25:* Finally, in the last iteration the test database $D_4$ is successfully merged with $D_{12}$.

## 11.4 Related Work

The closest related work to MRQP is the work on Information Disclosure like the one in [MS04]. This work addresses the question which information is disclosed by a set of views that are published over the same database instance. Moreover, there has also been some work on efficient algorithms and frameworks to produce large amounts of test data for a given statistical distribution [GSE+94; BC05]. This work is orthogonal to our work. All other existing approaches on test databases generation (e.g., [NML93; CDF+04]) focuses on other particular aspects not directly relevant for MRQP.

# Chapter 12

# Functional Testing of a Query Language

*The whole of science is nothing more than a refinement of everyday thinking.*

*– Albert Einstein, 1879-1955 –*

Functional testing of a query language like SQL, or more precisely functional testing the execution of a query, is a challenging task in practice. A basic problem is to create a comprehensive set of test queries and test databases which enable a high test coverage of the query processing functionality (e.g., of a relational DBMS). Consequently, many existing approaches (e.g., [Slu98], [PS04], [SP04], [GSE$^+$94], [HTW06], [BC05]) focus on the generation of test queries and test databases with various query and data characteristics. These approaches are often used to find errors by simply executing the generated test queries on different test databases. However, these approaches do not address the problem of automatically verifying the actual result of the test queries over the test databases which is a crucial task to reveal errors in the query processing functionality.

For instance, if we want to use the following SQL query for functional testing the query processing functionality of a relational DBMS, then the execution of the query over a database may only reveal some abnormal behavior such as a very long execution time or a system crash. However, functional errors like defects in the filter operation or in the aggregation operation can only be found by the verification of the actual query result.

```
SELECT o_orderdate,SUM(l_price) as sum1
FROM orders, lineitem
WHERE o_id=l_oid
AND o_orderdate>=date ’2005-01-01’
GROUP BY o_orderdate;
```

In order to verify the actual result of a test query over a certain database, the actual result has to be compared to the expected (correct) result. If the actual result is the same as the expected result, then the verification succeeds, otherwise the verification fails.

The problem of automatically computing the expected result of an arbitrarily complex test query over a given test database is not trivial. One solution is to first generate a set of test databases as well as test queries and then to compute the expected query results for each test query over the individual test databases by executing the test query on an alternative query processor with comparable capabilities (like the previous version, or a implementation of a different vendor) as proposed in [Slu98]. However, this solution is not feasible for the testing of the new features of a query language which are not yet supported by another query processor or to test a new query language like Entity SQL of the ADO.Net Entity Framework [ABMM07] where no comparable implementation exists. Another problem of this solution is that many test queries might return empty results which are of no interest for the functional testing of a query language.

In this chapter, we discuss a solution that addresses the verification of the actual query result in a different way. Instead of first generating a set of test databases as well as test queries and then computing the expected results, we first create one or more expected results for a given test query and then generate a test database individually for each combination of a test query and an expected result which returns the expected result if the test query is executed correctly.

In order to generate a database instance for a given test query and an expected result of that query, we use the RQP framework discussed in Part II. Our approach to verify the actual query result is based on the assumption that the implementation of RQP is correct[1].

The main benefit of our approach is that we have full control of the expected results of each test query. Thus, we can define the test cases for the functional testing of a query language from a totally different angle. For example, we can explicitly create two test databases for the same test query with a filter operation where the first test database returns a minimal result and the second test database returns a huge result with interesting boundary values that we intent to use for the functional testing of the filter operation of the test query.

A drawback of our methodology is that it is expensive to generate one test database $D$ individually for each combination of a test query $Q$ and an expected result $R$. Mutating a test query $Q$ into a test query $Q'$ with the condition that $Q'(D) = R$ still holds, allows us to reuse the test database $D$ in order to test query $Q'$ and verify the actual result of that query with the help of the same expected result $R$. An example mutation of a SQL query which does not change the expected result, is to add a self join on the *primary-key* of a relation which is used in the `FROM` clause of the query. Mutations are not discussed in this thesis.

[1]In functional testing, it is a general assumption that the testing tool is correct, i.e. a failed test run primarily indicates a bug in the application under test and not in the testing tool itself.

**Contributions:** In order to generate a test database for an arbitrary test query, our approach needs to automatically create a valid expected result for that test query first. However, generating a valid expected query result for an arbitrary test query (like the example query at the beginning of this Section) is not a trivial task. As a first contribution of this chapter, we extend RQP so that it automatically creates a valid result for a given test query. Moreover, we present a method which efficiently compares the actual with the expected result. As a third concrete contribution, we discuss the implementation of our approach for the query language Entity SQL of the ADO.NET Entity Framework and particularly focus on the extension of RQP for the nested relational data model in order to support this query language.

**Outline:** The remainder of this chapter is organized as follows: Section 12.1 discusses our new approach for the functional testing of the query language SQL. Section 12.2 then gives an overview of the ADO.Net Entity Framework and the query language Entity SQL. Subsequently, Section 12.3 describes the extensions of RQP in order to support that query language Entity SQL instead of SQL only. Finally, Section 12.4 discusses related work.

## 12.1 Functional Testing of SQL

Our approach for the functional testing of a query language like SQL can be divided into three phases which will be discussed in detail in this section: (1) In the first phase an expected query result $R$ is generated for a given test query $Q$ and a database schema $S$. The test query which must be provided as input to this phase can either be created manually by the tester or be generated by using an existing approach like RAGS [Slu98] or QGen [PS04]. The database schema is usually created manually by the tester. (2) Afterwards, a test database $D$ is generated for the given database schema which returns the expected result $R$ (generated in step 1), if the test query $Q$ is executed correctly over the database instance $D$, i.e. $Q(D) = R$. This step is completely based on RQP [2]. (3) The last step carries out the actual functional testing: the test query $Q$ is executed over the database $D$ and the actual result $R'$ is compared to the expected result $R$ for verification. If both results are the same, the verification succeeds.

### 12.1.1 Generating the Expected Query Result

The basic idea to generate a valid result for an arbitrary test query is to use the input schema of the root operator of the reverse query tree (called *result schema*) which is computed during the bottom-up query annotation phase of RQP. The input schema of the root operator describes all possible instantiations of a result of a query. To guide the result instantiation, the user can provide

[2]As future work, we want to explore different knobs for RQP which allow us to vary the characteristics of the generated test database.

values for the *result size* and a *set of constants* for each attribute which participates in the query result. If these values are not provided by the user, then the expected result generation use a default value for the result size and the complete domain of the attributes to instantiate a valid result which fulfills the constraints of the result schema.

Generating a query result which satisfies the result schema can be implemented by adding a reverse projection with an empty attribute list on top of the reverse query tree during *query compilation* of RQP. Thus, the result generation can reuse the algorithms of the reverse projection operator for the top-down data instantiation phase.

The top-down data instantiation phase of RQP is changed to get the result size and a set of constants as input instead of a valid result $R$. In a first step, this phase generates a result $R^{\emptyset}$ which consists of a set of *empty tuples*[3]. The number of empty tuples in $R^{\emptyset}$ is given by the result size. The result $R^{\emptyset}$ is used as input for the rewritten reverse query tree with the additional reverse projection on top. During reverse query execution, the additional reverse projection generates the expected result $R$ of the original reverse query tree as output. The constants which are provided as input to the top-down data instantiation phase are used to instantiate the values in $R$ (if possible). The necessary modifications in the system architecture of RQP (Query Compilation, Top-down Data Instantiation) which implement these changes are shown bold in Figure 12.1.

For example, assume that we want to generate a valid expected result $R$ for the reverse query tree in Figure 4.2 (b) (i.e., values for the attribute $SUM(price)$): The user inputs the result size of two tuples and gives the set of constants $\{0, MAX\_FLOAT\}$ for the attribute $SUM(price)$. Following the solution described before, the *query compilation* phase of RQP adds a reverse projection on top of the reverse aggregation. During reverse query processing, the *top-down data instantiation* phase creates two empty tuples as result $R^{\emptyset}$. $R^{\emptyset}$ is used as input for the additional reverse projection which generates a valid query result $R$ as output using the set of constants provided by the user in order to instantiate the values for the attribute $SUM(price)$.

### 12.1.2 Verifying the Actual Query Result

To verify the actual result $R'$ of a test query $Q$ over a database $D$ we compare $R'$ with the expected result $R$ which was used to generate the database $D$. In a first step, we check if both results have the same result size. If they do not have same size, then the verification of the actual query result fails. Otherwise, we have to compare the actual and the expected result by their result values.

If the test query contains an `ORDER BY` statement on the *primary-key* attribute(s) of the result schema of a reverse query tree, then it is guaranteed that the tuples of the actual and the expected result are in the same order. Otherwise, it makes sense to sort both results by the same sort criteria (either ascending or descending). If the result schema of the reverse query tree contains a *primary-*

[3]An *empty tuple* is a dummy tuple which defines no attribute values.

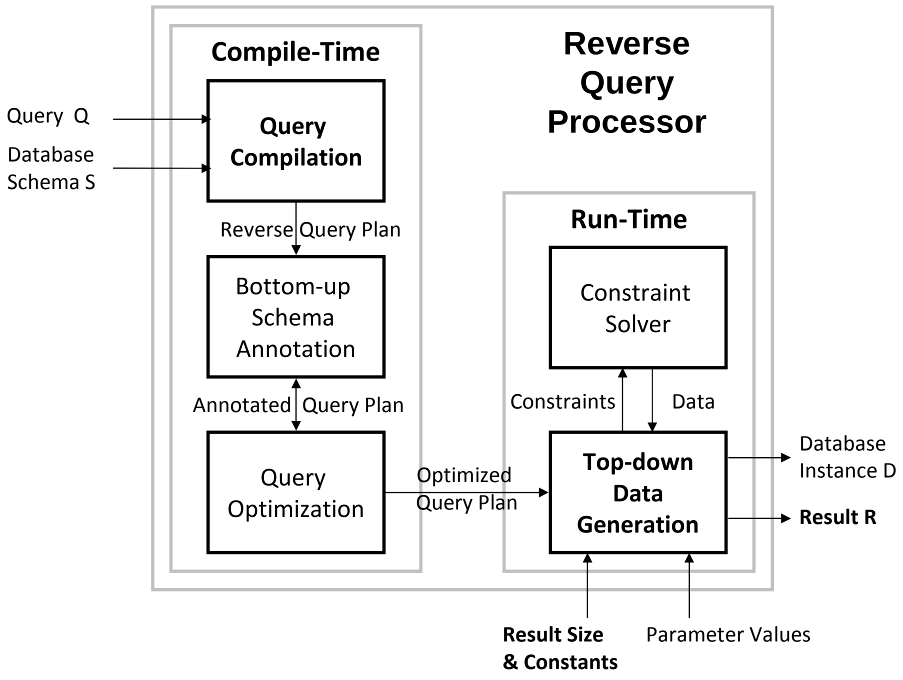


Figure 12.1: Modified RQP Architecture

*key* constraint, then we can use the key attributes to sort both results. Otherwise, we have to sort both results by all result attributes.

After sorting both results, we can compare the results tuple-by-tuple, i.e. we compare the first tuple of the actual result with the first tuple of the expected result, then the second tuple of the actual result with the second tuple of the expected result and so on. If one pair of tuples has a different number of result attributes or a different result value for one attribute, then the verification fails. If all tuples have the same value for each attribute, then the verification succeeds.

For example, if we want to verify the actual result of the SQL query in Figure 4.2 (b) which was used to generate the database instance (table $lineitem$ and $orders$) in Figure 4.2 (c), then we first execute that query on the generated database instance. Assume, that the actual result of that test query over that database instance contains the following two tuples `{<120>, <100>}`. In order to verify the actual result, we first check if the expected result (table $i$) in Figure 4.2 (c) has the same size as the actual result. As the result size is the same, we sort the actual and the expected result ascending by the attribute $SUM(l\_price)$ of the query result and compare both results tuple-by-tuple. The verification succeeds because the expected and the actual result are the same.

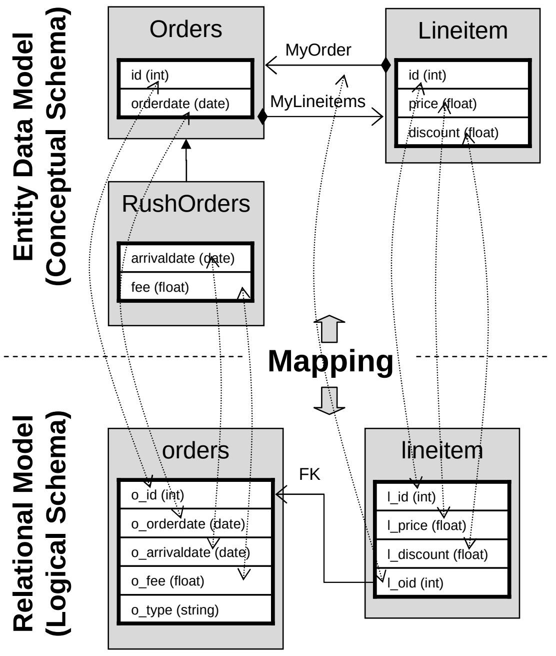


Figure 12.2: Entity Framework – Mapping

## 12.2 The ADO.NET Entity Framework

The ADO.NET Entity Framework is a data-access layer which enables developers to model and access their data on the client using a conceptual schema called Entity Data Model. The upper part of Figure 12.2 shows an example Entity Data Model. The Entity Data Model is a concrete implementation of the entity-relationship model [Che76].

The Entity Data Model defines entity types (e.g., $Orders$, $Lineitem$) and their associations. Entity types represent a structured record consisting of one or more properties. The properties of an entity type have a simple or a complex data type. A simple data type represents a scalar type (e.g., $int$, $string$), while a complex data type represents a structured property (e.g., $address$ which is not shown in the example). A complex data type is composed of one or more properties, which again have a simple or complex data type. Associations are used to relate (or, describe relationships between) two or more entity types (e.g., the association $MyOrder$ in Figure 12.2 relates the $Lineitem$ entity type with the $Order$ entity type). Moreover, the Entity Data Model also supports inheritance; i.e., an entity type can be derived from another entity type (e.g., in Figure 12.2 the entity type $RushOrders$ is derived from the entity type $Orders$).

Entities are instances of entity types. An entity is uniquely identified by a key which is formed out of one or more properties of the entity type (e.g., the key of the entity type $Lineitem$ is the property $id$), just like a key in the relational data model. The entities are organized in persistent collections called entity sets. An entity set of type $T$ holds entities of type $T$ or any type that derives from $T$.

The ADO.NET Entity Framework does not materialize the entities and association on the client. The Entity Data Model is mapped to a relational data model via a flexible mapping. The details of the mapping can be found in [MAB07]. An example mapping is shown by the arrows in Figure 12.2. The Entity Data Model and the relational data model in the example contain roughly the same elements. One difference is that the attributes in the relational data model use slightly different names than the properties of the Entity Data Model. Another difference is that the entity type $Orders$ and the derived entity type $RushOrders$ are mapped to one and not two tables in the relational data model. The table $orders$ holds the properties of both entities and has an additional column with the name $o\_type$ which stores information about the entity type of the tuple (e.g., the string '$Orders$' or '$RushOrders$'). Moreover, the associations of the Entity Data Model are implemented as an additional attribute $l\_oid$ and a *foreign-key* in the table $lineitem$.

Entity SQL is the data manipulation language for the Entity Data Model. An Entity SQL query retrieves entities from one or more entity sets. The following query is an example of an Entity SQL statement which queries the entity set $Lineitem$. For convenience, we assume that the entity set has the same name as the entity types in Figure 12.2.

```
SELECT l.price, l.MyOrder.orderdate
FROM Lineitem l
WHERE l.MyOrder is of RushOrders
```

Entity SQL supports expressions to navigate from one entity to a one or more entities reachable via a given association (e.g., `l.MyOrder` is the navigation from a $Lineitem$ entity to the corresponding $Orders$ entity). Moreover, filter operations support type interogation by using the `IS OF` expression (e.g., `IS OF RushOrders` checks if an entity is of the type $RushOrders$).

Query execution in the Entity Framework is delegated to the relational store. Thus, the Entity Framework translates an Entity SQL query into an equivalent SQL query which can be executed by the query processor of the underlying relational database. The translation is based on the so called query and update views which are derived from the mapping of the Entity Data Model to the relational data model. These views are used to translate queries and updates on instances of the Entity Data Model to queries and updates on the relational data.

The following query shows the translation of the Entity SQL query above into a corresponding SQL statement for the mapping which is defined in Figure 12.2. The navigation $l.MyOrder$ in the Entity SQL query is translated into a join on the tables $lineitem$ and $orders$. The `IS OF` predicate of the Entity SQL query is translated into a filter operation with a simple equality predicate on the column $o\_type$.

```
SELECT l_price, o_orderdate
FROM lineitem JOIN orders ON l_oid = o_id
WHERE o_type='RushOrder'
```

After executing the SQL query on the relational store, the result is reshaped by the ADO.Net Entity Framework according to the structures of the Entity Data Model (e.g., creating associations from *foreign-key* values) and returned to the client.

## 12.3 Reverse Query Processing Entity SQL

### 12.3.1 Discussion and Overview

Section 12.2 showed that an Entity SQL query is executed over a given database instance using the following three steps:

(1) An Entity SQL query is translated into a SQL query
(2) The SQL query is executed over the relational database
(3) The result of the SQL query is mapped into a result of the Entity Data Model

Functional testing of Entity SQL should be able to reveal errors in all the three steps of the query execution. The main steps of our approach for the functional testing of a query language (see Section 12.1) are the generation of an expected result $R$ and of a test database $D$ for a given test query $Q$ and a database schema $S$ using RQP. However, the existing RQP prototype supports only SQL and the relational data model but not Entity SQL and the Entity Data Model. Consequently, in order to reverse process an Entity SQL query, we either use the SQL query $Q'$ which is the output of step (1) above and the relational database schema which is used to store the data of the Entity Data Model as input for RQP or we extend RQP to support Entity SQL and the Entity Data Model directly[4].

The problem of using the SQL query $Q'$ as input for RQP is that the step (1) of processing an Entity SQL query could be erroneous and thus the output SQL query maybe *wrong*. Consequently, RQP would generate a test database $D$ for that wrong SQL query which means that the correctness criterion $Q(D) = R$ does not hold anymore for the generated test database $D$ and the Entity SQL query $Q$.

In this thesis, we describe a solution where RQP takes an Entity SQL query and an Entity Data Model directly as input in order to avoid this problem. Implementing RQP for Entity SQL and the

[4]We do not discuss the verification of a result of an Entity SQL query because this is a straightforward extension to Section 12.1.2.

Entity Data Model needs some extensions in the data model and the algebra of RQP. The Entity Data Model and the algebra for Entity SQL (called Command Trees [ABMM07]) are similar to the nested relational model and the nested relational algebra. In the following Sections 12.3.2 and 12.3.3, we define an *Extended* Nested Relational Data Model as well as a Reverse Nested Relational Algebra for RQP and discuss the mapping of the Entity Data Model and the query language Entity SQL to that data model and that algebra.

Based on that algebra and that data model an Entity SQL query $Q$ is reverse-processed in the following way: In the first step, the Entity Data Model $S$ is mapped into a corresponding Extended Nested Relational Data Model $S'$. Using $S'$, the *query compilation* phase of RQP translates a given Entity SQL query into an equivalent nested relational algebra expression and then replaces each *forward* operator by the corresponding *reverse* operator. During the *top-down data instantiation* phase, an expected query result $R$ is generated which satisfies the nested result schema (that is computed by the *bottom-up query annotation* phase using $S'$) and the input constraints of the user (e.g., the result size). Then, this nested result $R$ is pushed down the reverse query tree operator by operator to the leaves. The data model of the input and output of the reverse operators are nested relations of the extended nested relational data model. The only exception are the leaf operators which take a nested relation as input and create a set of entities and associations as output which satisfy the given Entity Data Model.

Afterwards, as an additional step, the generated entities and associations are mapped back to relational data model which is used to store the data of the Entity Data Model. A straightforward solution to implement this step is to use the update views which are provided by the Entity Framework. Using the update views is not problematic because our goal is to functional test the query execution (phases 1-3 above) of the Entity Framework and not the update capabilities.

For example, in order to reverse process the Entity SQL query shown in Section 12.2, the given Entity Data Model (see Figure 12.2) is translated into a extended nested relational data model (see structure of tables in Figure 12.3 and Figure 12.4). Subsequently, RQP compiles the given test query into a reverse query tree using the reverse nested relational algebra and then computes the nested result schema of that reverse query tree (which consists of the two attributes $Lineitem.price$ and $Lineitem.MyOrder.orderdate$). In the next step, RQP generates an expected result which satisfies the nested result schema and pushes that result down to the leaves of the reverse query tree which generate a set of $(Rush)Orders$ as well as $Lineitem$ entities and their associations for the given Entity Data Model (shown in Figure 12.2). These entities and associations are then mapped back to the tables $orders$ and $lineitem$ (defined by the mapping in Figure 12.2) in order to generate the test database, which means that e.g., the values of the *foreign-key* column $l\_oid$ must be created from the associations between the $(Rush)Orders$ and $Lineitem$ entities.

| **Orders**<br>type: entity<br>keys: {Orders.id, Orders.MyLineitems.id}<br>predicates: $\{Orders.@type = ‘Orders‘ \vert\vert Orders.@type = ‘RushOrders‘\}$ | | | | | |
|---|---|---|---|---|---|
| **id**<br>type: int | **orderdate**<br>type: date | **arrivaldate**<br>type: date | **fee**<br>type: float | **MyLineitems**<br>type: ref(Lineitem)<br>cardinality: * | **@type**<br>type: string |
| | | | | **id**<br>type: int | |
| 1 | 2005-01-01 | NULL | NULL | 1 | Orders |
| 2 | 2005-01-02 | 2005-01-03 | 10 | 2<br>3 | RushOrders |

Figure 12.3: Nested Relations $Orders$

| **Lineitem**<br>type: entity<br>keys: {Lineitem.id}<br>predicates: $\{Lineitem.price >= 0\&\&Lineitem.@type = ‘Lineitem‘\}$ | | | |
|---|---|---|---|
| **id**<br>type: int | **price**<br>type: float | **MyOrder**<br>type: ref(Orders)<br>cardinality: 1 | **@type** |
| | | **id**<br>type: int | |
| 1 | 999 | 1 | Lineitem |
| 2 | 1000 | 2 | Lineitem |
| 3 | 215 | 2 | Lineitem |

Figure 12.4: Nested Relations $Lineitem$

### 12.3.2 Extended Nested Relational Data Model

We define the *Extended Nested Relational Data Model* as an extension of the standard *Nested Relational Data Model* [AHV95]. The standard nested relational data model allows the type of an attribute which is defined by a *schema of a nested relation* to be a set of records (e.g., an attribute $Orders$ inside a schema of a nested relation $Lineitem$ which represents the order that a line-item belongs to) or a simple data type (like $int$, $string$), rather then requiring it to be a simple data type only. A *nested relation* is an instance of a schema of nested relation and a *nested tuple* is a row of such a nested relation. A *path expression* identifies an attribute inside a nested relation (e.g., $Lineitem.Orders.orderdate$ is a path expression which points to the attribute $orderdate$ nested inside an $Orders$ attribute of the nested relation $Lineitem$). The extensions of the standard *nested relational data model* that are necessary to map the Entity Data Model are discussed in this section when necessary.

An Entity type $T$ of the Entity Data Model is mapped to the extended nested relational data model as follows: a base entity type is mapped to a nested relation with the same name and the type $entity$. A property of an entity type which either has a simple or a complex data type is mapped to an attribute with the same name and a corresponding simple data type or a data type which represents set of records. Derived entity types and their properties are mapped to the same nested relation as the base entity type. Thus, a nested relation which represents an entity type $T$ defines all attributes of the entity type $T$ and the attributes of entity types that derive from T. An example of two nested relations which represent instances of the entity types $Orders$, $RushOrders$, and $Lineitem$ of the Entity Data Model in Figure 12.2 is shown in the Figures 12.3 and 12.4. Each base entity type ($Orders$, $Lineitem$) is mapped to one nested relation with the same name. The properties of the two entity types are mapped to the attributes $Orders.id$, $Orders.orderdate$, as well as $Lineitem.id$, $Lineitem.price$ with the same data types. The properties of the derived entity type $RushOrders$ are also mapped to the nested relation $Orders$. A tuple of that nested relation which represents an entity of type $Orders$ (e.g., the nested tuple with $Orders.id = 1$ in Figure 12.3) holds a $NULL$ value for all attributes which are defined by the derived entity type $RushOrders$.

A nested relation defines an additional attribute $@type$ which holds a string value that indicates the entity type $T$ of the nested tuple. The value of this column is restricted to the entity types which are represented by that nested relation (e.g., the $@type$ column of the nested relation $Orders$ in Figure 12.3 is restricted to the values '$Orders$' and '$RushOrders$').

Mapping associations of the Entity Data Model to the extended nested relational data model is also straightforward. The associations are implemented by an attribute (called *association* attribute) with the same name and the type *ref* (e.g., the association $MyOrder$ is implemented as an attribute with the same name in the nested relation $Lineitem$). Moreover, the association attribute (e.g., $Lineitem.MyOrder$) holds a set of nested attributes (called *reference* attributes) with the names of the key attributes of the referred entity (e.g., $Lineitem.MyOrder.id$). The type *ref* of the association attribute can be seen as a *foreign-key* which constrains the values of this attribute to the key values of the referred entity. The *cardinality* of the association attribute is an extension of the standard nested relational data model and is used to represent the cardinality of an association. For instance, a $Lineitem$ entity refers to exactly one $Orders$ entity. Thus, the cardinality of the association attribute $Lineitem.MyOrder$ is 1.

Moreover, a schema of a nested relational must be able to hold the constraints of the Entity Data Model (keys, predicates) in order to be suitable for the *bottom-up query annotation* phase of RQP. Thus, we extend the standard nested relational data model in such a way that a nested schema can also hold a set of $keys$ and $predicates$ of a nested relation. For instance, the $keys$ of the nested relation $Orders$ are on the attributes $Orders.id$ and $Orders.MyLineitems.id$. This means, that the values of these attributes must be unique in the nested relation $Orders$.

### 12.3.3 Reverse Nested Relational Algebra

The operators of the Reverse Nested Relational Algebra that are presented in this section allow the reverse processing of a subset of possible Entity SQL queries. For each reverse operator of that algebra, we first discuss the Entity SQL expressions that are mapped to this operator during *query compilation* of RQP. Analog to the reverse relational algebra in Part II where the definition, e.g. of the *reverse* selection operator is based on the *forward* selection operator of the relational algebra, we first define the *forward* operator of the Nested Relational Algebra which implements the *forward* processing capabilities of certain Entity SQL expressions and then we define the *reverse* operators of the *Reverse* Nested Relational Algebra based on the definition of the corresponding *forward* operator. The implementation details of the bottom-up query annotation phase and top-down data instantiation phase of RQP for that algebra are omitted for brevity.

**Reverse ScanEntity ($ScanEntity^{-1}_{name,alias}$):**

During query compilation, RQP maps an entity set which is listed in the `FROM` clause of an Entity SQL query to a *reverse ScanEntity* operator.

The *forward ScanEntity* operator of the nested relational algebra scans an entity set with a given $name$ and produces a nested relation as output (the mapping was described in Section 12.3.2). The output nested relation has the name of the scanned entity set. For instance, the output of a forward ScanEntity operator which scans the entity set $Orders$ of the Entity Data Model in Figure 12.2 could be a nested relation that is shown in Figure 12.3. Moreover, the forward ScanEntity operator implements the renaming of an entity set to an optional $alias$.

Accordingly, the *reverse ScanEntity* operator gets a nested relation as input and creates a set of entities as well as the associations of a given Entity Data Model as output. With the help of the column $@type$ in the input, the reverse ScanEntity operator instantiates the correct entity type and initializes the property values of the entities using the attribute values of the nested tuples. The associations are created by dereferencing the attribute values which represent the associations. For example, a reverse ScanEntity operator which gets the nested relation of Figure 12.3 as input, creates one $Orders$ and one $RushOrders$ entity of the Entity Data Model in Figure 12.2 as output. The association $MyLineitems$ is instantiated by dereferencing the values of the attribute $Orders.MyLineitems.id$ (e.g., for the entity which is created for the $Orders$ tuple with $id = 1$, an association to a $Lineitem$ entity with $id = 1$ is instantiated).

**Reverse Ref-Key Join ($\bowtie^{-1}_{\{r_1,\ldots,r_n\}=\{k_1,\ldots,k_n\}}$):**

Navigations along associations inside an Entity SQL query are mapped to a *reverse Ref-Key Join* operator during the compilation phase of RQP. An association of the Entity Data Model is rep-

| **Orders** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **id** | **orderdate** | **arrivaldate** | **fee** | **MyLineitems** | | | | **@type** |
| | | | | **id** | **price** | **...** | **@type** | |
| 1 | 2005-01-01 | NULL | NULL | 1 | 999 | ... | Lineitem | Orders |
| 2 | 2005-01-02 | 2005-01-03 | 10 | 2 | 1000 | ... | Lineitem | RushOrders |
| | | | | 3 | 215 | ... | Lineitem | |

Figure 12.5: Output/Input of the Forward/Reverse Ref-Key Join

resented by a set of reference attributes ($r_1, ..., r_n$) in the extended nested relational data model which point to the $key$ attributes ($k_1, ..., k_n$) of a nested relation.

The *forward Ref-Key Join* operator gets the two nested relations which participate in the association as input: the input which defines the reference attributes is called the *reference input* and the other input which defines the key attributes is called the *key input*. The forward Ref-Key Join operator joins the two input relations in the following way: it replaces the values of the reference attributes of all tuples in the reference input by the nested tuples of the key input which have the same value for the key attributes. Thus, the forward Ref-Key is similar to an equi-join with a join predicate which expresses the equality of the reference and the key attributes, i.e. $r_1 = k_1 \&\& ... \&\& r_n = k_n$.

For example, the navigation $Orders.MyLineitems$ in the following Entity SQL query is implemented as a forward Ref-Key Join with the reference attribute $Orders.MyLineitems.id$ and the key attribute $Lineitem.id$.

```
SELECT Orders.MyLineitems as lineitems
FROM Orders
```

If this join gets the nested relations of Figure 12.3 and Figure 12.4 as input, then it replaces the values of the attribute $Orders.MyLineitems.id$ with tuples of the nested relation $Lineitem$ which have the same value for the attribute $Lineitem.id$, i.e. $Orders.MyLineitems.id = Lineitem.id$ is true in the output. Figure 12.5 shows the output of that join.

Correspondingly, the *reverse Ref-Key Join* splits a given input into two nested relations to create its output (called the *reference* and the *key output*). The *key output* is created by extracting the nested tuples in the input relation which are identified by the reference attribute. For example, the reverse Ref-Key Join $\bowtie^{-1}_{\{Orders.MyLineitems.id\}=\{Lineitem.id\}}$ which gets the nested relation in Figure 12.5 as input, extracts the nested tuples which are identified by the reference attribute ($Orders.MyLineitems.id$) in order to create the *key output*. Afterwards, the reverse Ref-Key Join operator deletes all attributes in the input relation which are at the same level as the reference attribute, but not the reference attribute itself, in order to produce the reference output. In our example, the attributes $Orders.MyLineitems.price$, ..., and $Orders.MyLineitems.@type$ in the input relation (Figure 12.5) are deleted to create the *reference output*. The two output relations

| lineitems | | | |
|---|---|---|---|
| **id** | **price** | **...** | **@type** |
| 1 | 999 | ... | Lineitem |
| 2 | 1000 | ... | Lineitem |
| 3 | 215 | ... | Lineitem |

Figure 12.6: Output/Input of the Forward/Reverse Projection

of that reverse join are shown by the nested relation in Figure 12.3 (*reference output*) and the nested relation in Figure 12.4(*key output*).

**Reverse Projection ($\pi^{-1}_{\{p_1,...,p_n\},\{a_1,...,a_n\}}$):**

The compilation phase of RQP maps the `SELECT` clause of an Entity SQL query to the *reverse projection* operator. The projection list ($p_1, ..., p_n$) of this operator defines a set of *path expressions*. Optionally, aliases ($a_1, ..., a_n$) can be given for each path expression in the projection list.

The *forward projection* of the extended nested relational algebra is similar to the projection operator of the relational algebra. It pulls up the attributes in the nested input relation which are identified by the path expressions to the top-level and renames these attributes according to the given aliases to create the output. All attributes in the input that are not in the projection list are deleted in the output.

For example, a forward projection which implements the `SELECT` clause of the Entity SQL query shown for the reverse nested selection operator, pulls-up the attribute which is identified by the given path expression $Orders.MyLineitems$ to the top-level of relation and renames this attribute by the alias $lineitems$. All other attributes (e.g., $Orders.id$, $Orders.orderdate$, ..., $Orders.@type$) are deleted in the output. If this projection operator gets the nested relation in Figure 12.5 as input, then the output consists of two nested tuples shown Figure 12.6: the first tuple holds the $Lineitem$ with $id = 1$, and the second tuple holds the two $Lineitems$ with $id = \{2, 3\}$.

The *reverse projection* operator reverts the forward projection: It takes the attributes in the input relation that are identified by the aliases ($a_1, ..., a_n$) and initializes the attributes in the output relation which are identified by the corresponding path expressions ($p_1, ..., p_n$) of the projection list. The values of all other attributes that are deleted by the forward projection must be generated by the reverse projection. For instance, the reverse projection operator $\pi^{-1}_{\{Orders.MyLineitems\},\{lineitems\}}$ which gets the nested relation in 12.6 as input, creates its output by initializing the attribute $Orders.MyLineitems$ in the output with the values of the $lineitems$ attribute in the input. Afterwards, the values are generated for the attributes which are deleted by the forward projection (e.g., $Orders.id$, $Orders.orderdate$, ..., $Orders.@type$ in Figure 12.5).

| **o** | | | | | |
|---|---|---|---|---|---|
| **id** | **orderdate** | **arrivaldate** | **fee** | **MyLineitems** | **@type** |
| | | | | **id** | |
| 2 | 2005-01-02 | 2005-01-03 | 10 | 2<br>3 | RushOrders |

Figure 12.7: Output/Input of the Forward/Reverse Selection

**Reverse Selection ($\sigma_p^{-1}$):**

The `WHERE` clause of an Entity SQL query is mapped to a *reverse selection* operator during the compilation of RQP. The selection predicate $p$ is an arbitrarily complex predicate which uses path expressions instead of attributes to express the filter condition (e.g., $Orders.price >= 100$). An `IS OF` predicate in the `WHERE` clause of a Entity SQL query is also mapped to a reverse selection operator with a simple equality predicate. For example, the `WHERE` clause of the following Entity SQL query is implemented by the predicate $o.@type = `RushOrders`$.

```
SELECT o
FROM Orders o
WHERE o IS OF RushOrders
```

As in the relational algebra, the *forward selection* operator filters all tuples in the input which do not satisfy the predicate. The path expressions which are used in the predicate must point to a scalar value and not a set of records; e.g., a forward selection with the predicate $Orders.MyLineitems.price > 100$ on the input (Figure 12.5) is not allowed because the path expressions points to a set of integer values. However, a forward selection with the predicate $o.@type = `RushOrders`$ which gets the nested relation in Figure 12.5 as input would be allowed. The output of this operator is shown by Figure 12.7:

In the simplest case, the *reverse selection* operator can be implemented as the identity function. However, the reverse selection can also add some additional nested tuples to its output which satisfy the negative selection predicate. The number of tuples which are added by the reverse selection could be another parameter which is provided by the user as input for the top-down data instantiation phase. For example, if the *reverse selection* with the predicate $o.@type = `RushOrders`$ is executed on the input nested relation in Figure 12.5, then it could add some nested tuples to the output which satisfy $o.@type! = `RushOrders`$ and the constraints ($Orders.@type = `Orders`||Orders.@type = `RushOrders`$) of the schema of the nested relation in Figure 12.3.

## 12.4 Related Work

To the best of our knowledge, there has been almost no work on the automatic result verification for the functional testing of a query language. [Slu98] suggested the method to execute the test query on a comparable query processor to obtain the expected query result which can be compared to the actual result for verification. The software engineering community has also addressed this problem (known as the computation of a *test oracle*) for the testing of different applications [Bin96] but not for the testing of a query language.

The work in [BCT06] tackles the problem of controlling some characteristics like the cardinality of the (intermediate) query results for a given test query by generating query parameters for a given test query. This approach could be used to partially verify the actual query results by comparing the controllable characteristics of the expected query result with the same characteristics of the actual result. However, that approach is not as powerful as our approach.

Most existing approaches for the functional testing of a query language focused on the generation of test queries and test database instances (see Section 2.2.2). These approaches are often used to first generate a set of test database instances and test queries and then execute these test queries on the generated database instances to find some errors without verifying the actual query result.

# Chapter 13

# Other Applications

*Science never solves a problem without creating ten more.*

*– George Bernard Shaw, 1856-1950 –*

Apart from test database generation, RQP has also a lot of other potential applications. As its main contribution this section sketches how RQP can be useful for those applications. However, much additional research is required in order to further explore these applications.

**Data Security:** One future application of RQP is to study the query-view security problem. This problem addresses the question if a set of views that are published over the same database instance disclose any information about a query that a potential attacker wants to execute. RQP could be used to generate different test databases from the published view data (queries and query results) in order to test the confidentiality of the view data [MS04].

**SQL Debugging:** Another practical application of RQP is to debug database applications with embedded SQL code. If a query produces the wrong query results, then RQP can be used to step-wise reverse engineer the query based on its query plan and find the operators that are responsible for the wrong query results; e.g., a wrong or missing join predicate.

**Program Verification:** RQP can also be an important component for Hoare's Grand Challenge project of program verification [HM05]. In order to prove the correctness of a program, all possible states of a program must be computed. In order to compute all states of a database program (e.g., Java plus embedded SQL), RQP is needed for finding all necessary conditions of the database in order to reach certain program states.

**Updating Views:** The SQL standard is conservative and specifies that only views on base tables without aggregates are updatable. Many applications make heavy use of SQL view definitions and, therefore, require a more relaxed specification of updatable views. For example, Microsoft's ADO.NET allows the client-side update of data, regardless of the kind of view that was used to generate that data. The reason why SQL is conservative is that updates to certain views are ambiguous. RQP could be used in order to find all possible ways to apply an update (possible infinitely many). Additional application code can then specify which of these alternatives should be selected.

**Database Sampling, Compression:** Some databases are large and query processing might be expensive even if materialization and indexing is used. One requirement might be to provide a compressed, read-only variant of a database that very quickly gives approximate answers to a predefined set of parametrized queries. Such database variants can be generated using RQP in the following way: First, take a sample of the queries (and their parameters) and execute those queries on the original (large) database. Then, use RQP on the query results and the sample queries in order to find a new (smaller) database instance.

# Part IV

# Symbolic Query Processing

# Chapter 14

# Motivating Applications

*Logic will get you from A to B. Imagination will take you everywhere.*

*– Albert Einstein, 1879-1955 –*

The complexity of database management systems (DBMS) makes the addition of new features or the modifications of existing features difficult. The impact of the modifications on system performance and on other components is hard to predict. Therefore, after each modification, it is necessary to run tests to evaluate the relative system improvements and the overall system quality under a wide range of test cases and workloads.

Today a common methodology for testing a database system is to generate a comprehensive set of test databases and then study the before-and-after system behavior by executing many test queries over the generated data. Current database generation tools allow a user to define the sizes and the data characteristics (e.g., value distributions and inter/intra-table correlations) of the base tables (see Section 2.2.2). Based on the generated test databases, the next step is to either create test queries manually, or stochastically generate many valid test queries by query generation tools such as RAGS [Slu98] or QGEN [PS04], and then execute them to test the DBMS.

Unfortunately, the current testing methodology is inadequate to test individual features of the database systems because very often it is necessary to control the input/output of the intermediate operators of a query during a test. For example, assume that the technical team of a DBMS product wants to test how a newly designed memory manager influences the performance of multi-way hash join queries (i.e., how the per-operator memory allocation strategy of the memory manager affects the resulting execution plans). Figure 14.1 shows such a sample test case (figure extracted from [BCT06]). A test case is a parametric query $Q_P$ with a set of constraints defined on each operator. In Figure 14.1, the test query of the test case first joins a large filtered table $S$ with a

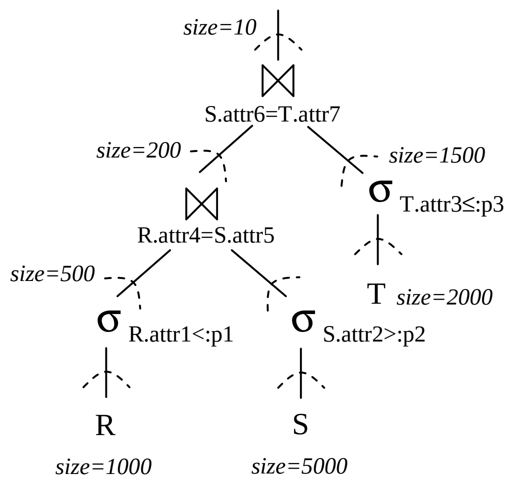


Figure 14.1: A test case: a query with operator constraints

filtered table $R$ to get a small join result. Then the small intermediate join result is joined with a filtered table $T$ to obtain a small final result. Since the memory requirements of a hash join is determined by the size of its inputs, it would be beneficial if the input/output of each individual operator in the query tree can be controlled/tuned according to the test requirements. For example, the memory allocated to $\bowtie_{S.attr_6=T.attr_7}$ by the memory manager, can be studied by defining the output cardinality constraint on the join $\sigma(R) \bowtie \sigma(S)$ and the output cardinality constraint on $\sigma(T)$ in the test case. However, even though the tester can instruct the database engine to evaluate the test query by a specific physical execution plan (e.g., fixing the join order and forcing the use of hash-join as the join algorithm), there is currently no easy way to control the (intermediate) results of a query because those results depend on the *content* of the test database.

Testing the features of DBMS requires the execution of a test query on a test database. Usually the test query is given by the testers (e.g., the one in Figure 14.1). In general, a good test database should cover the test cases (i.e., the database content is possible to give the desired intermediate query results for a test query when the query is executed on it). However, existing test database generators do not take the test query as part of the inputs. Therefore, unless with intensive manual tuning on the database content, it is hard to guarantee that executing the test query on the test database can obtain the desired (intermediate) query results that are defined in the test case. Figure 14.2 (a) shows this problem. In the figure, there are two test cases, $T_1$ and $T_2$ (denoted by dots) and there are three generated test database instances (denoted by squares). The three generated test databases (Databases 1, 2, and 3) do not cover test case $T_2$ at all (i.e., executing test query $Q_P$ of $T_2$ on Databases 1, 2, and 3 can never fulfill the constraints that are defined in $T_2$). Even if a test

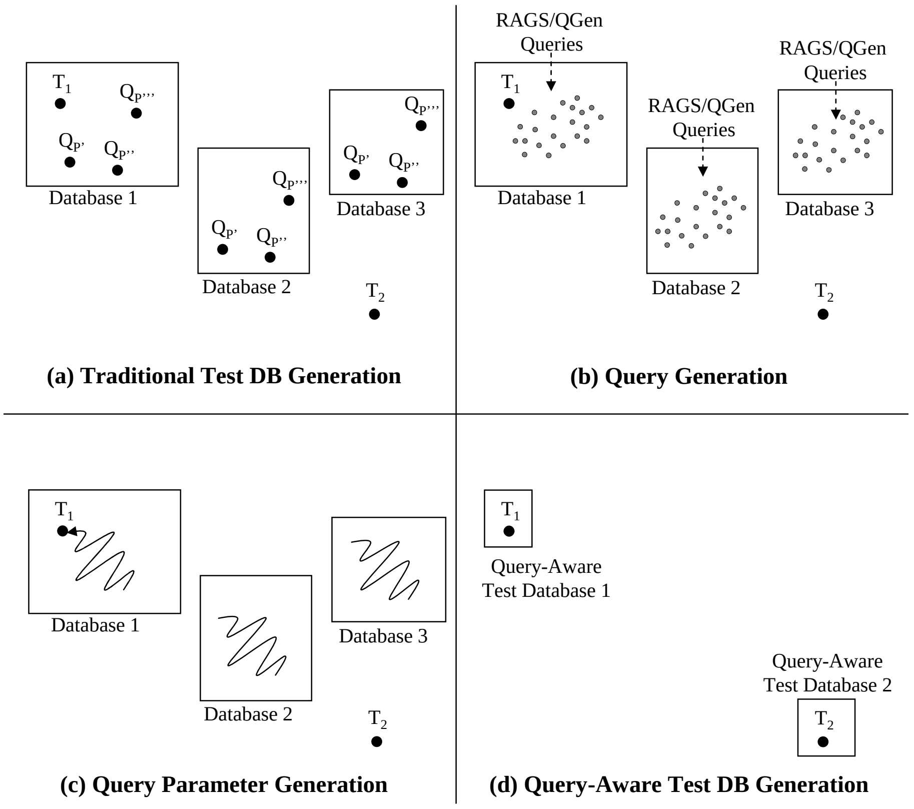


Figure 14.2: The DBMS Feature Testing Problem

database covers a test case (e.g., Database 1 covers $T_1$), it is difficult to manually find the correct parameter values $P$ of test query $Q_P$ such that the query results match the constraints in the test case. For instance, it is unlikely that instantiating test query $Q_P$ in Figure 14.2 (a) with three sets of parameter values $P'$, $P''$, $P'''$ manually can match the requirements of test case $T_1$.

Given a test database, query generation tools such as RAGS and QGEN generate many queries in order to cover a variety of test cases. However, RAGS and QGEN were not designed for testing an individual DBMS component. To test an individual DBMS component, the desired test query is usually given by a tester (e.g., the query in Figure 14.1). In this situation, RAGS and QGEN may need to generate many queries in order to match the test query and the requirements of the test case (see Figure 14.2 (b)). In addition, RAGS and QGEN also rely on what databases they are working on or otherwise they never can generate a test query that matches the test case (e.g., $T_2$).

The problem of testing DBMS features has been pointed out by [BCT06]. Given a test database $D$, a parametric conjunctive query $Q_P$, and cardinality constraints $C$ over the sub-expressions of $Q_P$, they studied how to find the parameter values $P$ of $Q_P$ such that the output cardinality of each operator in $Q_P$ fulfills $C$. In their pioneering work, they found that their formulation of the problem is $\mathcal{NP}$-hard. Their approach is illustrated in Figure 14.2 (c). Given the predefined test databases (e.g., Databases 1, 2, and 3), it may be possible that there are no parameter values that can let test query $Q_P$ match the requirements in test case $T_2$. Even if a test database covers a test case (e.g., Database 1), since the solution space is too large, only simple select-project-join queries with single-sided predicates (e.g., $p_1 \leq a$ or $a \leq p_2$) or double-sided predicates (e.g., $p_1 \leq a \leq p_2$) (where $a$ is an attribute and $p_1$ and $p_2$ are parameter values) can be supported.

We observe that the test database generation process is the main culprit of ineffective DBMS feature testing. Currently, test databases are generated without taking the test queries as input. Thus the generated databases cannot guarantee that executing the test query on them can obtain the desired (intermediate) query results that are specified in the test case. Therefore, the only way for meaningful testing is to do a painful trial-and-error test database generation process (i.e., generating test databases one-by-one, or manually tune the database content, until we find a good test database that matches the test case), and execute queries generated by RAGS/QGEN, or execute test queries with parameters instantiated by [BCT06].

In this thesis, we address the DBMS feature testing problem in a different and novel way: Instead of first generating a test database and then seeing if it is possible for the test query to obtain the desired query results that match the test case (otherwise use a trial-and-error approach to find another test database), we propose to generate a specific test database for each test case (see Figure 14.2 (d)). To that end, we propose a new technique called *Symbolic Query Processing* or SQP, for short. Given a database schema $S$, a logical query plan $Q$, and a set of user-defined constraints $C$ on each query operator, SQP directly generates a database $D$ such that executing $Q$ on $D$ guarantees that the user requirements imposed on the query operators are fulfilled.

Consequently, SQP implements the *Test Case Aware Database Generation* for the testing of individual DBMS components. Traditional database generators (see Section 2.2.2) allow constraints to be defined only on the base tables (e.g., a join key distribution is defined on the base tables). As a result, a tester cannot specify operator constraints (e.g., the output cardinality of a join) in an explicit way. SQP allows a user to annotate constraints on each operator and on each base table directly, and thus the users can easily get a meaningful test database for a distinct test case.

The test databases generated by SQP can be used in a number of ways for the testing of DBMS components. For example, in addition to testing the memory manager, testers can use SQP to generate a test database that guarantees the size of the intermediate join results to test the accuracy of the cardinality estimation components (e.g., histograms) inside a query optimizer by fixing the join order.[1] As another example, testers can use SQP to generate a test database that guarantees the input and the output sizes (the number of groups) for an aggregation operator (GROUP-BY) in order to evaluate the performance of the aggregation algorithm under a variety of cases such as in multi-way join queries or in nested queries.

[1]However, it is inapplicable to test the join reordering feature of a query optimizer directly because in this case the *physical* join ordering should not be fixed by the tester; and the intermediate cardinalities guaranteed by SQP may affect the optimizer to return a different physical execution plan with different intermediate results.

**Contributions:** The main contribution of this part is the conceptual framework for SQP and a prototype implementation called QAGen (Query-Aware Database Generator). QAGen can generate test databases for a variety of complex queries such as TPC-H [TPCb] queries efficiently. In some cases, the test database generation process still involves solving an $\mathcal{NP}$-hard problem such that QAGen generates a test database in which the test query execution gets approximate cardinalities instead of exact cardinalities as defined on the test case. For example, QAGen may generate a test database in which executing the test query in Figure 14.1 gets a join result with 12 tuples rather than 10 tuples in the join $S.attr_6 \bowtie T.attr_7$. In practice, this relaxation is desirable because for testing the feature of a DBMS, it usually does not matter whether or not the final join result size exactly matches the test case requirements. In many cases, a good approximate answer is sufficient and it turns out that such relaxation allows QAGen to efficiently support a much richer class of SQL queries.

Sometimes it would be advantageous to add new kinds of constraints to an operator in addition to the cardinality constraint during testing. For instance, the aggregation (GROUP-BY) operator may not only need to control the output size (i.e., the number of groups), but may also need to control how to distribute the input to the predefined output groups (i.e., some groups have more tuples while others have fewer). Thus, QAGen is designed to be extensible in order to incorporate new operator constraints easily.

The final contribution of this part is the design and implementation of a semi-automatic DBMS testing framework. The framework automates the step of manually constructing DBMS test cases like the one in Figure 14.1. As a result, testers may not need to explicitly specify the constraint details (e.g., the cardinality constraint $size = 500$ in Figure 14.1), but let the framework to automatically create and execute a set of test cases which cover different testing requirements.

**Outline:** The remainder of this part is organized as follows: Chapter 15 gives an overview of SQP. Chapter 16 to 18 describe the prototype implementation for SQP called QAGen. Chapter 19 presents the semi-automatic DBMS testing framework that generates and executes test cases. This framework is built on top of QAGen. Chapter 20 presents the experimental results. Chapter 21 discusses related work.

Chapter 15

# SQP Overview

*We can't solve problems by using the same kind of thinking we used when we created them.*

*– Albert Einstein, 1879-1955 –*

## 15.1 Problem Statement and Decidability

This thesis addresses the following problems: (1) The first problem is to identify a subset of constraints $C$ in a given set of constraints (e.g., output cardinality, distribution of a certain attribute) that could be controlled for each sub-expression (i.e., output of an operator) of a logical query plan $Q$, and a database schema $S$ (including integrity constraints). (2) Given a valuation $V$ for each constraint in $C$ (e.g., concrete values for the output cardinality or a concrete distribution of an attribute), the second problem is to find a test database instance $D$ that satisfies $V$ and $S$. In general, there are many different database instances which can be generated for a given logical query plan $Q$ and the constraint valuation $V$. The purpose of this thesis is to find any possible database instance $D$.

**Theorem 15.1** *Given a logical query plan $Q$, a valuation $V$ of the constraints $C$, and a database schema $S$, it is undecidable whether there exists a database instance $D$ that satisfies $V$ and $S$ or not.*

**Proof (Sketch) 15.2** *We can use the same argument as in Section 4.1 for RQP. In order to show that SQP is undecidable, we reduce the query equivalence problem to SQP. Since the query equivalence problem is undecidable [Klu80], we prove the theorem.*

*Let $Q_1$ and $Q_2$ be two arbitrary SQL queries. In order to decide whether $Q_1$ and $Q_2$ are equivalent, we can use SQP to decide whether a database instance $D$ exists for the query $Q = \chi_{COUNT(*)}((Q_1 - Q_2) \cup (Q_2 - Q_1))$, a cardinality constraint on the query result $R$, and a valuation of that constraint which defines that the cardinality $c$ of $R$ must satisfy $c > 0$. Moreover, $D$ should meet the constraints of the given database schema $S$. If SQP can find such a database instance $D$, then $Q_1$ and $Q_2$ are not equivalent (i.e., if $Q_1$ and $Q_2$ would be equivalent, the result of $Q$ must be empty). Otherwise, if SQP can not find such a database instance $D$, it immediately follows that $Q_1$ and $Q_2$ are equivalent.*

Furthermore, there are obvious cases where no $D$ exists for a given $V$ and $Q$ (e.g., if values in $v$ violate each other; e.g., when the output cardinality of a selection operator is greater than the output cardinality of its child). Again, the approach presented in this thesis, cannot be complete. It is a best-effort approach: it will either fail (return an *error* because it could not find a $D$) or return a valid $D$.

## 15.2 SQP Architecture

SQP is a framework that gives a best effort solution for the problem statement discussed before. The data generation process of SQP consists of two phases: (1) the symbolic query evaluation phase, and (2) the data instantiation phase. The goal of the symbolic query evaluation phase is to capture the user-defined constraints on the query into the target database. To process a query without concrete data, SQP integrates the concept of symbolic execution [Kin76] from software engineering into traditional query processing. Symbolic execution is a well known program verification technique, which represents values of program variables with symbolic values instead of concrete data, and manipulates expressions based on those symbolic values. Borrowing this concept, SQP first instantiates a database which contains a set of symbols instead of concrete data (thus the generated database in this phase is called a *symbolic database*). Figure 15.1 shows an example of a symbolic database with three *symbolic relations* $R$, $S$ and $T$. Essentially, a symbolic relation is just a normal relational table which consists of a set of *symbolic tuples*. Inside each symbolic tuple, the values are represented by *symbols* rather than by concrete values. For example, symbol $a1$ in symbolic relation $R$ in Figure 15.1 represents any value under the domain of attribute $a$. The formal definition of these symbolic database related terms will be given in Chapter 17. For the moment, let us just treat the symbolic relations as normal relations and treat the symbols as variables. Since the symbolic database is a generalization of relational databases and provides an abstract representation for concrete data, this allows SQP to control the output of each operator of the query.

The symbolic query evaluation phase leverages the concept of traditional query processing. First, the input query is analyzed by a *query analyzer*. Then, the user specifies her desired requirements

| | a | b |
|---|---|---|
| t1: | a1 | b1 |
| t2: | a2 | b2 |

Table $R$

| | c | d |
|---|---|---|
| t3: | c1 | d1 |
| t4: | c2 | d2 |
| t5: | c3 | d3 |
| t6: | c4 | d4 |

Table $S$

| | a | b = c | d |
|---|---|---|---|
| t7: | a1 | b1 | d1 |
| t8: | a1 | b1 | d2 |
| t9: | a1 | b1 | d3 |
| t10: | a2 | b2 | d4 |

$R \bowtie_{b=c} S$

| | e | f |
|---|---|---|
| t11: | e1 | f1 |
| t12: | e2 | f2 |

Table $T$

Figure 15.1: Example of pre-grouped input data

on the operators of the query tree. Afterwards, the input query is executed by a *symbolic query engine* just like in traditional query processing; i.e., each operator is implemented as an iterator, and the data flows from the base tables up to the root of the query tree [Gra93]. However, unlike in traditional query processing, the symbolic execution of operators deals with symbolic data rather than concrete data. Each operator manipulates the input symbolic data according to the operator's semantics and user-defined constraints, and incrementally imposes the constraints defined on the operators to the symbolic database. After this phase, the symbolic database is a query-aware database that captures all constraint on the intermediate query results defined in the test case (but without concrete data).

The data instantiation phase follows the symbolic query evaluation phase. This phase reads the tuples from the symbolic database that are prepared by the symbolic query evaluation phase and instantiates the symbols in the tuples by a constraint solver. The instantiated tuples are then inserted into the target database.

To allow a user to define different test cases for the same query, the input query of SQP is in the form of a relational algebra expression. For example, if the input query is a 2-way join query $(\sigma_{age>p_1} customer \bowtie orders) \bowtie lineitem$, then the user can specify a join key distribution (e.g., a Zipf distribution) between the line items and the orders that join with customers with an age greater than $p_1$. On the other hand, if the input query is $(orders \bowtie lineitem) \bowtie \sigma_{age>p_1} customer$, then the user can specify the join key distribution between all orders and all lineitems.

Figure 15.2 shows the general architecture of SQP. It consists of the following components: a Query Analyzer, a Symbolic Query Engine, a Symbolic Database and a Data Instantiator.

### 15.2.1 Query Analyzer

In the beginning of the symbolic query evaluation phase, SQP first takes a query plan $Q$ and the database schema $S$ as input. The query $Q$ is then analyzed by the query analyzer component in SQP. The query analyzer has two functionalities:

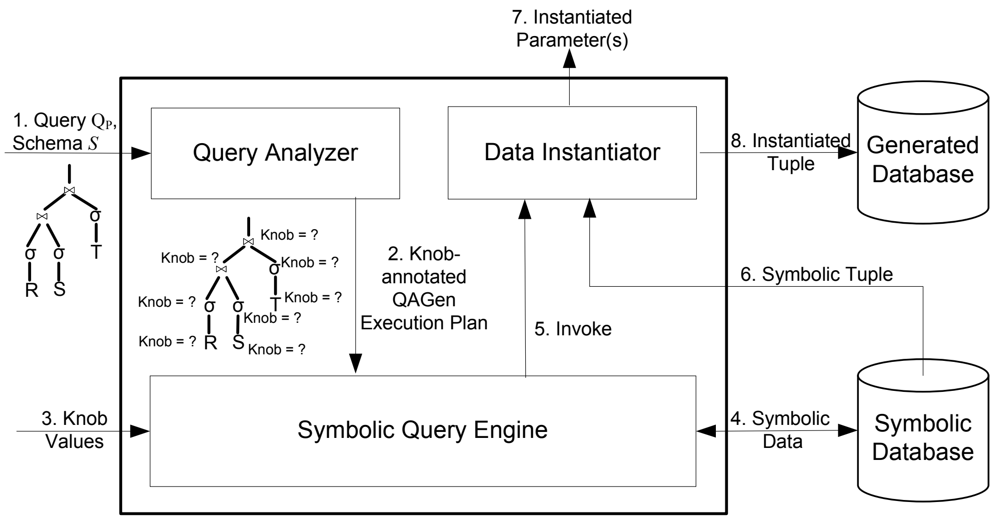


Figure 15.2: SQP Architecture

**(1) Correct knob selections:** The query analyzer analyzes the input query and determines which constraints (*knobs*) are available for each operator. A knob can be regarded as a parameter of an operator that controls the output. A basic knob that is offered by SQP is the *output cardinality* [1]. This knob allows a user to control the output size of an operator. However, whether such a knob is applicable depends on the operator and its input characteristics. This step is fairly simple and the query analyzer can accomplish it without analyzing the input data of each operator. Thus,the query analyzer essentially annotates the appropriate knob(s) to each operator. As a result, the output of the query analyzer is an annotated query tree with the appropriate knob(s) on each operator. As an example, for a simple aggregation query `SELECT MAX(a) FROM R`, the cardinality constraint knob should not be available for the aggregation operator ($\chi$), because the output cardinality is always one if $R$ is not empty or zero if $R$ is empty. Chapter 16 will present the details of this step.

**(2) Assign physical implementations to operators:** As shown above, different knobs are available under different input characteristics. In general, different (combinations of) knobs of the same operator need separate implementation algorithms. Moreover, even for the same (combination of) knobs of the same operator, different implementation algorithms are conceivable (this is akin to traditional query processing where an equi-join operation can be implemented by a hash-join or a sort-merge join). Consequently, the other function of the query analyzer is to assign the correct (knob-supported) implementation to an operator. As a result, the output of the query analyzer is

[1]The output cardinality of an operator can be specified as an absolute value or as a selectivity. Essentially they are equivalent.

a knob-annotated query execution plan. Chapter 17 will present the implementation algorithms for each symbolic operation implemented in QAGen. In general, the job of the query analyzer is analogous to the job of the query optimizer in traditional query processing. However, in the current version of QAGen, only one implementation algorithm for each (combination of) knob is available. If there are more than one possible implementation of the same symbolic operation, then the query analyzer can be extended to be a query optimizer.

### 15.2.2 Symbolic Query Engine and Database

The symbolic query engine of SQP is the heart of the symbolic query processing phase and it is similar to a normal query engine. However, before the symbolic query engine starts execution, the user can specify the value(s) for the available knob(s) of each operator in the knob-annotated execution plan. It is fine for a user to fill up values for some but not all knobs. In this case, the symbolic query engine will evaluate those operators by using default knob values which are defined by the creator(s) of those knob(s).

The symbolic query execution is also based on the iterator model. That is, an operator reads in symbolic tuples from its child operator(s) one-by-one, processes each tuple, and returns the resulting tuple to the parent operator. Similar to traditional query processing, most of the operators in symbolic query processing can be processed in a pipelined mode, but some cannot. For example, the equi-join operator is a blocking operator under a special case. In these cases, the symbolic query engine materializes the intermediate results into the symbolic database if necessary. Moreover, the table in a query tree is regarded as a special operator. During its $open()$ method, the table operator initializes a symbolic relation based on the input schema $S$ and the user-defined constraints (e.g., table sizes) on the base tables.

During processing, a symbolic operation evaluates the input tuples according to its own semantics. On the one hand, it imposes additional constraints to each input tuple in order to reflect the constraints defined on the operator. On the other hand, it controls its output to its parent operator so that the parent operator can work on the right tuples. As a simple example, assume the input query is a simple selection query $\sigma_{a \geq p_1} R$ on symbolic relation $R$ in Figure 15.1 and the user specifies the output cardinality as 1 tuple. Then, if the getNext() method of the selection operator iterator is invoked by its parent operator, the selection operator reads in tuple $t1$ from $R$, annotates a *positive* constraint $[a_1 \geq p_1]$ to symbol $a_1$ and returns tuple $\langle a_1, b_1 \rangle$ to its parent. When the getNext() method of the selection operator is invoked a second time, the selection operator reads in the next tuple $t2$ from $R$, and annotates a *negative* constraint $[a_2 < p_1]$ to symbol $a_2$. However, this time it does *not* return this tuple to its parent, because the cardinality constraint (1 tuple) is already fulfilled.

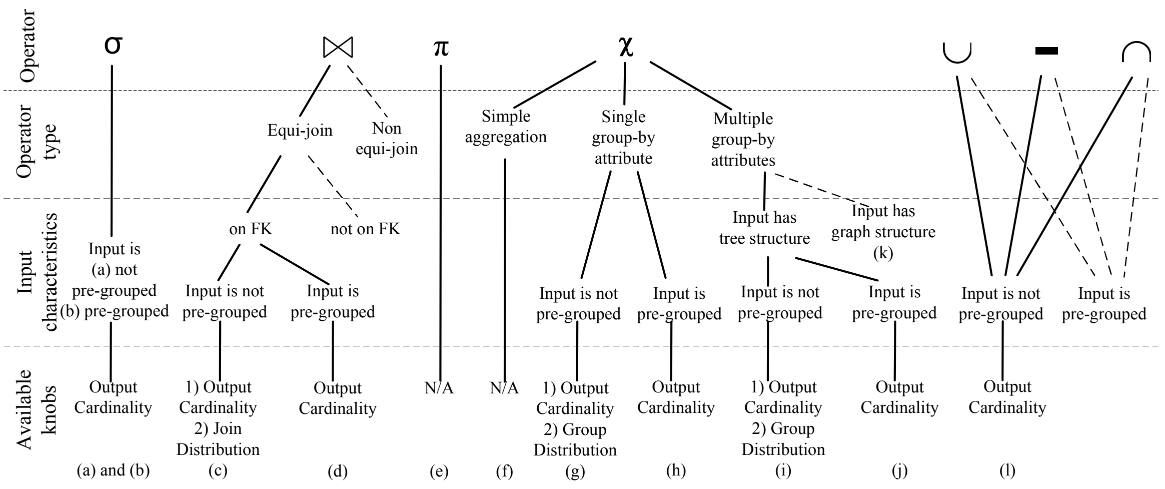


Figure 15.3: QAGen Framework

It is worth noting that sometimes a user may specify some contradicting knob values on the knob-annotated query tree given by the query analyzer. For instance, a user may specify the output cardinality of the selection in the above example as 10 tuples even if she specified table $R$ to have only two tuples. In the following sections of this chapter, we assume that the users are experienced testers and the test case has no contradicting knob values.

### 15.2.3 Data Instantiator

The data instantiation phase starts after the symbolic query engine of SQP has finished processing. The data instantiator reads in the symbolic tuples from the symbolic database and instantiates the symbols inside each symbolic tuple by a constraint solver. In SQP, we treat the constraint solver as an external black box component where it takes a constraint formula (in propositional logic) as input and returns a possible instantiation on each variable as output. For example, if the input constraint formula is $40 < a1 + b1 < 100$, then the constraint solver may return $a1 = 55, b1 = 11$ as output (or any other possible instantiation). Once the data instantiator has collected all the concrete values for a symbolic tuple, it inserts a corresponding tuple (with concrete values) into the target database.

## 15.3 Supported Symbolic Operations

In the following chapters, we consider only a limited class of relational algebra expressions that are able to be processed by our current SQP prototype implementation QAGen. In particular, we consider expressions that use the following relational algebra operators: selection ($\sigma$), projection ($\pi$), equi-join ($\bowtie$), aggregation ($\chi$), union ($\cup$), minus ($-$) and intersection ($\cap$). The set of comparison operators is restricted to $=, \neq, \leq, \geq, <, >, <>$. Furthermore, we assume the number of possible values in the domain of a group-by attribute is greater than the number of tuples to be output for an aggregation operator.

As shown in the previous section, in many cases, the available knobs for an operator depend on its input characteristics. Details about the input characteristics and formal definitions will be given in Chapter 16. Figure 15.3 shows a summary of the class of SQL queries that QAGen supports. The solid lines denote the cases or operators supported by the current version of QAGen. The dotted lines show the cases or operators that the future version of QAGen should support. According to Figure 15.3, the current version of QAGen already suffices to cover 13 out of 22 complex TPC-H queries. In general, supporting new operators (e.g., theta join), or adding new knobs (which may depend on new input characteristics) to an operator is straightforward in QAGen. For example, adding a new knob to an operator simply means incorporating the new QAGen implementation of that operator into the symbolic query engine and then updating the query analyzer about the input characteristics that this new knob depends on.

Chapter 16

# Query Analyzer

*The information encoded in your DNA determines your unique biological characteristics, such as sex, eye color, age and Social Security number.*

*– Dave Berry, born 1947 –*

The query analyzer has two functionalities: (1) the correct knob selection for the operators of a given relational algebra expression, and (2) the assignment of the physical implementation to each operator of the relational algebra expression. QAGen currently supports only one physical implementation for each possible combination of knobs per relational algebra operator. As a result, (2) is straightforward and we do not focus on it. This chapter focuses on (1), which describes how to analyze the query and determine the available knob(s) for each operator in the input query. Figure 15.3 shows the knobs of each operator offered by QAGen under different cases.

The general procedure for the correct knob selection is as follows: First, the query analyzer determines the input characteristics of each operator of the relational algebra expression in order to decide what kinds of knobs are available for each operator. To determine the input characteristics for each operator in the query tree, the query analyzer computes the set of functional dependencies that holds in each intermediate result of the input query in a bottom-up fashion. In Part II, we presented how to compute the functional dependencies for queries in detail. Thus, starting from the base tables, the query analyzer computes the set of functional dependencies that holds in each intermediate result in a bottom-up fashion. Since the definition of the input characteristics of an operator solely depends on the functional dependencies, the type of knobs available for an operator can be easily determined according to Figure 15.3. In symbolic query processing, there are four types of input characteristics: *pre-grouped*, *not pre-grouped*, *tree-structure*, and *graph-structure*. Let $A$ be the set of attributes of the input of an operator. The input characteristic definitions are as follows:

**Definition 16.1** *(Pre-grouped / Not pre-grouped:) Let $A$ be the set of attributes of the input of an operator. Then the input of an operator is* not pre-grouped *with respect to an attribute $a \in A$, iff there is a functional dependency $a \rightarrow \{A - a\}$ (which means that $a$ is distinct) that holds in the input. Otherwise, the input of the operator is* pre-grouped *with respect to attribute $a$.*

**Definition 16.2** *(Tree-structure / Graph-structure:) A set of attributes $A' \subset A$ of the input of an operator has a* tree-structure, *iff either the functional dependency $a_i \rightarrow a_j$ or $a_j \rightarrow a_i$ holds in the input of the operator (for all $a_i, a_j \in A'$ and $a_i \neq a_j$). Otherwise, the set of attributes $A' \subset A$ of the input of the operator has a* graph-structure.

For example, look at the base tables $R$ and $S$ in Figure 15.1. As we will see in the next chapter, all symbols in the base tables are distinct initially. As a result, the initial set of functional dependencies for the base tables can be determined easily; e.g., the base table $R$ in Figure 15.1 contains two functional dependencies: $\{a\} \rightarrow \{b\}$, and $\{b\} \rightarrow \{a\}$. Following the rules in Part II and the definitions above, the intermediate result $R \bowtie_{b=c} S$ (where $c$ is a foreign-key referring to $b$) of the query $(R \bowtie_{b=c} S) \bowtie_{a=e} T$ has three *pre-grouped* attributes $a$, $b$ and $c$ (where $b = c$) and has one attribute $d$ that is not pre-grouped. This is because:

- Initially, the set of functional dependencies of $R$ is $\{\{a\} \rightarrow \{b\}, \{b \rightarrow a\}\}$ and the set of functional dependencies of $S$ is $\{\{c\} \rightarrow \{d\}, \{d\} \rightarrow \{c\}\}$.

- According to the functional dependency calculation rule for joining, two more functional dependencies $\{b\} \rightarrow \{c\}$ and $\{c\} \rightarrow \{b\}$ for the equi-join predicate $b = c$ are added, and $\{c\} \rightarrow \{d\}$ is removed because one tuple from $R$ can join with many tuples from $S$ (due to the foreign-key from $c$ referring to $b$) and the attribute $c$ is initialized with the values of $a$ in the output of the equi-join. The final set of functional dependencies of $F_{out}$ of the intermediate join result $R \bowtie_{b=c} S$ is $\{\{a\} \rightarrow \{b, c\}, \{b\} \rightarrow \{a, c\}, \{c\} \rightarrow \{a, b\}, \{d\} \rightarrow a, b, c\}$.

- Among the set of attributes $A = \{a, b, c, d\}$ in the intermediate result, attribute $d$ functionally determines all attributes in $A$ whereas the others do not. As a result, according to the definition of pre-grouping, $d$ is not *pre-grouped* and $a$, $b$, and $c$ are pre-grouped in the intermediate result.

We use another example to illustrate the concept of tree and graph input characteristics. Assume the following table is an intermediate result of a query:

| a | b | c | d |
|---|---|---|---|
| a1 | b1 | c1 | d1 |
| a2 | b1 | c1 | d2 |
| a3 | b2 | c1 | d2 |
| a4 | b3 | c2 | d1 |

Assume the following functional dependencies hold on the above intermediate result: $\{\{a\} \to \{b, c, d\}, \{b\} \to \{c\}\}$. Following the definitions of *tree-* and *graph-structure*, the attribute set $A = \{a, b, c\}$ has a *tree-structure* because all attributes are functional dependent on each other. On the other hand, the attribute set $A = \{a, b, d\}$ has a *graph-structure* because there is no functional dependency between $b$ and $d$ (i.e., neither $\{b\} \to \{d\}$, nor $\{d\} \to \{b\}$ holds on the intermediate result).

After the input characteristics are determined, the query analyzer annotates the correct knob(s) according to Figure 15.3. As an example, the available knob(s) of an equi-join ($\bowtie$) depends on whether the input is *pre-grouped* or not on the join keys. If the input is pre-grouped, the equi-join can only offer the output cardinality as a single knob (Figure 15.3 case (d)). If the input is not pre-grouped, the user is allowed to tune the join key distribution as well (Figure 15.3 case (c)).

For example, consider a 2-way join query $(R \bowtie_{b=c} S) \bowtie_{a=e} T$ on the three symbolic relations $R$, $S$, and $T$ in Figure 15.1. When symbolic relation $R$ first joins with symbolic relation $S$ on attributes $b$ and $c$, it is possible to specify the join key distribution such as joining the first tuple $t1$ of $R$ with the first three tuples of $S$ (i.e., $t3$, $t4$, $t5$); and the last tuple $t2$ of $R$ joins with the last tuple $t6$ of $S$ (kind of like Zipf distribution [Zip49]). However, after the first join, the intermediate join result of $R \bowtie S$ is *pre-grouped* w.r.t. attributes $a$, $b$ and $c$ (e.g., symbol $a1$ is not distinct on attribute $a$ in the join result). Therefore, if this intermediate join result further joins with symbolic relation $T$ on attributes $a$ and $e$, then the distribution cannot be freely specified by a user. That is because if the first tuple $t11$ of $T$ joins with the first tuple $t7$ of the intermediate results, this implies that $e1 = a1$ and thus $t11$ must join with $t8$ and $t9$ as well.

The above example shows that it is necessary to analyze the query in order to offer the right knobs to the users. For this purpose, the query analyzer parses the input query in a bottom-up manner (i.e., starting from input schema $S$) and incrementally pre-computes the output characteristics of each operator (e.g., annotates an attribute of the output of an operator as pre-grouped if necessary). In the example, the query analyzer annotates attributes $a$, $b$, and $c$ as *pre-grouped* in the output of $R \bowtie S$. Based on this information, the query analyzer disables the join key distribution knob on the next equi-join that joins with $T$.

Chapter 17

# Symbolic Query Engine

*And I also trust that there's more than one way to do something.*

*– Dennis Muren, born 1946 –*

In this chapter, we first define the data model of symbolic data and discuss how to physically store the symbolic data. Then we present the algorithms for the operators in symbolic query engine through a running example.

## 17.1 Symbolic Data Model

### 17.1.1 Definitions

**Definition 17.1** *(Symbolic Relation:) A* symbolic relation *consists of a* relation schema *and a* symbolic relation instance*. The definition of a relation schema is exactly the same as the classical definition of a relation schema [*Cod70*]. Let $R(a_1{:}dom(a_1), \ldots, a_i{:}\ dom(a_i), \ldots, a_n{:}\ dom(a_n))$ be a relation schema with $n$ attributes; and for each attribute $a_i$, let $dom(a_i)$ be the domain of attribute $a_i$.*

**Definition 17.2** *(Symbolic Relation Instance:) A* symbolic relation instance *is a set of* symbolic tuples $T$*. Each symbolic tuple $t \in T$ is a $n$-tuple with $n$* symbols*: $\langle s_1, s_2, \ldots, s_n \rangle$. As a shorthand, symbol $s_i$ in tuple $t$ can be referred by $t.a_i$. A symbol $s_i$ is associated with a set of* predicates *$P_{s_i}$ (where $P_{s_i}$ can be empty). The value of symbol $s_i$ represents any one of the values in the domain of attribute $a_i$ that satisfies* all *predicates in $P_{s_i}$. A predicate $p \in P_{s_i}$ of a symbol $s_i$ is a propositional formula that involves at least $s_i$, and zero or more other symbols that appear* *in different symbolic relation instances. Therefore, a symbol $s_i$ with its predicates $P_{s_i}$ can be represented by a conjunction of propositional logic formulas.*

**Definition 17.3** *(Symbolic Database:) A* symbolic database *is defined as a set of symbolic relations and there is a one-to-many mapping between one symbolic database and many traditional relational databases.*

### 17.1.2 Data Storage

Symbolic databases are a generalization of relational databases and provide an abstract representation of concrete data. Given the close relationship between relational databases and symbolic databases, and the maturity of relational database technology, it may not pay off to re-design another physical model for storing symbolic data. QAGen opts to leverage existing relational databases to implement the symbolic database concept. To that end, a natural idea for storing symbolic data is to store the data in columns of tables, introduce a user-defined type (UDT) to describe the columns, and use SQL user-defined functions to implement the symbolic operations. However, symbolic operations (e.g., a join that controls the output size and distribution) are too complex to be implemented by SQL user-defined functions. As a result, we propose to store symbols (and associated predicates) in relational databases by simply using the `varchar` SQL data type and let the QAGen symbolic query engine operate on a relational database directly. For that reason, we integrate the power of various access methods brought by the relational database engine into symbolic query processing.

The next interesting question is how to normalize a symbolic relation for efficient symbolic query processing. From the definition of a symbol, we know that a symbol may be associated with a set of predicates. For example, symbol $a_1$ may have a predicate $[a1 \geq p_1]$ associated with it. As we will see later, most of the symbolic operations impose some predicates (from now on, we use the term predicate instead of constraint) on the symbols. Therefore, a symbol may be associated with many predicates. As a result, QAGen stores the predicates of a symbol in a separate relational table called $PTable$. Reusing Figure 15.1 again, symbolic relation $R$ can be represented by a normal table in a RDBMS named $R$ with the schema: R(a: *varchar*, b: *varchar*) and a table named $PTable$ with the schema: PTable(symbol: *varchar*, predicate: *varchar*). After a simple selection $\sigma_{a \geq p_1} R$ on table $R$, the relational representation of symbolic table $R$ is:

| a | b |
|---|---|
| $a1$ | $b1$ |
| $a2$ | $b2$ |

(i) Table $R$ (2 tuples)

| symbol | predicate |
|---|---|
| $a1$ | $[a1 \geq p_1]$ |
| $a2$ | $[a2 < p_1]$ |

(ii) PTable (2 tuples

## 17.2 Symbolic Operations

The major difference between symbolic query execution and traditional query processing is that the input (and thus the output) of each operator is symbolic data but not concrete data. The flexibility of symbolic data allows an operator to control its internal operation and thus its output. As in traditional processing, an operator is implemented as an iterator. Therefore the interface of an operator is the same as in traditional query processing which consists of three methods: *open()*, *getNext()* and *close()*.

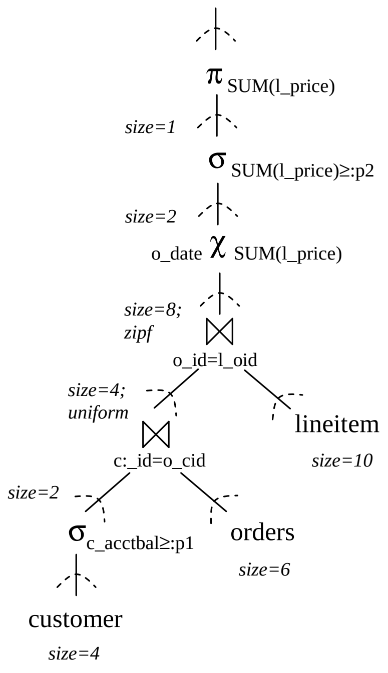
Next, we present the knobs and the algorithms for each operator through a running example. Unless stated otherwise, the following subsections only show the details of the *getNext()* method of each operator. All other aspects (e.g., *open()* and *close()*) are straightforward so that they may be omitted for brevity. The running example is a 2-way join query which can demonstrate the details of the symbolic execution of selection, equi-join, aggregation and projection. We also discuss some special cases of these operators. Figure 17.1 (a) shows the input query tree (with all knobs and their values given). The example is based on the following simplified TPC-H schema:

```
CREATE TABLE customer (
 c_id INTEGER PRIMARY KEY, c_acctbal FLOAT
)

CREATE TABLE orders (
 o_id INTEGER PRIMARY KEY, o_date DATE,
 o_cid INTEGER REFERENCES Customer.c_id
)

CREATE TABLE lineitem (
 l_id INTEGER PRIMARY KEY, l_price FLOAT,
 l_oid INTEGER REFERENCES Orders.o_id
)
```

### 17.2.1 Table Operator

| **Knob:** | Table Size (compulsory) |
|---|---|

In QAGen, a base table in a query tree is regarded as an operator. During the open() method, it creates a relational table in a RDBMS with the attributes specified on input schema $S$. According to the designed storage model, all attributes are in the SQL data type `varchar`. Next, it fills up the table by creating new symbolic tuples until it reaches the defined table size. Each symbol in



**(a) Input Query Tree**

| c_id | c_acctbal |
|---|---|
| $c\_id1$ | $c\_acctbal1$ |
| $c\_id2$ | $c\_acctbal2$ |
| $c\_id3$ | $c\_acctbal3$ |
| $c\_id4$ | $c\_acctbal4$ |

(i) Customer (4 tuples)

| o_id | o_date | o_cid |
|---|---|---|
| $o\_id1$ | $o\_date1$ | $o\_cid1$ |
| $o\_id2$ | $o\_date2$ | $o\_cid2$ |
| $o\_id3$ | $o\_date3$ | $o\_cid3$ |
| ... | ... | ... |
| $o\_id6$ | $o\_date6$ | $o\_cid6$ |

(ii) Orders (6 tuples)

| l_id | l_price | l_oid |
|---|---|---|
| $l\_id1$ | $l\_price1$ | $l\_oid1$ |
| $l\_id2$ | $l\_price2$ | $l\_oid2$ |
| $l\_id3$ | $l\_price3$ | $l\_oid3$ |
| ... | ... | ... |
| ... | ... | ... |
| $l\_id10$ | $l\_price10$ | $l\_oid10$ |

(iii) Lineitem (10 tuples)

| symbol | predicate |
|---|---|
| | |

(iv) PTable

**(b) Initial Symbolic Database**

| c_id | c_acctbal |
|---|---|
| $c\_id1$ | $c\_acctbal1$ |
| $c\_id2$ | $c\_acctbal2$ |
| $c\_id3$ | $c\_acctbal3$ |
| $c\_id4$ | $c\_acctbal4$ |

(i) Customer (4 tuples)

| o_id | o_date | o_cid |
|---|---|---|
| $o\_id1$ | $o\_date1$ | $c\_id1$ |
| $o\_id2$ | $o\_date2$ | $c\_id1$ |
| $o\_id3$ | $o\_date1$ | $c\_id2$ |
| $o\_id4$ | $o\_date2$ | $c\_id2$ |
| $o\_id5$ | $o\_date5$ | $c\_id3$ |
| $o\_id6$ | $o\_date6$ | $c\_id3$ |

(ii) Orders (6 tuples)

| l_id | l_price | l_oid |
|---|---|---|
| $l\_id1$ | $l\_price1$ | $o\_id1$ |
| $l\_id2$ | $l\_price1$ | $o\_id1$ |
| $l\_id3$ | $l\_price1$ | $o\_id1$ |
| $l\_id4$ | $l\_price1$ | $o\_id1$ |
| $l\_id5$ | $l\_price5$ | $o\_id2$ |
| $l\_id6$ | $l\_price5$ | $o\_id2$ |
| $l\_id7$ | $l\_price1$ | $o\_id3$ |
| $l\_id8$ | $l\_price5$ | $o\_id4$ |
| $l\_id9$ | $l\_price9$ | $o\_id5$ |
| $l\_id10$ | $l\_price10$ | $o\_id6$ |

(iii) Lineitem (10 tuples)

| symbol | predicate |
|---|---|
| $c\_acctbal1$ | $[c\_acctbal1 \geq p_1]$ |
| $c\_acctbal2$ | $[c\_acctbal2 \geq p_1]$ |
| $c\_acctbal3$ | $[c\_acctbal3 < p_1]$ |
| $c\_acctbal4$ | $[c\_acctbal4 < p_1]$ |
| $l\_price1$ | $[aggsum1 = 5 \times l\_price1]$ |
| $l\_price5$ | $[aggsum2 = 3 \times l\_price5]$ |
| $aggsum1$ | $[aggsum1 \geq p2]$ |
| $aggsum2$ | $[aggsum2 < p2]$ |

(iv) PTable

**(c) Final Symbolic Database**

Figure 17.1: Running Example

the newly created tuples is named using the attribute name as prefix and a unique identification number. Therefore, at the beginning of symbolic query processing, each symbol in the base table should be unique. Figure 17.1 (b) shows the relational representation of the three symbolic relations $customer$, $orders$ and $lineitem$ for the running example. The *getNext()* method of the table operator is the same as the traditional Table-Scan operator that returns a tuple to its parent or returns null (an end-of-result message) if all tuples have been returned. Note that if the same table is used multiple times in the query, then the table operator only creates and fills the base symbolic table once.

*Primary-keys*, *unique* and *not null* constraints are already enforced because all symbols are initially unique. *Foreign-key* constraints related to the query are taken care of by the join operator directly.

### 17.2.2 Selection Operator

| | |
|---|---|
| **Knob:** | Output Cardinality $c$ (optional; default value = input size) |

Let $I$ be the input and $O$ be the output of the selection operator $\sigma$ and let $p$ be the selection predicate. The symbolic execution of the selection operator controls the cardinality $c$ of the output. Depending on the input characteristics, the difficulty of the problem and the solutions are completely different. Generally, there are two different cases.

#### Case 1: Input is not pre-grouped w.r.t. the selection attribute(s)

This is case (a) in Figure 15.3 and the selections in the running example (Figure 17.1a operator (ii) and (vi)) are in this case. This implementation is chosen by the query analyzer when the input is not pre-grouped w.r.t. the selection attribute(s) and it is the usual case for most queries. In this case, the selection operator controls the output as follows:

1. During its getNext() method, read in a tuple $t$ by invoking getNext() on its child operator and process with [Positive Tuple Annotation] if the output cardinality has not reached $c$. Else proceed to [Negative Tuple Post Processing] and then return null to its parent.
2. [Positive Tuple Processing] If the output cardinality has not reached $c$, then (a) for each symbol $s$ in $t$ that participates in the selection predicate $p$, insert a corresponding tuple $\langle s, p\rangle$ to the $PTable$; and (b) return this tuple $t$ to its parent.
3. [Negative Tuple Post Processing] However, if the output cardinality has reached $c$, then fetch all the remaining tuples $I^-$ from input $I$. For each symbol $s$ of tuple $t$ in $I^-$ that participates in the selection predicate $p$, insert a corresponding tuple $\langle s, \neg p\rangle$ to the $PTable$, and repeat this step until calling getNext() on its child has no more tuples (returns null).

| c_id | c_acctbal |
|---|---|
| $c\_id1$ | $c\_acctbal1$ |
| $c\_id2$ | $c\_acctbal2$ |

(i) Output of $\sigma$; 2 tuples

| symbol | predicate |
|---|---|
| $c\_acctbal1$ | $[c\_acctbal1 \geq p_1]$ |
| $c\_acctbal2$ | $[c\_acctbal2 \geq p_1]$ |
| $c\_acctbal3$ | $[c\_acctbal3 < p_1]$ |
| $c\_acctbal4$ | $[c\_acctbal4 < p_1]$ |

(ii) PTable

Figure 17.2: Symbolic Database after selection

Each getNext() call on the selection operator returns a *positive* tuple to its parent that satisfies the selection predicate $p$ until the output cardinality has been reached. Moreover, to ensure that all negative tuples (i.e., tuples obtained from the child operator after the output cardinality has been reached) would not get some instantiated values later in the data instantiation phase that ends up passing the selection predicate, the selection operator associates the negation of predicate $p$ to those negative tuples. In the running example, attribute $c\_acctbal$ in the selection predicate $[c\_acctbal \geq p_1]$ of operator (ii) is not pre-grouped, because the data comes directly from the base $customer$ table. Since the output cardinality $c$ of the selection operator is 2, the selection operator associates the positive predicate $[c\_acctbal \geq p_1]$ to symbols $c\_acctbal1$ and $c\_acctbal2$ of the first two input tuples and associates the negated predicate $[c\_acctbal < p_1]$ to symbols $c\_acctbal3$ and $c\_acctbal4$ of the rest of the input tuples. Figure 17.2 (i) shows the output of the selection operator and Figure 17.2 (ii) shows the content of the $PTable$ after the selection.

#### Case 2: Input is pre-grouped w.r.t. the selection attribute(s)

This is case (b) in Figure 15.3. This implementation is chosen by the query analyzer when the input is pre-grouped with respect to any attribute that appears in the selection predicate $p$. In this case, we can show that the problem of controlling the output cardinality is reducible to the subset-sum problem.

The subset-sum problem [GJ90] takes as input an integer sum $c$ and a set of integers $C = \{c_1, c_2, ..., c_m\}$, and outputs whether there exists a subset $C^+ \subseteq C$ such that $\sum_{c_i \in C+} c_i = c$. Consider Figure 17.3, which is an example of pre-grouped input of a selection. Input $I$ defines one attribute $k$ and has in total $\sum c_i$ rows. The rows in $I$ are clustered in $m$ groups, where the $i$-th group has exactly $c_i$ tuples using the same symbolic value $k_i$ ($i \leq m$). We now search for a subset of those $m$ groups in $I$ such that the output has the size $c$. Assume, we find such a subset, i.e., the symbolic values of those groups which result in the output with size $c$. The groups returned by such a search induce a solution for the original subset-sum problem.

| k | |
|---|---|
| $k_1$ | } *e.g.* $c_1 = 5$ *times* |
| $k_2$ | } *e.g.* $c_2 = 4$ *times* |
| $k_3$ | } *e.g.* $c_3 = 3$ *times* |
| $k_4$ | } *e.g.* $c_4 = 1$ *times* |
| ... | |
| $k_m$ | } $c_m$ *times* |

Input $I$

Figure 17.3: Pre-grouped selection

The subset-sum problem is a weakly $\mathcal{NP}$-complete problem and there exists a pseudopolynomial algorithm which uses dynamic programming to solve it [GJ90]. The complexity of the dynamic programming algorithm is $\mathcal{O}(cm)$, where $c$ is the desired output cardinality and $m$ is the number of different groups in $I$. When $c$ is large, the dynamic programming algorithm runs very slow. Furthermore, it is also possible that there is no subset in the input whose sum exactly meets $c$ as well. As a result, when the query analyzer detects that the input of a selection is pre-grouped, it allows the user to specify the following knob in addition to the output cardinality knob:

| **Knob:** | Approximation ratio $\epsilon$ |
|---|---|

The approximation ratio knob allows the selection to return an approximate number of tuples rather than the exact number of tuples that is specified by the testers, which is acceptable in DBMS feature testing.

There are several approximation schemes in the literature to solve the subset-sum problem (e.g., [IK75], [Prz02], [KMPS03]). However, these approximation schemes are not directly applicable in our case. We illustrate this problem using the test case in Figure 17.4. The test query in the test case is a two-way join query with an aggregation. In Figure 17.4, the tester defines that the output cardinality of the selection is $5$ tuples with an approximation ratio of $0.1$. Assume that the input of the selection in Figure 17.4 has eight tuples but they are pre-grouped into three clusters (a cluster $c_1$ consists of four tuples, and two clusters $c_2$ and $c_3$ consist of two tuples each) with respect to both attributes $attr_1$ and $attr_2$ after the two-way join. In order to pick the right subset of pre-grouped tuples with a total cardinality as $c$ ($c = 5$ in the example), the selection operator needs to solve the subset-sum problem by an approximation scheme. Unfortunately, all existing approximation schemes would return a subset whose sum is *smaller than* (or equal to) the target sum. For example, it is possible that the approximation scheme suggests picking clusters $c_2$ and $c_3$ from the pre-grouped input, such that the selection returns a total of four tuples (which is actually the optimal solution) as output. However, if the selection really returns four

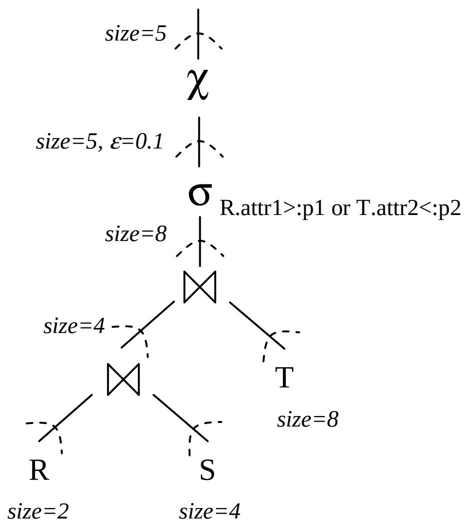


Figure 17.4: A test case with the approximation ratio knob

tuples, then the upper aggregation operator $\chi$ in Figure 17.4 would experience a “lack-of-tuple” error (it expects to have five or more input tuples). Even though the target users of QAGen are experienced testers and we assume there are no contradicting knob values in the input test case, it is often difficult for the testers to specify a semantically correct test case when the system allows tolerances on the operator’s cardinality constraint. This practical problem drove us to develop an approximation scheme that returns a subset with sum *greater than or equal to* the target sum $c$ and has an approximation ratio $\epsilon$. We call this new problem as the *Overweight Subset-Sum Problem* and it requires non-trivial modifications to the current approximation schemes.

Our new approximation scheme is based on the “quantization method” [IK75] and consists of two phases. It takes a list $C$ of sorted numbers as input. Then, it first separates the input list of numbers into two lists: large number list $L$ and small number list $S$. In the first phase, it tries to quickly come up with a set of approximation solutions by only considering the numbers with large values (i.e., only elements in $L$). Then, in the second phase, it tries to fine tune the approximation solutions by the set of small numbers in $S$.

Figure 17.5 shows the pseudocode of the approximation scheme. In the beginning, it trims input list $C$ if it contains more than one number which has a value greater than or equal to the target sum $c$. For example, assume input list $C$ is $[1, 2, 5, 6, 13, 27, 44, 47, 48]$, the target sum $c$ is 30, and the approximation ratio $\epsilon$ is 0.1. After Line (1–2), $C$ becomes $[1, 2, 5, 6, 13, 27, 44]$ because the number 47 and 48 cannot be part of the answer. Then it tries to quantize the large subset-sum values into different buckets (Line 4-7) in order to minimize the number of subsequent operations from Line 11 to Line 24. Essentially, based on the quantization factor $d$, the algorithm quantizes

```
**Algorithm** APPROXIMATE_OVERWEIGHT_SUBSET_SUM($P$)
**Input:** (a) A list of sorted integers $C = [c_1, c_2, ..., c_m]$ where $c_i < c_{i+1}$ (b) Target sum $c$, (c) Approximation
    ratio $\epsilon$
**Output:** A subset of integers $C^+ \subseteq C$ such that $c \leq \sum_{c_i \in C^+} c_i$ with approximation ratio $\epsilon$
1.  **if** $\exists c_i \in C, c_i \geq c$
2.      **then** Trim $C$ by removing elements $c_{i+1}, ..., c_m$
3.  Set the largest possible optimal solution $p$ as $p = c_1 + c_2 + ... + c_r \geq c$ where $c_1 + c_2 + ... + c_{r-1} < c$.
    If $c_r \geq c$, **return** $\{c_r\}$. If no such $r$ exists, **return** “no solution exists".
4.  Set quantization factor $d = (\epsilon/2)^2 p$
5.  Set number of buckets $g = \lceil p/d \rceil + \min\{r, \lceil 2/\epsilon \rceil\}$
6.  Initialize $g + 1$ approximate answer buckets $\mathcal{B} = \{B_0, B_1, ..., B_g\}$
7.  Initialize a subset-sum array $X$ of size $g + 1$ where $X[i]$ stores the subset-sum of the elements in $B_i$.
    Set $X[0] = 0$ and $X[i] = -1$ ($1 \leq i \leq g$)
8.  Set list $S = [c_1, c_2, ..., c_u]$ where $c_u < (\epsilon/2)p$
9.  Set list $L = [c_{u+1}, c_{u+2}, ..., c_m]$ where $c_{u+1} \geq (\epsilon/2)p$
10. Return $S$ as the answer if $L$ is empty
11. **for** each number $c_i \in L$
12.         Set the quantized value of $v_i$ of $c_i$ as $\lceil c_i/d \rceil$
13.         **for** each $j = g - v_i$ down-to 0
14.                 **if** $X[j] \neq -1$
15.                     **then if** $X[j + v_i] < X[j] + c_i$
16.                             **then** set $B_{j+v_i} = B_j \cup \{c_i\}$,
17.                                  set $X[j + v_i] = X[j] + c_i$
18. **for** each bucket $B_i \in \mathcal{B}$ with $X[i] \neq -1$
19.         set $j = 0$
20.         **while** $X[i] < c$
21.                 set $B_i = B_i \cup \{c_j\}$, where $c_j$ is the $j$-number in list $S$,
22.                 set $X[i] = X[i] + c_j$
23.                 $j = j + 1$
24. **return** $B_i$, where $X[i] = min(X[j])$ for all $0 \leq j \leq g$ and $X[j] \geq c$;
```

Figure 17.5: Approximation scheme for the Overweight Subset-Sum Problem

the input list of numbers into $g$ buckets. The quantization factor $d$ is carefully chosen such that it is large enough to give a manageable number of buckets and at the same time respecting the error bound given by the approximation ratio $\epsilon$ [IK75]. The quantization factor $d$ is computed based on the approximation ratio $\epsilon$ and one of the possible subset-sums $p$. Such a $p$ value is found (Line 3) by adding $c_1, c_2, \ldots$ until the sum is at least the target sum $c$; if no such value is found, the sum of all values in $C$ must be less than $c$, and we can conclude that there is no solution for the overweight subset-sum problem. An interesting special case is that, if the last value of the sum, $c_r$, is at least $c$, we immediately know $\{c_r\}$ is the desired *optimal* solution to the overweight subset-sum problem. $X$ is a subset-sum array. Entry $X[i]$ stores the subset-sum of the elements in bucket $B_i$ (Line 7). Initially, $X[0]$ is set to 0 as a boundary condition and $X[i]$ (where $i \neq 0$) is set to -1 to make sure a subset-sum cannot exceed $i \times d$ in any case.

In the example, $p = 1+2+5+6+13+27 = 54$, and thus the quantization level $d$ and the number of buckets $g$ are 0.135 and 406, respectively. Afterwards, the algorithm creates $g+1$ approximate answer buckets $\mathcal{B}$ and a subset-sum array $X$, where each approximate answer bucket $B_i$ will hold a set of numbers whose sum is close to a factor $i$ of the quantization factor $d$ (i.e., the subset-sum is close to $i \times d$) and $X[i]$ represents the total sum of numbers in $B_i$.

As mentioned, the input list of numbers is separated into two lists $S$ and $L$ according to the numbers' value (Lines 8–9). In the example, the small list $S$ consists of the first two numbers 1 and 2 in the input list $C$ and the large list $L$ consists of all the rest of the numbers $[5, 6, 13, 27, 44]$. Then, the first phase (Lines 11–17) begins by examining each number in the large number list $L$ and tries to assign the number into different buckets. For example, the first number in $L$ is 5 and its quantized values is $\lceil 5/0.135 \rceil = 38$. Therefore, the algorithm sets $B_{38} = \{5\}$ and the corresponding subset-sum array entry $X[38]$ has a value of 5. Similarly, for the second number 6 in $L$, its quantized value is $\lceil 6/0.135 \rceil = 44$. As a result, the algorithm sets $B_{44}$ to be $\{6\}$, updates $X[44]$ to be 6, sets $B_{82}$ to be $\{5, 6\}$ and updates $X[82]$ to have a value of 11 ($= 5+6$). If a bucket is non-empty, the algorithm only updates the bucket (and its corresponding subset-sum in $X$) if the updated subset-sum is larger than the current subset-sum of that bucket (Lines 15–17).

In the second phase (Lines 18–23), the algorithm tries to fine tune each approximate answer bucket $B$ by adding the numbers in the small list $S$, one-by-one, until it exceeds the target sum $c$. Afterwards, the algorithm scans array $X$ and identifies the subset which has the smallest subset-sum that is greater than the target sum $c$. Finally, it returns the corresponding subset in $B$ as the final result.

The complexity of our proposed approximation scheme is $\mathcal{O}(m/\epsilon^2)$. We put the correctness proof and the complexity analysis in Appendix A. We now reuse Figure 17.3 to illustrate the overall algorithm of the selection operator. Assume that the input has 13 tuples which are clustered into 4 groups with symbol $k_1$, $k_2$, $k_3$, and $k_4$ respectively. Furthermore, assume that the output cardinality and the approximation ratio is defined as 7 tuples and 0.2 respectively. The pre-grouped input selection controls the output as follows:

1. [Subset-sum solving] During its open() method, (a) materialize input $I$ of the selection operator; (b) extract the pre-group size (e.g., $c_1 = 5, c_2 = 4, c_3 = 3, c_4 = 1$) of each symbol $k_i$ by executing "*Select Count(k) From I Group By k Order By Count(k)*" on the materialized input; (c) invoke the approximation scheme in Figure 17.5 with the pre-group sizes (the set of numbers), the output cardinality (the target sum), and the error tolerance $\epsilon$ as input. The output of this step is a subset of symbols $K^+$ in $I$ such that the output cardinality (approximately) matches the requirement (e.g., $K^+ = \{k1, k3\}$ because $c_1 + c_3 = 5 + 3 = 8 \geq c$). If no such a subset exist, then stop processing and report this error to the user.

2. [Positive Tuple Processing] During getNext(), (a) for each symbol $k_i$ in $K^+$, read all tuples $I^+$ from the materialized input of $I$ which have $k_i$ as the value of attribute $k$; (b) for each symbol $s$ that participates in the selection predicate $p$ in tuple $t$ of $I^+$, insert a corresponding tuple $\langle s, p \rangle$ to the $PTable$; (c) return tuple $t$ to the parent.

3. [Negative Tuple Post Processing] This step is the same as the Negative Tuple Post Processing step in the simple case (Section 17.2.2 case 1) that annotates negative predicates to each negative tuple.

Note that, in this case, the selection is a blocking operation because it needs to read all the tuples from input $I$ first in order to solve the subset-sum problem. One optimization for this case is that if $c$ is equal to the input size of $I$, then all input tuples must be returned to its parent and thus the subset-sum solving function can be skipped even though the input data is pre-grouped.

### 17.2.3 Equi-Join Operator

| | |
|---|---|
| **Knob:** | Output Cardinality $c$ (optional; default value = size of the non-distinct input) |

Let $R$ and $S$ be the inputs, $O$ be the output, and $p$ be the simple equality predicate $j = k$ where $j$ is the (non-pregrouped) join attribute on $R$, and $k$ is the join attribute on $S$ that refers to $j$ by a foreign-key relationship. The symbolic execution of the equi-join operator ensures that the join result size is $c$. Again, depending on whether the input is pre-grouped or not, the solutions are different.

#### Case 1: Input is not pre-grouped w.r.t. join attribute $k$.

This is case (c) in Figure 15.3, where join attribute $k$ in input $S$ is not pre-grouped. In this case, it is possible to support one more knob on the equi-join operation:

| | |
|---|---|
| **Knob:** | Join Key Distribution $b$ (optional; choices = [Uniform or Zipf]; default = Uniform) |

The join key distribution $b$ defines how many tuples of input $S$ join with each individual tuple in input $R$. For example, if the join key distribution is uniform, then each tuple in $R$ joins with roughly the same number of tuples in $S$. Both join operators in Figure 17.1 (a) fall into this case. In this case, the equi-join operator (which supports both output cardinality $c$ and distribution $b$) controls the output as follows:

1. [Distribution instantiating] During its open() method, instantiate a distribution generator $D$, with the size of $R$ as domain (denoted by $n$), the output cardinality $c$ as frequency, and the distribution type $b$ as input. This distribution generator $D$ can be the one that has been proposed earlier (e.g., [GSE$^+$94], [CN97]) or any statistical packages that generate $n$ numbers $m_1, m_2, \ldots, m_n$ following Uniform or Zipf [Zip49] distribution with a total frequency of $c$. The distribution generator $D$ is an iterator with a getNext() method. For the $i$-th call on the getNext() method ($0 \leq i \leq n$), it returns the expected frequency $m_i$ of the $i$-th number under distribution $b$.

2. During its getNext() call, if the output cardinality has not yet reached $c$, then (a) check if $m_i = 0$ or if $m_i$ has not yet initialized, and, if so, initialize $m_i$ by calling getNext() on $D$ and get a tuple $r^+$ from $R$ ($m_i$ is the total number of tuples from $S$ that should join with $r^+$); (b) get a tuple $s^+$ from $S$ and decrease $m_i$ by one; (c) join tuple $r^+$ with $s^+$ according to [Positive Tuple Joining] below; (d) return the joined tuple to the parent. However, during the getNext() call, if the output cardinality has reached $c$ already, then process [Negative Tuple Joining] below, and return null to its parent.

3. [Positive Tuple Joining] If the output cardinality has not reached $c$, then (a) for tuple $s^+$, replace symbol $s^+.k$, which is the symbol of the join key attribute $k$ of tuple $s^+$, by symbol $r^+.j$, which is the symbol of the join key attribute $j$ of tuple $r^+$. After this, tuple $r^+$ and tuple $s^+$ should share exactly the same symbol on their join attributes. Note that the replacement of symbols in this step is done on both tuples loaded in the memory and the related tuples in base table as well (using an SQL statement like "*Update $k.BaseTable$ Set k=$r^+.j$ WHERE k=$s^+.k$*" to update the symbols on the base table where join attribute $k$ comes from); (b) perform an equi-join on tuple $r^+$ and $s^+$.

4. [Negative Tuple Joining] However, if the output cardinality has reached $c$, then fetch all the remaining tuples $S^-$ from input $S$. For each tuple $s^-$ in $S^-$, randomly look up a symbol $j^-$ on the join key $j$ in the set minus between the base table where join attribute $j$ originates from and $R$ (using an SQL statement with the `MINUS` keyword), replace $s^-.k$ with symbol $j^-$. This replacement is done on the base tables only because these tuples are not returned to the parent.

In the running example (Figure 17.1), after the selection on table $customer$ (operator ii), the next operator is a join between the selection output (Table (i) in Figure 17.2) and table $orders$ (Table (ii) in Figure 17.1 (b)). The output cardinality $c$ of that join (operator iii) is 4 and the join key distribution is uniform. Since the input of the join on the join key $o\_cid$ is not pre-grouped, the query analyzer uses the algorithm above to perform the equi-join. First, the distribution generator $D$ generates 2 numbers (which is the size of input $R$), with total frequency of 4 (output cardinality), and uniform distribution. Assume $D$ returns the sequence {2, 2}. This means that the first customer $c\_id1$ should take 2 orders ($o\_id1$ and $o\_id2$) and the second customer $c\_id2$ should also

| c_acctbal | o_id | o_date | **c_id=o_cid** |
|---|---|---|---|
| $c\_acctbal1$ | $o\_id1$ | $o\_date1$ | **c_id1** |
| $c\_acctbal1$ | $o\_id2$ | $o\_date2$ | **c_id1** |
| $c\_acctbal2$ | $o\_id3$ | $o\_date3$ | **c_id2** |
| $c\_acctbal2$ | $o\_id4$ | $o\_date4$ | **c_id2** |

(i) Output of ($\sigma(Customer) \bowtie Order$); 4 tuples

| o_id | o_date | **o_cid** |
|---|---|---|
| $o\_id1$ | $o\_date1$ | **c_id1** |
| $o\_id2$ | $o\_date2$ | **c_id1** |
| $o\_id3$ | $o\_date3$ | **c_id2** |
| $o\_id4$ | $o\_date4$ | **c_id2** |
| $o\_id5$ | $o\_date5$ | ***c_id3*** |
| $o\_id6$ | $o\_date6$ | ***c_id4*** |

(ii) Orders (4 pos, 2 neg)

Figure 17.6: Symbolic Database after join

| c_id | c_acctbal | o_date | o_cid | l_id | l_price | **o_id = l_oid** |
|---|---|---|---|---|---|---|
| $c\_id1$ | $c\_acctbal1$ | $o\_date1$ | $o\_cid1$ | $l\_id1$ | $l\_price1$ | **o_id1** |
| $c\_id1$ | $c\_acctbal1$ | $o\_date1$ | $o\_cid1$ | $l\_id2$ | $l\_price2$ | **o_id1** |
| $c\_id1$ | $c\_acctbal1$ | $o\_date1$ | $o\_cid1$ | $l\_id3$ | $l\_price3$ | **o_id1** |
| $c\_id1$ | $c\_acctbal1$ | $o\_date1$ | $o\_cid1$ | $l\_id4$ | $l\_price4$ | **o_id1** |
| $c\_id1$ | $c\_acctbal1$ | $o\_date2$ | $o\_cid1$ | $l\_id5$ | $l\_price5$ | **o_id2** |
| $c\_id1$ | $c\_acctbal1$ | $o\_date2$ | $o\_cid1$ | $l\_id6$ | $l\_price6$ | **o_id2** |
| $c\_id2$ | $c\_acctbal2$ | $o\_date3$ | $o\_cid2$ | $l\_id7$ | $l\_price7$ | **o_id3** |
| $c\_id2$ | $c\_acctbal2$ | $o\_date4$ | $o\_cid2$ | $l\_id8$ | $l\_price8$ | **o_id4** |

i) Output of ($\sigma(Customer) \bowtie Order) \bowtie Lineitem$. 8 tuples

| l_id | l_price | **l_oid** |
|---|---|---|
| $l\_id1$ | $l\_price1$ | **o_id1** |
| $l\_id2$ | $l\_price2$ | **o_id1** |
| $l\_id3$ | $l\_price3$ | **o_id1** |
| $l\_id4$ | $l\_price4$ | **o_id1** |
| $l\_id5$ | $l\_price5$ | **o_id2** |
| $l\_id6$ | $l\_price6$ | **o_id2** |
| $l\_id7$ | $l\_price7$ | **o_id3** |
| $l\_id8$ | $l\_price8$ | **o_id4** |
| $l\_id9$ | $l\_price9$ | ***o_id5*** |
| $l\_id10$ | $l\_price10$ | ***o_id6*** |

(ii) Lineitem (8 pos, 2 neg)

Figure 17.7: Symbolic Database after 2-way join

take 2 orders ($o\_id3$ and $o\_id4$). As a result, symbols $o\_cid1$ and $o\_cid2$ from the Orders table should be replaced by $c\_id1$ and symbols $o\_cid3$ and $o\_cid4$ from the Orders table should be replaced by $c\_id2$ (Step 3 above). In order to fulfill the foreign-key constraint on those tuples which do not join, Step 4 above (Negative Tuple Joining) replaces $o\_cid5$ and $o\_cid6$ by customers that did not pass through the selection filter (i.e., customer $c\_id3$ and $c\_id4$) randomly. Figure 17.6 (i) below shows the output of the join and Figure 17.6 (ii) shows the updated $orders$ table (updated join keys are **bold**).

After the join operation above, the next operator in the running example is another join between the above join results (Figure 17.6(i)) and the base $lineitem$ table (Figure 17.1b(iii)). Again, the input of the join on the join key $l\_oid$ of the $lineitem$ table is not pre-grouped and thus the above equi-join algorithm is chosen by the query analyzer. Assume that the distribution generator generates a Zipf sequence {4,2,1,1} for the four tuples in Figure 17.6 (i) to join with 8 out of 10 line items (where 8 is the user-specified output cardinality of this join operation). Therefore it produces the output in Figure 17.7 (i) (updated join keys are **bold**):

Finally, note that if the two inputs of an equi-join are base tables (with foreign-key constraint), then the output cardinality knob is disabled by the query analyzer. This is because in that case, all tuples from input $S$ must join with a tuple from input $R$ and thus the output cardinality must be

| j |
|---|
| $j1$ |
| $j2$ |
| $j3$ |
| ... |
| ... |
| $jl$ |

(i)Table $R$

| k | |
|---|---|
| $k1$ | } *e.g.* $c_1 = 5$ *times* |
| $k2$ | } *e.g.* $c_2 = 4$ *times* |
| $k3$ | } *e.g.* $c_3 = 3$ *times* |
| $k4$ | } *e.g.* $c_4 = 1$ *times* |
| ... | |
| $km$ | } $c_m$ *times* |

(ii) Table $S$

Figure 17.8: Pre-grouped equi-join

same as the size of $S$.

**Case 2: Input is pre-grouped w.r.t. join attribute $k$.**

This is case (d) in Figure 15.3 and this implementation is chosen by the query analyzer when input $S$ is pre-grouped w.r.t. join attribute $k$. This sometimes happens when a preceding join introduces a distribution on $k$ as in the example in Figure 15.1. In the following we show that if the input is pre-grouped w.r.t. join attribute $k$ of an equi-join, then the problem of controlling the output cardinality (even without the join key distribution) is also reducible to the subset-sum problem.

Consider tables $R$ and $S$ in Figure 17.8, which are the inputs of such a join. Table $R$ has one attribute $j$ with $l$ tuples all using distinct symbolic values $ji$ ($i \leq l$). Table $S$ also defines only one attribute $k$ and has in total $\sum c_i$ rows. The rows in $S$ are clustered into $m$ groups, where the $i$-th group has exactly $c_i$ tuples using the same symbolic value $ki$ ($i \leq m$). We now search for a subset of those $m$ groups in $S$ that join with arbitrary tuples in $R$ so that the output has size $c$. Assume that we find such a subset, i.e., the symbolic values of those groups which result in the output with size $c$. The groups returned by such a search induce a solution for the original subset-sum problem.

For testing the feature of a DBMS, again, it is sufficient for the equi-join to return an approximate number of tuples that is close to the user specified cardinality. As a result, when the query analyzer detects that one of the equi-join inputs is pre-grouped, then it allows the user to specify the following knob in addition to the output cardinality knob:

**Knob:** Approximation Ratio $\epsilon$

Again, this is a blocking operator because it needs to read all the input tuples from $S$ first (to solve the subset-sum problem). Similar to the optimization in the selection operator, if $c$ is equal to the input size of $S$, then all tuples of $S$ must be joined with $R$ and the subset-sum solving function can be skipped even though the data is pre-grouped.

We reuse Figure 17.8 to illustrate the algorithm. Assume the join is on Table $R$ and Table $S$ and the join predicate is $j = k$. Assume Table $R$ has three tuples ($\langle j1\rangle, \langle j2\rangle, \langle j3\rangle$), and Table $S$ has 12 tuples which are clustered into 4 groups with symbols $k1$, $k2$, $k3$, $k4$ respectively. Furthermore, assume the join on $R$ and $S$ is specified with an output cardinality as $c = 7$. The pre-grouped input equi-join controls the output as follows:

1. [Subset-sum solving] During its open() method, (a) materialize input $S$ of the join operator; (b) extract the pre-group size (e.g. $c_1 = 5$, $c_2 = 4$, $c_3 = 2$, $c_4 = 1$) of each symbol $ki$ by executing `Select Count(k) From S Group By k Order By Count(k) Desc` on the materialized input; (c) invoke the approximation scheme in Figure 17.5 with the pre-group sizes (the set of numbers), the output cardinality (the target sum), and the approximation ratio $\epsilon$ as input. The output of this step is a subset of symbols $K^+$ in $I$ such that the output cardinality (approximately) matches the requirement (e.g., $K^+ = \{k1, k3\}$ because $c_1 + c_3 = 5 + 3 = 8 \geq c$). If no such subset exists, then stop processing and report this error to the user.

2. [Positive Tuple Joining] During getNext(), (a) for each symbol $ki$ in $K^+$, read all tuples $S^+$ from the materialized input of $S$ which have $ki$ as the value of attribute $k$; (b) afterwards, call getNext() on $R$ once and get a tuple $r$, join all tuples in $S^+$ with $r$ by replacing the join key symbols in $S^+$ with the join key symbols in $r$. For example, the first five $k1$ symbols in $S$ are replaced with $j1$ and the three $k3$ symbols in $S$ are replaced with $j2$ (again, these replacements are done on symbols loaded in the memory and the changes are propagated to the base tables where $j$ and $k$ originate from); (c) return the joined tuples to the parent.

3. [Negative Tuple Joining] This step is the same as the Negative Tuple Joining step in the simple case (Section 17.2.3 case 1) that joins the negative tuples in input $R$ with the negative tuples in input $S$.

### 17.2.4 Aggregation Operator

| **Knob:** | Output Cardinality $c$ (optional; default value = input size) |
|---|---|

Let $I$ be the input and $O$ be the output of the aggregation operator and $f$ be the aggregation function. The symbolic execution of the aggregation operator controls the size of the output $c$.

#### Simple Aggregation

This is the simplest case of aggregation where there is no grouping operation (i.e,. no GROUP-BY keyword) defined on the query. In this case, the query analyzer disables the output cardinality knob because the output cardinality is either 1 (not-empty input) or 0 (empty input). In SQL, there are five aggregation functions: SUM, MIN, MAX, AVG, COUNT. For simple aggregation, the solutions are very similar for both pre-grouped or non-pre-grouped input on the attribute(s) in $f$. The following shows the case of non-pre-grouped input:

Let $expr$ be the expression in the aggregation function $f$ which consists of at least a non-empty set of symbols $S$ in $expr$ and let the size of input $I$ be $n$.

1. SUM($expr$). During its getNext() method, (a) the aggregation operator consumes all $n$ tuples from $I$; (b) for each symbol $s$ in $S$, adds a tuple $\langle s, [aggsum = expr_1+expr_2+\ldots+expr_n]\rangle$ to the $PTable$, where $expr_i$ is the corresponding expression on the $i$-th input tuple; and (c) returns symbolic tuple $\langle aggsum\rangle$ as output. As an example, assume there is an aggregation function SUM(l_price) on top of the join result in Figure 17.7 (i) of the previous section. Then, this operator returns one tuple $\langle aggsum\rangle$ to its parent and adds 8 tuples (e.g., the 2nd inserted tuple is $\langle l\_price2, [aggsum = l\_price1 + l\_price2 + \ldots + l\_price8]\rangle$) to the $PTable$.

   In fact, the above is a base case. If there are no additional constraints that will be further imposed on the predicate symbols, the aggregation operator will optimize the number and the size of the above predicates by inserting only one tuple $\langle l\_price1, [aggsum = l\_price1 \times 8]\rangle$ to the $PTable$ and replacing symbols $l\_price2$, $\ldots$, $l\_price8$ by symbol $l\_price1$ on the base table. One reason for doing that is the size of the input may be very big, if that is the case, the extremely long predicate may exceed the SQL `varchar` size upper bound. Another reason is to insert fewer tuples in the $PTable$. However, the most important reason is that the cost of a constraint solver call is exponential to the size of the input formula in the worst case. Therefore, this optimization reduces the time of the later data instantiation phase. However, there is a trade-off: for each input tuple, the operator has to update the corresponding symbol in the base table where this symbol originates from.

2. MIN($expr$). The MIN aggregation operator also uses similar predicate optimization as SUM aggregation if possible. During its getNext() method, (a) it regards the first expression $expr_1$ as the minimum value and returns $\langle expr_1\rangle$ as output; and (b) replaces the expression $expr_i$ in the remaining tuples (where $2 < i \leq n$) by the second expression $expr_2$ and inserts two tuples $\langle expr_1, [expr_1 < expr_2]\rangle$ and $\langle expr_2$, $[expr_1 < expr_2]\ \rangle$ to the $PTable$. Note that the above optimization must be aware of whether the input is pre-grouped or not. If it is, not only the first but all tuples with $expr_1$ are kept and the remaining are replaced with symbol $expr_2$.

   As an example, assume that there is an aggregation function MIN(l_price) on top of the join result in Figure 17.7(i). Then, this operator returns $\langle l\_price1\rangle$ as output and inserts 2 tuples to the $PTable$: $\langle l\_price1$, $[l\_price1 < l\_price2]\rangle$ and $\langle l\_price2$, $[l\_price1 < l\_price2]\rangle$ to the $PTable$. Moreover, $l\_price3$, $l\_price4$, $\ldots$, $l\_price8$ are replaced by $l\_price2$ on the base table.

3. MAX($expr$). During its getNext() method, (a) it regards the first expression $expr_1$ as the maximum value and returns $\langle expr_1\rangle$ as output; and (b) replaces the expression $expr_i$ in the remaining tuples (where $2 < i \leq n$) by the second expression $expr_2$ and inserts two tuples $\langle expr_1, [expr_1 > expr_2]\rangle$ and $\langle expr_2$, $[expr_1 > expr_2]\ \rangle$ to the $PTable$.

4. COUNT($expr$). The aggregation operator handles the COUNT aggregation function similar to traditional query processing. During its getNext() method, (a) it counts the number of input tuples, $n$; (b) add a tuple $\langle aggcount, aggcount = n\rangle$ to the $PTable$; and (c) returns a symbolic tuple $\langle aggcount\rangle$ as output.

5. AVG($expr$). It is the similar to the case of the SUM aggregation. During its getNext() method, (a) the aggregation operator consumes all $n$ tuples from $I$; (b) for each symbol $s$ in $S$, it adds a tuple $\langle s, [aggavg = (expr_1 + expr_2 + \ldots + expr_n)/n]\rangle$ to the $PTable$, where $expr_i$ is the corresponding expression on the $i$-th input tuple; and (c) returns symbolic tuple $\langle aggavg\rangle$ as output. The optimization can be illustrated by our example: It adds only one tuple $\langle l\_price1, [aggavg = l\_price1]\rangle$ to the $PTable$ and replaces symbols $l\_price2$, . . . , $l\_price8$ by symbol $l\_price1$ on the base table.

In general, combinations of different aggregation functions in one operator (e.g. MIN($expr1$) + MAX($expr2$)) need different but similar solutions. Their solutions are straightforward and we do not cover them here.

### Single GROUP-BY Attribute

When the aggregation operator has one group-by attribute, the output cardinality $c$ defines how to assign the input tuples into $c$ output groups. Let $g$ be the single grouping attribute. For all algorithms we assume that $g$ has no unique constraint in the database schema. Otherwise, the grouping is predefined by the input already and the query analyzer disables all knobs on the aggregation operator for the user. Again, this symbolic operation of aggregation can be divided into two cases:

**Case 1: Input is not pre-grouped w.r.t. the grouping attribute** In addition to the cardinality knob, when the symbols of the grouping attribute $g$ in the input are not pre-grouped, it is possible to support one more knob:

**Knob:** Group Distribution $b$ (optional; choices = [Uniform or Zipf]; default = Uniform)

The group distribution $b$ defines how to distribute the input tuples into the $c$ predefined output groups. In this case, the aggregation operator controls the output as follows:

1. [Distribution instantiating] During its open() method, instantiate a distribution generator $D$, with the size of $I$ (denoted by $n$) as frequency, the output cardinality $c$ as domain, and the distribution type $b$ as input. The distribution generator is the same one as the one for doing equi-join (Section 17.2.3). It generates $c$ numbers $m_1, m_2, \ldots, m_c$, and the $i$-th call on its getNext() method $(0 \leq i \leq c)$ returns the expected frequency $m_i$ of the $i$-th number under distribution $b$.

2. During getNext(), call $D.getNext()$ to get a frequency $m_i$, fetch $m_i$ tuples (let them be $I_i$) from $I$ and execute the following steps. If there are no more tuples from its child operator, return null to the parent.

3. [Group assigning] For each tuple $t$ in $I_i$, except the first tuple $t'$ in $I_i$, replace symbol $t.g$, which is the symbol of the grouping attribute $g$ of tuple $t$, by symbol $t'.g$. $t'.g$ is the symbol of the grouping attribute $g$ of the first tuple $t'$ in the $i$-th group. Note that, the replacement of symbols in this step is done on both tuple loaded in the memory and the related tuples in the base table as well.

4. [Aggregating] Invoke the Simple Aggregation Operator in the previous section (Section 17.2.4) with all the symbols participated in the aggregation function in $I_i$ as input.

5. [Result Returning] Construct a new symbolic tuple $\langle t'.g, agg_i\rangle$ to its parent where $agg_i$ is the symbolic tuple returned by the Simple Aggregation Operator for the $i$-th group. Return the constructed tuple to its parent.

Sometimes, during the open() method, the distribution generator $D$ may return 0 when the distribution is very skew (e.g., Zipf distribution with high skew factor). In this case, it may happen that an output group does not get any input tuple and the final number of output groups may less than the output cardinality requirement. There are several ways to handle this case. One way is to regard this as an runtime error which let the user know that she should not specify such a highly skewed distribution when she asks for many output groups. Another way is to adjust the distribution generator $D$ such that it first assigns one tuple to each output group (which consumes $c$ tuples), and then it starts assigning the rest $n - c$ tuples according to the distribution generation algorithm. This way, it ensures that the cardinality requirement is fulfilled but the final distribution may not strictly adhere to the original distribution. Here, we assume the user does not specify any contradicting requirements, therefore QAGen uses the first approach.

**Case 2: Input is pre-grouped w.r.t. the grouping attribute** When the input on the grouping attribute is pre-grouped, it is understandable that this operation does not support the group distribution knob as in the above case. But if the input is pre-grouped w.r.t. the grouping attribute and the output cardinality is the only specified knob, it is not a hard problem.

The aggregation operator (v) in the running example (Figure 17.1 (a) falls into this case. Referring to Figure 17.7 (i), which is the input of the aggregation operator in the example. The grouping attribute in the example is $o\_date$, after several joins, the data in $o\_date$ is pre-grouped into 4 pre-groups ($o\_date1 \times 4$; $o\_date2 \times 2$; $o\_date3 \times 1$; $o\_date4 \times 1$). In this case, the aggregation operator controls the output by assigning tuples from the same pre-group to the same output group and each pre-group is assigned into $c$ output groups in a round-robin fashion. In the example, the output cardinality of the aggregation operator is 2. The aggregation operator assigns the first pre-group (with $o\_date1$) which includes 4 tuples into the first output group. Then the second pre-group

| o_date | SUM(l_price) |
|---|---|
| $o\_date1$ | $aggsum\_1$ |
| $o\_date2$ | $aggsum\_2$ |

(i) Output of $\chi$ (2 tuples)

| symbol | predicate |
|---|---|
| $c\_acctbal1$ | $[c\_acctbal1 \geq p_1]$ |
| $c\_acctbal2$ | $[c\_acctbal2 \geq p_1]$ |
| $c\_acctbal3$ | $[c\_acctbal3 < p_1]$ |
| $c\_acctbal4$ | $[c\_acctbal4 < p_1]$ |
| $l\_price1$ | $[aggsum\_1 = 5 \times l\_price1]$ |
| $l\_price5$ | $[aggsum\_2 = 3 \times l\_price5]$ |

(ii) PTable

Figure 17.9: Symbolic Database after aggregation

| o_date | SUM(l_price) |
|---|---|
| $o\_date1$ | $aggsum1$ |

Figure 17.10: Output of HAVING clause (1 tuple)

(with $o\_date2$) which includes 2 tuples is assigned to the second output group. When the third pre-group (with $o\_date3$) which includes 1 tuple is being assigned to the first output group (because of round-robin), the aggregation operator replaces $o\_date3$ with $o\_date1$ in order to put the 5 tuples into the same group. Similarly, the aggregation operator replaces $o\_date4$ from the input tuple with $o\_date2$. For the aggregation function, each output group $g_i$ invokes the Simple Aggregation Operator in Section 17.2.4 with all the symbols participated in the aggregation function as input, and gets a new symbol $agg_{g_i}$ as output. Finally, for each group, the operator constructs a new symbolic tuple $\langle g_i, agg_{g_i} \rangle$ and returns it to the parent. Figure 17.9 (i) shows the output of the aggregation operator, and Figure 17.9 (ii) shows the updated $PTable$ after the aggregation in the running example. Furthermore, since the aggregation operator involves attributes $o\_date$ and $l\_price$, the $orders$ table and the $lineitem$ table are also updated (Figure 17.1 (c) shows the updated tables).

### HAVING and Single GROUP-BY Attribute

In most cases, dealing with a HAVING clause is the same as having a selection operator on top of the aggregation result.

Figure 17.1 (c) shows the $PTable$ content after the HAVING clause. It imposes two more constraints: $[aggsum1 \geq p2]$ which is the positive tuple and $[aggsum2 < p2]$ which is the negative tuple, and it returns Figure 17.10 to the parent.

**Special case of GROUP-BY with HAVING:** There is a special case for the aggregation operator together with the HAVING clause. When there are more than one parameter in the query which influences the number of tuples of each output group implicitly, it is necessary to ask the user to define the count of each output group explicitly. The following is an example:

```
SELECT o_date, SUM(l_price)
FROM Orders, Lineitem
WHERE o_id = l_oid
AND l_price>=:p1
GROUP BY o_date
HAVING SUM(l_price)<=:p2
```

In this query, the parameter $p1$ and $p2$ implicitly affect the number of tuples that can pass through the HAVING clause. For example, during data instantiation phase, if $p1$ gets a value of 50 and $p2$ gets a value of 200, then only groups with less than 4 tuples can pass through the HAVING clause. In other words, if the user wants to control the output cardinality of the HAVING clause, she has to first control the number of tuples of each group. When the query analyzer detects this case, it prepares the following knobs for the user:

| | |
|---|---|
| **Knobs:** | (a) positive group-count $gc^+$ and number of positive output groups $c^+$ (optional; default: $gc^+ >= 1$, $c^+$ = input size) |
| | (b) negative group-count $gc^-$ and number of negative output groups $c^-$ (optional; default: $gc^- >= 1$, $c^-$ =0) |

The knob $c^+$ defines the number of groups which should pass through the HAVING selection and its coexisting knob $gc^+$ defines the number of tuples for every positive group. The knob $c^-$ defines the number of groups which should not pass through the HAVING selection and its coexisting knob $gc^-$ defines the number of tuples for every negative group. The positive group-count ($gc^+$) and the negative group-count ($gc^-$) can be given in terms of a lower or a upper bound. The number of positive and negative groups together must be the same as the output cardinality of the aggregation operator (i.e. $c^+ + c^- = c$).

In the following, we discuss the algorithms to implement this special case. The hardness of the problem depends on whether the input is pre-grouped w.r.t. the group-by attribute or not. Note that, in both cases the user cannot control how to assign the input tuples into different output groups because this would conflict with the above knobs.

**Special case of GROUP-BY with HAVING, sub-case 1: Input is not pre-grouped w.r.t. the group-by attribute** Assume that the aggregation operator of the query above gets an input of 10 tuples which is not pre-grouped w.r.t. the group-by attribute $o\_date$. Furthermore, the user defines the following knob values: $gc^+ \geq 2$, $c^+ = 3$, $gc^- \leq 1$, $c^- = 2$. Thus the output cardinality of the aggregation operator is $c = 5$ in the example. The following illustrates the desired output:

| |
|---|
| $gc^+ \geq 2$ |
| $gc^+ \geq 2$ |
| $gc^+ \geq 2$ |
| $gc^- \leq 1$ |
| $gc^- \leq 1$ |

In this special case, the symbolic execution of the aggregation operator controls the output as follows:

1. [Assigning tuples to output groups with a upper bound group-count] During its open() method, it first assigns one tuple to each output group with upper bound group-count.
2. [Assigning tuples to output groups with a lower bound group-count] Assign the minimum number of tuples to each output group with lower bound group-count.
3. [Post-processing] If there are still some tuples in the input which are not assigned to an output group, then assign these input tuples to the output groups as follows: (a) if there are some output groups with lower bound group-count, then assign all remaining tuples to one of these output groups; (b) if there are only output groups with upper bound group-count, then assign tuples to those output groups until its upper-bound has been reached.
4. [Aggregating] During each getNext() call, get an output group $O_i$, invoke the Simple Aggregation Operator (Section 17.2.4) like the normal case does.
5. [Result Returning] Construct a new symbolic tuple $\langle t.g, agg_i \rangle$ and returns this tuple to its parent, where $agg_i$ is the symbolic tuple returned by the Simple Aggregation Operator for the group $O_i$ and $t.g$ is the symbol of the group-by attribute of $O_i$. Return the constructed tuple to its parent.

In the example, the negative output groups uses $gc^- \leq 1$ as the knob value. Therefore, each of the two negative group gets one tuple during Step 1. The positive output groups uses $gc^+ \geq 2$ as the knob value. Thus each of the three positive output groups gets two tuples during Step 2. The two remaining tuples out of the 10 input tuples are distributed to the first positive output group.

**Special case of GROUP-BY with HAVING, sub-case 2: Input is pre-grouped w.r.t. the group-by attribute** This sub-case contains the $\mathcal{NP}$-complete Group Assignment Problem defined in Appendix B and is therefore $\mathcal{NP}$-hard. In fact, this special case rarely happens in practice. Nonetheless, we present an efficient heuristic that solves most of the instances that arise in practice. In case there are some group-count constraints that cannot be satisfied, the system alerts the user and suggests her to change the knob values.

The heuristic is inspired by the best fit decreasing algorithm (BFD) for the bin packing problem [Joh74]. The basic idea of the BFD algorithm is that it considers the items in the order of non-increasing item sizes. Among the possible bins for an item, the algorithm always chooses the one that would have minimum leftover space after addition of that item. If an item fits in no bin, a new bin is opened.

In our context, we treat a resulting output group as a bin and a pre-group of $k$ input tuples as an item in size $k$. When all group constraints are upper bound constraints (e.g., $gc^{+} \leq 2$ and $gc^{-} \leq 2$), we have a classical bin packing problem with different bin sizes and a fixed number of bins. Basically, the resulting problem asks for a feasible packing for the given bin sizes. For this case we propose to execute the BFD algorithm as sketched above (with all bins being initially open).

When the group constraints consist of mixed greater equal and lower equal constraints (sub-case 1 above is in this case), we have a bin packing and filling problem with $p$ *packing* (lower equal constraints) and $c$ *covering* (greater equal constraints) bins. It becomes trivial to fulfill all lower equal constraints. Without loss of generality, for the $p$ lower equal constraints we can assign the $i$-th smallest item (pre-group) to the $i$-th smallest packing bin (output group) for $1 \leq i \leq p$.

It remains to clarify how to deal with the covering bins. For this problem we propose to iteratively search for a solution that satisfies as many constraints as possible. To this end we search for solutions that cover the $c' \leq c$ w.l.o.g. smallest covering bins, starting at $c' = c$. Although theoretically a binary search would be faster for finding the maximum $c'$ we expect that for real instances $c'$ will be very close to $c$, which justifies a linear search. For a given $c'$ the algorithm relies on the observation that a good cover of the bins overpacks these as little as possible. Therefore, we propose an analogous approach to best fit decreasing. Execute the best fit decreasing algorithm to fill the bins as good as possible.

### Multiple GROUP-BY Attributes

If there is a set of group-by attributes $G$ (with multiple attributes), then the implementation of the aggregation operator depends not only on whether the input is pre-grouped, but also depends on whether the group-by attributes in the input have a tree-structure or have a graph-structure (see Chapter 16). QAGen currently supports queries with tree-structure group-by attributes (see Figure 15.3). Studying the problem of controlling the output cardinality of an aggregation operator with graph-structure group-by attributes is part of the future work.

The aggregation operator treats aggregation with multiple group-by attributes in the same way as the case of a single group-by attribute (Section 17.2.4). Assume attribute $a_n$ is the attribute in $G$ which is functional dependent on the least number of other attributes in $G$. The aggregation operator treats $a_n$ as the single group-by attribute and set the rest of attributes in $A$ to a constant value $v$ (attribute $a_n$ is selected because it has the largest number of distinct symbols in the input comparing to the other attributes).

As an example, assume the following table is an input to an aggregation operator.

| SUM(l_price) |
|---|
| $aggsum1$ |

Figure 17.11: Output of $\pi$(1 tuple)

| $b$ | $c$ | $d$ |
|---|---|---|
| $b_1$ | $c_1$ | $d_1$ |
| $b_2$ | $c_1$ | $d_1$ |
| $b_3$ | $c_2$ | $d_1$ |

Assume the set of group-by attributes $A$ is $\{b, c, d\}$, and the functional dependencies which hold on the input of the aggregation operator are: $\{b\} \rightarrow \{c, d\}$ and $\{c\} \rightarrow \{d\}$. According to the definition in Chapter 16, the set of group-by attributes $G$ has a tree-structure.

In the input above, attribute $b$ is functional dependent on least other attributes in $G$ ($b$ is functional dependent on no attributes where $d$ is functional dependent on $b$ and $c$). As a result, the aggregation operator treats attribute $b$ as the single group-by attribute and invokes the single group-by aggregation implementation. Other attributes use the same symbol for all input tuples (e.g., set all symbols for attribute $c$ to be $c1$).

Since the aggregation operator with multiple-group attributes essentially is handled by the aggregation operator that supports a single group-by attribute, it shares the same special cases (HAVING clause on top on an aggregation where the parameter values control the group count) as the case of aggregation with a single group-by attributes.

### 17.2.5 Projection Operator

Symbolic execution on a projection operator is exactly the same as the traditional query processing, it projects the specified attributes and no additional constraints are added. As a result, the final projection operator in the running example takes in the input from Figure 17.10 and ends with the result shown in Figure 17.11.

### 17.2.6 Union Operator

In SQL, the UNION operator eliminates the duplicates if they exist. On the other hand, the UNION ALL operator does not eliminate the duplicates. In SQP, the query analyzer does not offer any knob to the user to tune the UNION ALL operation. Therefore, the symbolic execution of the UNION ALL operation is straightforward to implement: it reuses the UNION ALL operator in RDBMS and unions the two inputs into one.

For the UNION operation, in SQP, the query analyzer offers the following knob to the user:

**Knob:** Output Cardinality $c$ (optional; default value = size of $R$ + size of $S$)

Let $R$ and $S$ be the inputs of the UNION operation which are not pre-grouped. The symbolic execution of the UNION operator controls the output as follows:

1. During its getNext() call, if the output cardinality has not yet reached $c$, then (a) get a tuple $t$ from $R$ (or from $S$ alternatively); and (b) return $t$ to its parent. However, during the getNext() call, if the output cardinality has reached $c$ already, then process [Post-processing] below, and return null to its parent.

2. [Post-processing] Fetch the remaining tuples $T^-$ from inputs $R$ and $S$, set the symbols in tuple $t^- \in T$ to have the same symbol as one of the returned tuple $t$ in the previous step.

### 17.2.7 Minus Operator

In SQL, the MINUS operator selects all distinct rows that are returned by the query on the left hand side but not by the query on the right hand side.

Let $R$ and $S$ be the non-pregrouped inputs of the MINUS operation. In this case, the query analyzer offers the following knob to the user:

**Knob:** Output Cardinality $c$ (optional; default value = size of $R$)

The symbolic execution of the MINUS operator controls the output as follows:

1. During its getNext() call, if the output cardinality has not yet reached $c$, then (a) get a tuple $r^+$ from $R$, and; (b) return $r^+$ to its parent. However, during the getNext() call, if the output cardinality has reached $c$ already, then process [Post-processing] below, and return null to its parent.

2. [Post-processing] Fetch a tuple $r^-$ from $R$, fetch all tuples $S^-$ from $S$, set the symbols in tuple $s^- \in S^-$ to have the same symbol as $r^-$.

### 17.2.8 Intersect Operator

**Knob:** Output Cardinality $c$ (optional; default value = size of $R$)

In SQL, the INTERSECT operator returns all distinct rows selected by both queries. Currently, QAGen supports INTERSECT with non-pregrouped inputs. Let $R$ and $S$ be the input of the INTERSECT operator, the symbolic execution of the INTERSECT operator is as follows:

1. During its getNext() call, if the output cardinality has not yet reached $c$, then (a) get a tuple $r^+$ from $R$, and get a tuple $s^+$ from $S$; (b) set the symbols of $s^+$ as same as $r^+$ and return $r^+$ to its parent. However, during the getNext() call, if the output cardinality has reached $c$ already, return null to its parent.

### 17.2.9 Processing Nested Queries

Nested queries in symbolic query processing reuses the techniques in traditional query processing because queries can be unnested by using join operators [GW87]. In order to allow a user to have full control on the input, the user should give the input query in its unnested format. If the inner query and the outer query refer to the same table(s), then the query analyzer disables some knobs on operators that may allow a user to specify different constraints on the operators that work on the same table in both inner and outer query.

Chapter 18

# Data Instantiator

*Logic takes care of itself; all we have to do is to look and see how it does it.*

*– Ludwig Wittgenstein, 1889-1951 –*

The final phase of the data generation process is the data instantiation phase. The data instantiator fetches the symbolic tuples from the symbolic database and uses a constraint solver (strictly speaking, the constraint solver is the decision procedure of a model checker [CGP00]) to instantiate concrete values for them. The constraint solver takes a propositional formula (remember that a predicate can be represented by a formula in propositional logic) as input and returns a set of concrete values for the symbols in the formula that satisfies all the input predicates and the actual data types of the symbols. If the input formula is unsatisfiable, the constraint solver returns an error. Such errors, however, cannot occur in this phase because we assume there are no contradicting knob values. A constraint solver call is an expensive operation. In the worst case, the cost of a constraint solver call is exponential to the size of the input formula [CGP00]. As a result, the objective of the data instantiator is to minimize the number of calls to the constraint solver if possible. Indeed, the predicate size optimizations during symbolic query processing (e.g. reducing $aggsum = l\_price1 + \ldots + l\_price8$ to $aggsum = l\_price1 \times 8$) are designed for this purpose. After the data instantiator has collected all the concrete values of a symbolic tuple, it inserts the instantiated tuple into the final test database. The details of the data instantiator are as follows:

1. The process starts from any one of the symbolic tables.
2. It reads in a tuple $t$, say $\langle c\_id1, c\_acctbal1 \rangle$, from the symbolic tables.
3. [Look up symbol-to-value cache] For each symbol $s$ in tuple $t$, (a) it first looks up $s$ in a table called $SymbolValueCache$ in the symbolic database. The $SymbolValueCache$ is a table in the symbolic

database that stores the concrete values of the symbols that have been instantiated by the constraint solver; (b) if symbol $s$ has been instantiated with a concrete value, then the symbol is initialized with the same cached value and then proceeds with the next symbol in $t$.

In the running example, assume the constraint solver randomly instantiates the $customer$ table (4 tuples) first. Since symbol $c\_id1$ is the first symbol to be instantiated, it has no instantiated value stored in the $SymbolValueCache$ table. However, assume later when instantiating the first two tuples of $orders$ table (with $o\_id1$, $o\_id2$), their $o\_cid$ values will use the same value as instantiated for $c\_id1$ by looking up the $SymbolValueCache$.

4. [Instantiate values] Look up predicates $P$ of $s$ from the $PTable$. (a) If there are no predicates associated with $s$, then instantiate $s$ by a unique value that matches the actual domain of $s$ in input schema $S$.

   In the example, $c\_id1$ does not have any predicates associated with it (see $PTable$ in Figure 17.1). Therefore, the data instantiator does not instantiate $s$ with a constraint solver but instantiates a unique value $v$ (because $c\_id$ is a primary-key), say, 1, to $c\_id1$. Afterwards, insert a tuple $\langle s, v\rangle$ (e.g., $\langle c\_id1, 1\rangle$) to the $SymbolValueCache$.

   (b) However, if $s$ has some predicates $P$ in the $PTable$, then compute the *predicate closure* of $s$. The predicate closure of $s$ is computed by recursively looking up all the directly correlated or indirectly correlated predicates of $s$.

   For example, the predicate closure of $l\_price1$ is [$aggsum1$ = 5 $\times$ $l\_price1$ AND $aggsum1 \geq p2$]. Then the predicate closure (which is in the form of a formula in propositional logic) is sent to the constraint solver (symbols that exist in the $SymbolValueCache$ are replaced by their instantiated values first). The constraint solver instantiates all symbols in the formula in a row (e.g., $l\_price1 =$ 10, $aggsum1 = 50$, $p2 = 18$).

   For efficiency purposes, before a predicate closure is sent to the constraint solver, the data instantiator looks up another cache table called $PredicateValuesCache$ in the symbolic database. This table caches the instantiated values of predicates. Since many predicates in the $PTable$ essentially share the same pattern, the predicates stored in $PredicateValuesCache$ are in the predicate pattern format. For example, predicates [$c\_acctbal1 \geq p1$] and [$c\_acctbal2 \geq p1$] in Figure 17.1 (c) share the same pattern: [$c\_acctbal \geq p1$]. As a result, after the instantiation of predicate [$c\_acctbal1 \geq p1$], the data instantiator inserts an entry $\langle [c\_acctbal \geq p1], c\_acctbal1, p1\rangle$ into the $PredicateValuesCache$ table. When the next predicate closure [$c\_acctbal2 \geq p1$] needs to be instantiated, the data instantiator looks up the predicate in $PredicateValuesCache$ by its pattern; if the same predicate pattern is in $PredicateValuesCache$, then the data instantiator skips the instantiation of this predicate and reuses the instantiated value of $c\_acctbal1$ in the $SymbolValueCache$ table for symbol $c\_acctbal2$ (same for $p1$).

The number of constraint solver calls is minimized by the introduction of the $SymbolValueCache$ and $PredicateValuesCache$ tables. Experiments show that this feature is crucial or otherwise generating a 1G query-aware database takes weeks instead of hours. Finally, note that in Step 4 (a), if a symbol $s$ has no predicate associated with it, the data instantiator assigns a value to $s$

according to its domain and its related integrity constraints (e.g., primary-keys). In general, those values can be assigned randomly or always use the same value. However, it is also possible to instantiate some extra data characteristics (e.g., distribution) for those symbols to test certain aspects of the query optimizer even though those the values of symbols would not affect the query results.

Chapter 19

# Semi-Automatic DBMS Testing

*The first rule of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency.*

*– Bill Gates, born 1955 –*

So far, the discussion of QAGen is restricted to having a complete test case as input and generating a query-aware test database as output. A test case, as shown in Figure 14.1, has to consist of a logical query plan of a SQL query $Q_P$ and a set of knob values defined on each query operator. In practice, the most tricky job is to determine different sets of interesting knob values for the test query in order to form different useful test cases. Currently, the knob values of a test case are manually chosen by the testers. In this chapter, we discuss the possibilities of automating this step.

In software engineering, there exist different test design techniques and coverage metrics which assist the tester in creating a useful test suite for a program (test object) by generating test cases with different combinations of *interesting* parameter values [AO94]. One way of choosing the interesting values of a parameter is called the Category Partition (CP) method [OB88]. The CP method suggests the tester first partitions the domain of a parameter into subsets (called *partitions*) based on the assumption that all points in the same subset result in a similar behavior from the test object. The tester should select one value from each partition to form the set of interesting values.

Consider a simple query $R \bowtie S$ joining two tables $R$ and $S$. Assume table $R$ has 1000 tuples and table $S$ has 2000 tuples and the two tables are not connected by foreign-key constraint. In this case, the interesting values for the output cardinality knob for the join could be formed by partitioning the possible knobs values into, say 4 partitions: Extreme case partition (0 tuple), Minimum case partition (1 tuple), Normal case partition (500 tuples), and Maximum case partition (1000 tuples).

$$\begin{array}{l} T_1 : \{A = a1, B = b1, C = c1\} \\ T_2 : \{A = a2, B = b2, C = c2\} \end{array}$$

(a) Each-used Coverage

$$\begin{array}{l} T_1 : \{A = a1, B = b1, C = c1\} \\ T_2 : \{A = a1, B = b2, C = c2\} \\ T_3 : \{A = a1, B = b1, C = c2\} \\ T_4 : \{A = a2, B = b1, C = c1\} \\ T_5 : \{A = a2, B = b2, C = c2\} \\ T_6 : \{A = a2, B = b2, C = c1\} \end{array}$$

(b) Pair-wise Coverage

Figure 19.1: Coverage Example

In addition, Uniform distribution and Zipf distribution can be regarded as two partitions of the join key distribution knob.

Having decided the set of interesting values for each parameter (knob), the next step is to combine those values to form different test cases (i.e., a test suite). There are different algorithms (known as combination strategies) to combine the interesting values and form different test suites. Each algorithm will result in a test suite that achieves certain *coverage*. The following are some well-known coverage criteria for combination strategies:

- *Each-used*. The *Each-used* coverage is also known as *1-wise* coverage. It is the simplest coverage criterion that requires every interesting value of every parameter to be included in at least one test case in the test suite. Consider a program with three parameters $A$, $B$ and $C$ and the interesting values (selected from each partition) of each parameter are $\{a1, a2\}$,$\{b1, b2\}$ and $\{c1, c2\}$ respectively. An example test suite that satisfies the *Each-used* coverage is shown in Figure 19.1 (a), which includes two test cases $T_1$ and $T_2$.

- *Pair-wise*. The *Pair-wise* coverage is also known as 2-wise coverage. It requires that every possible pair of intersecting values of any two parameters is included in some test cases in the test suite. Consider the same example program as above, an example test suite that satisfies the *Pair-wise* coverage is shown in Figure 19.1 (b), which includes six test cases.

- *T-wise*. The *t-wise* coverage [WP01] is a generalization of the above two coverages which requires that every possible combination of intersecting values of $t$ parameters to be included in some test cases in the test suite.

- *Variable strength*. The *Variable strength* coverage [CGMC03] allows different coverages on different sets of parameters. For example, it requires a higher coverage (e.g., *2-wise*) among the parameter $A$ and $B$ and a lower coverage (e.g., *1-wise*) on the parameter $C$ in our example program.

- *N-wise*. The *N-wise* coverage requires if there are $N$ parameters, then all possible combinations of interesting values should be included in some test cases in the test suite.

$$T_1 : \{\sigma_1 = 1(min), \bowtie = 1(min), \sigma_2 = 1(min)\}$$
$$T_2 : \{\sigma_1 = 1(min), \bowtie = 1(max), \sigma_2 = 1(min)\}$$
$$T_3 : \{\sigma_1 = 1(min), \bowtie = 2000(max), \sigma_2 = 2000(max)\}$$
$$T_4 : \{\sigma_1 = 1000(max), \bowtie = 1(min), \sigma_2 = 1(min)\}$$
$$T_5 : \{\sigma_1 = 1000(max), \bowtie = 2000(min), \sigma_2 = 2000(max)\}$$
$$T_6 : \{\sigma_1 = 1000(max), \bowtie = 1(max), \sigma_2 = 1(min)\}$$

Figure 19.2: A pair-wise test suite generated by current combination strategies

Each coverage criterion has its own pros and cons and they serve for different types of applications. There are different combination strategies to generate test suites that satisfy different coverage criteria. For example, the AETG algorithm [CDFP97] is a non-deterministic algorithm that generates test suites which satisfy the *Pair-wise* coverage.[1] As another example, the *Each Choice* algorithm [AO94] is a deterministic algorithm that generates test suites which satisfy the *Each-used* coverage. However, these algorithms cannot be directly applied to our automatic testing framework.

The first problem is that the knobs are correlated to each other in a knob-annotated QAGen execution plan. As a result, it is not easy to do category partitioning. As an example, it is difficult to partition the cardinality of the root (aggregation) operator of TPC-H Query 8 (see Figure 20.2 (a)) because the interesting value of the maximum case partition (i.e., the maximum number of output groups) depends on the cardinalities of its child operators.

The second problem is that the correlation of operators in a knob-annotated QAGen execution plan causes existing combination strategies to generate test suites that may not satisfy the coverage criterion. For example, consider a select-join query $\sigma_1(R) \bowtie \sigma_2(S)$ where $R$ has 1000 tuples and $S$ has 2000 tuples, and $S$ has a foreign-key referring to $R$ on the join attribute. Assume that we are able to determine the minimum and the maximum cardinality of each operator:

| | min | max |
|---|---|---|
| $\sigma_1$ | 1 | 1000 |
| $\sigma_2$ | 1 | 2000 |
| $\bowtie$ | 1 | 2000 |

Then, according to the existing *Pair-wise* test suite combinational strategies, a test suite like the one in Figure 19.2 would be returned. However, if we look closer to the test suite in Figure 19.2, we can find out that the generated test suite actually does not strictly fulfill the *Pair-wise* criterion. For test case $T_1$ and $T_2$, the selections on $R$ and $S$ return 1 tuple (minimum case partition). Consequently, no matter the output cardinality of the join is defined as the minimum case partition ($T_1$) or the maximum case partition ($T_2$), the join can only return 1 tuple. As a result, $T_1$ and $T_2$ are essentially the same and the final test suite does not make sure every possible pair of interesting values of any two knobs is included.

[1]Non-deterministic algorithms means that it may generate different test suites every time.

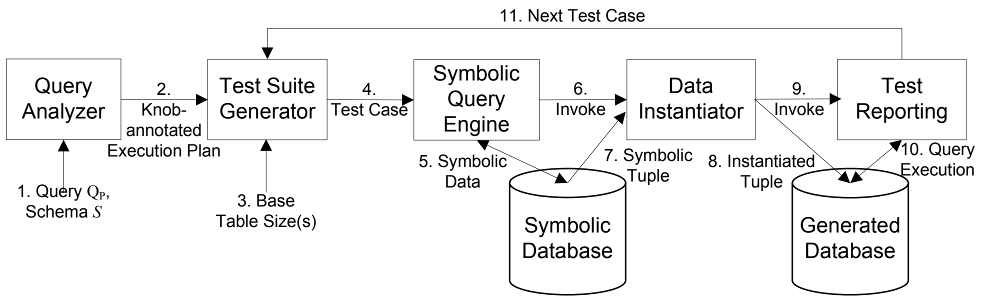


Figure 19.3: Semi-automatic Testing Framework

## 19.1 The Framework

To automate the task of creating a set of meaningful test cases, it is necessary to devise a new set of combination strategies for each coverage that are aware of the aforementioned correlations in a logical query plan. In the next section, a simple method for generating *1-wise* test suites is presented. Discussion on how to design different combination strategies that satisfy different coverages would be an interesting research topic for the software engineering community but is out of the scope of this thesis.

Figure 19.3 shows the semi-automatic DBMS feature testing framework. It is an extension of the QAGen architecture in Figure 15.2. As usual, the tester gives a parametric query $Q_P$ and the schema $S$ as input. After the query analyzing phase, the tester specifies the size of the base tables, and a test suite that satisfies the *1-wise* coverage is generated from the test suite generator. Each test case is then processed by the Symbolic Query Engine and the Data Instantiator and a query-aware test database is generated as usual. Finally, the test query of the test case is automatically executed against the generated database, and the execution details (e.g., the execution plan, cost, time, etc) is inserted into the test report.

Note that, in general, testers use their domain knowledge in order to create input test queries. However, this step can also be automated by query generation tools (e.g., RAGS [Slu98] and QGEN [PS04]). In this case, the framework is a fully-automatic testing framework which is useful to do some higher level testings such as regression test or integration test on the system.

## 19.2 Test Case Generation

The current testing framework can generate a test suite that satisfies the *1-wise* coverage. One reason for using *1-wise* coverage in the framework is that there may be many knobs available in a QAGen query execution plan. Defining coverage stronger than *1-wise* (e.g., *2-wise*) may then result in a very large test suite. In addition, based on *1-wise* coverage, it is possible to design an algorithm so that the knob values are not affected by the correlations of the output cardinalities between operators in a query.

The following shows the test case generation algorithm that is used inside the test suite generator. It takes a knob-annotated query plan as input and returns a set of test cases as output.

1. [Creating a test case for each cardinality *1-wise* partition] For each partition $g$ of the output cardinality knob, create a temporary test case $T_g$.

2. [Assigning *1-wise* value to distribution knob] For each temporary test case $T_g$, create a test case $T_{gd}$ from $T_g$ using one distribution knob value $d$. The value $d$ should not be repeated until each value is used once at least.

3. [Assigning real values to the cardinality partition] For each test case $T_{gd}$, parse test query $Q$ of $T_{gd}$ in a bottom-up manner and assign cardinality values to $T_{gd}$ according to Table 19.1. This table shows the minimum and maximum partitions for each symbolic operation. The notation used in the table follows the discussion of Chapter 17. For example, $R$ denotes the input of an unary operator and $|R|$ denotes its cardinality.

Figure 19.4 shows the test case generation process of a simple query $\sigma(R) \bowtie S$. In the current framework, we only consider the minimum and the maximum partitions for the cardinality knob and only Zipf and Uniform distribution for the distribution knob. Although the test generation algorithm is simple, experimental results show that the generated test suite can effectively generate different query-aware test databases that show different system behaviors of a commercial database system. In this thesis, we regard this simple SQL test case generation algorithm as a starting point for this new SQL test case generation problem. In fact, the current testing framework has several restrictions. First, it requires that the same table cannot be used twice in the input of a binary operator, for example, the query $R \bowtie R$ is prohibited. Second, Table 19.1 does not capture the cases of pre-grouping input and the cases of having two disjoint subqueries [Elk89] for a binary operator. Therefore, the computed knob value may not be accurate in these cases. As a result, in the current framework, if the query analyzer detects that there are some operators with pre-grouped input or with disjoint subqueries in the query execution plan, it will prompt the tester to verify that automated computed test case before QAGen starts execution. As part of the future work, we plan to further improve the framework in order to eliminate these restrictions.

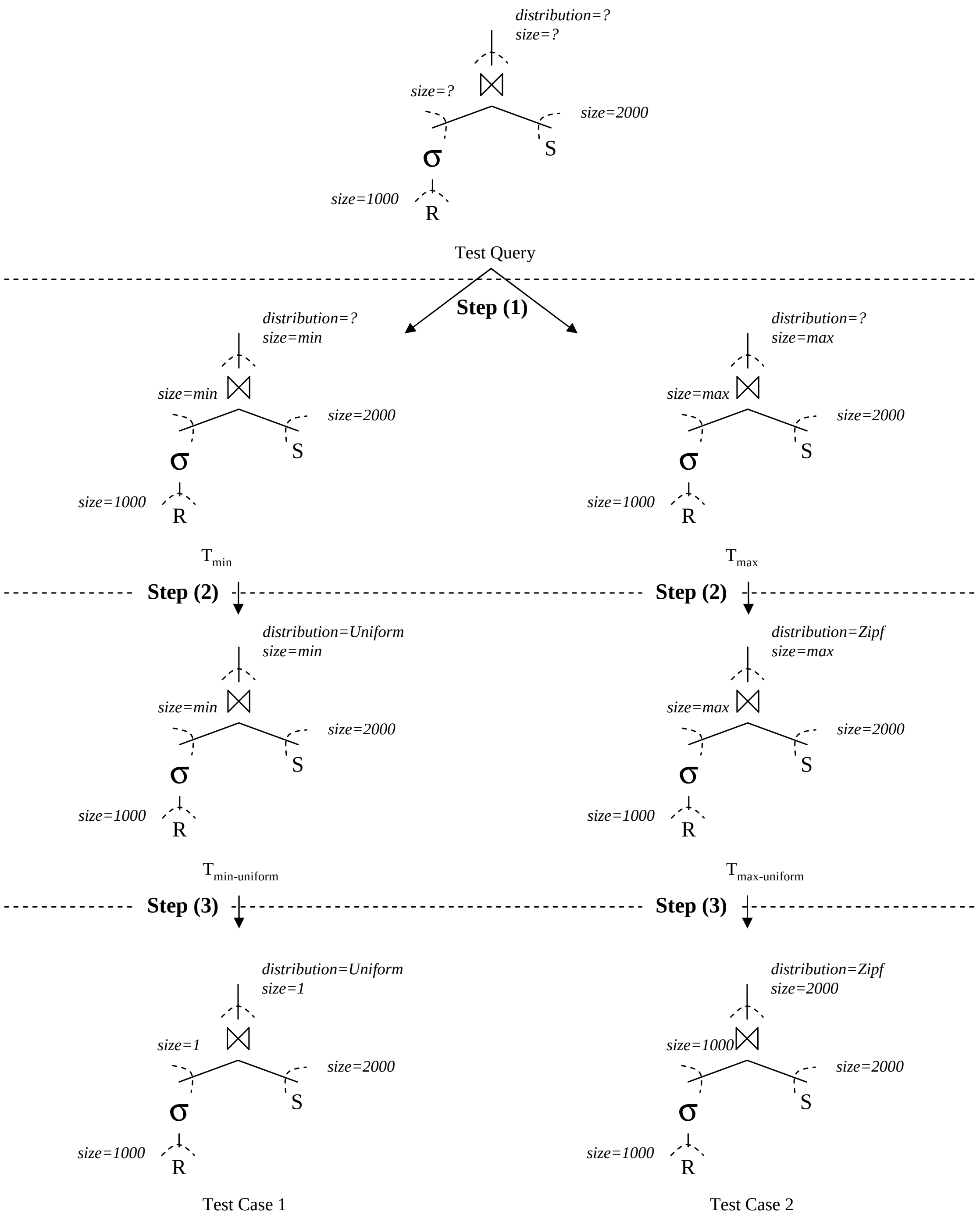


Figure 19.4: Testing Case Generation Example

| Operator | Minimum Partition | Maximum Partition |
|---|---|---|
| Selection | 1 | $\|R\|$ |
| Aggregation | 1 | $\|R\|$ |
| Join | 1 | $\|S\|$ |
| Union | $max(\|R\|, \|S\|)$ | $\|R\| + \|S\|$ |
| Minus | $\|R\| - \|S\|$ | $\|R\|$ |
| Intersect | 1 | $min(\|R\|, \|S\|)$ |

Table 19.1: Knob value table.

# Chapter 20

# Experiments

*The true method of knowledge is experiment.*

*– William Blake, 1757-1827 –*

We have run a set of experiments to test a prototype implementation of QAGen. The implementation is written in Java and is installed on a Linux AMD Opteron 2.2 GHz Server with 4 GB of main memory. The symbolic database and the target database use PostgreSQL 7.4.8 and they are installed on the same machine. As a constraint solver, a publicly available constraint solver called Cogent [CKS05] is used. During the experiments, if the approximation ratio knob is enabled by the query analyzer, the value 0.1 is used.

We execute three sets of experiments with the following objectives: The first experiment (Section 20.1) studies the efficiency of the symbolic execution of individual operators. The second experiment (Section 20.2) studies the scalability of QAGen for generating different database sizes for different queries. The last experiment (Section 20.3) uses the semi-automatic testing framework to generate different test databases for the same query in order to study if the generated test cases could effectively affect the behavior of a commercial database.

## 20.1 Efficiency of Symbolic Operations

The objective of this experiment is to evaluate (1) the running time of individual symbolic operators, (2) their scalability, and (3) the running time of the data instantiation phase by generating three query-aware databases in different scales (10M, 100M, and 1G). The input query is query 8 in the TPC-H benchmark. Its logical query plan is shown in Figure 20.1. We have chosen

TPC-H query 8 because it is one of the most complex queries in TPC-H with 7-way joins and aggregations. This query has various input characteristics to the operators enabling us to evaluate the performance of different operator implementations (e.g., the normal equi-join and the special case of equi-join that needs solving the subset sum problem). The experiments are carried out as follows: First, three benchmark databases are generated using *dbgen* from the TPC-H benchmark. As a scaling factor, we use 10 MB, 100 MB, and 1GB. Then, we execute query 8 on top of the three TPC-H databases, and collect the base table sizes and the cardinality of each intermediate result under the three scaling factors. The extracted cardinality of each intermediate result of query 8 is shown in Table 20.1 (Output-size) columns. Next, we generate three TPCH-query-8-aware databases with the collected base table sizes and output cardinalities as input and measure the efficiency of QAGen for generating databases that produces the same cardinality results. For this experiment, the value distribution between two joining tables is the uniform distribution.

Table 20.1 shows the cost breakdown of generating query-aware databases for TPC-H query 8 in detail. QAGen only takes about 10 minutes for generating a 10MB query-aware database. The symbolic query processing phase is fast and scales linearly. It takes about 1 minute for 10MB and less than 3 hours for 1G database. The longest SQP operations are the initialization of the large symbolic table $Lineitem$ (Line 10 in Table 20.1), and the join between the intermediate result $R5$ and $Lineitem$ (Line 11). That join requires a long time because it accesses the large $Lineitem$ table frequently to update the symbolic values of the join attributes. In query 8, the input is pre-grouped on the last join (Line 17 in Table 20.1 and operator (17) in Figure 20.2) and the approximation ratio knob is enabled. Nevertheless, the equi-join finishes quickly because the input size is not big. Table 20.1 also shows that the symbolic execution of each individual operator scales well.

The data instantiation phase dominates the whole data generation process. It takes about 9 minutes to instantiate a 10M query 8 aware database and about 17 hours to instantiate a 1G query 8 aware database. Nevertheless, about 40% of time is the overhead of reading symbolic tuples and inserting concrete tuples (not shown in the Table). In the experiments, the number of constraint solver (cogent) calls is small – there are only 14 calls for 3 scaling factors. The number of calls is constant because the data instantiator caches the patterns of the predicates but not the concrete predicates. We indeed repeat the same experiment by turning off the caching feature of QAGen, but it ends up that the data instantiation phase for a 1G database cannot finish within 2 weeks because the constraint solver takes a lot of time. It proves that the predicate optimization in SQP and the caching in the data instantiator work effectively.

| # | Symbolic operation | $size = 10M$ | | $size = 100M$ | | $size = 1G$ | |
|---|---|---|---|---|---|---|---|
| | | Output-size | Time | Output-size | Time | Output-size | Time |
| 1 | $Region$ | 5 | $< 1s$ | 5 | $< 1s$ | 5 | $< 1s$ |
| 2 | $\sigma(Region) = R1$ | 1 | $< 1s$ | 1 | $< 1s$ | 1 | $< 1s$ |
| 3 | $Nation$ | 25 | $< 1s$ | 25 | $< 1s$ | 25 | $< 1s$ |
| 4 | $(R1 \bowtie Nation) = R2$ | 5 | $< 1s$ | 5 | $< 1s$ | 5 | $< 1s$ |
| 5 | $Customer$ | $1.5k$ | $< 1s$ | $15.0k$ | $5s$ | $150k$ | $49s$ |
| 6 | $(R2 \bowtie Customer) = R3$ | $0.3k$ | $1s$ | $3.0k$ | $7s$ | $299.5k$ | $75s$ |
| 7 | $Orders$ | $15.0k$ | $4s$ | $150.0k$ | $45s$ | $1.5m$ | $553s$ |
| 8 | $\sigma(Orders) = R4$ | $4.5k$ | $8s$ | $45.0k$ | $67s$ | $457.2k$ | $709s$ |
| 9 | $(R3 \bowtie R4) = R5$ | $0.9k$ | $3s$ | $9.0k$ | $22s$ | $91.2k$ | $277s$ |
| 10 | $Lineitem$ | $60.0k$ | $26s$ | $600.5k$ | $237s$ | $6001.2k$ | $2629s$ |
| 11 | $(R5 \bowtie Lineitem) = R6$ | $3.6k$ | $34s$ | $35.7k$ | $348s$ | $365.1k$ | $4694s$ |
| 12 | $Part$ | $2.0k$ | $< 1s$ | $20.0k$ | $5s$ | $200k$ | $60s$ |
| 13 | $\sigma(Part) = R7$ | 12 | $1s$ | 147 | $8s$ | 1451 | $72s$ |
| 14 | $(R7 \bowtie R6) = R8$ | 29 | $3s$ | 282 | $27s$ | 2603 | $533s$ |
| 15 | $Supplier$ | $0.1k$ | $< 1s$ | $1k$ | $< 1s$ | $10k$ | $3s$ |
| 16 | $(Supplier \bowtie R8) = R9$ | 29 | $< 1s$ | 282 | $1s$ | 2603 | $6s$ |
| 17 | $(Nation \bowtie R9) = R10$ | 29 | $< 1s$ | 282 | $< 1s$ | 2603 | $3s$ |
| 18 | $\chi(R8) = R11$ | 2 | $< 1s$ | 2 | $1s$ | 2 | $10s$ |
| Symbolic Query Processing | | $01m : 20s$ | | $12m : 53s$ | | $161m : 13s$ | |
| Data Instantiation (# Cogent-call) | | $09m : 31s$ (14) | | $96m : 03s$ (14) | | $1062m : 54s$ (14) | |
| Total | | $10m : 51s$ | | $108m : 56s$ | | $1224m : 07s$ | |

Table 20.1: QAGen Execution Time for TPC-H Query 8

## 20.2 Scalability of QAGen

The objective of this experiment is to evaluate the scalability of QAGen for generating a variety of query-aware test databases. Currently, QAGen supports 13 out of 22 TPC-H queries. It does not support some queries because those queries either fall into the special cases of QAGen (e.g., query 5 (Q5) falls into the special case of the selection operator in Section 17.2.2 case 2); or because some of them use non-equi-joins (e.g., Q16, Q22). Nevertheless, we generate query-aware databases for the rest of the queries in three different scaling factors 10M, 100M and 1G. Table 20.2 shows the detailed results. These results show that both phases scale well for all 13 TPC-H queries and the data instantiation (DI) phase is still the time dominating phase.

## 20.3 Effectiveness of the Semi-Automatic Testing Framework

The objective of this experiment is to show how the test databases that are generated by the semi-automatic testing framework can show different behavior of a commercial database. In this experiment, the target database size is fixed at 100MB and the input query is query 8 in TPC-H. The experiments are carried out in the following way: First, we generate four query-aware databases

| Query | Phase | 10M | 100M | 1G |
|---|---|---|---|---|
| 1 | SQP | 02m:40s | 26m:45s | 321m:27s |
| | DI | 07m:42s | 78m:35s | 844m:52s |
| | Total | 10m:22s | 105m:10s | 1166m:19s |
| 2 | SQP | 00m:09s | 01m:32s | 16m:47s |
| | DI | 02m:27s | 24m:55s | 249m:50s |
| | Total | 02m:36s | 26m:27s | 256m:37s |
| 3 | SQP | 01m:35s | 16m:18s | 185m:21s |
| | DI | 09m:34s | 97m:07s | 1016m:59s |
| | Total | 11m:09s | 113m:25s | 1202m:20s |
| 4 | SQP | 02m:32s | 23m:23s | 221m:17s |
| | DI | 06m:10s | 67m:22s | 627m:11s |
| | Total | 08m:42s | 80m:45s | 848m:28s |
| 6 | SQP | 01m:52s | 64m:36s | 180m:22s |
| | DI | 10m:36s | 333m:31s | 1121m:06s |
| | Total | 12m:28s | 398m:07s | 1301:28s |
| 9 | SQP | 03m:08s | 31m:59s | 445m:16s |
| | DI | 09m:01s | 92m:16s | 967m:24s |
| | Total | 12m:09s | 124m:15s | 1412m:40s |
| 10 | SQP | 01m:16s | 12m:56s | 156m:22s |
| | DI | 09m:42s | 98m:13s | 1107m:10s |
| | Total | 10m:58s | 111m:09s | 1263m:32s |
| 12 | SQP | 02m:11s | 21m:32s | 244m:07s |
| | DI | 12m:01s | 123m:04s | 1387m:27s |
| | Total | 14m:12s | 144m:36s | 1631m:34s |
| 14 | SQP | 01m:39s | 08m:47s | 95m:49s |
| | DI | 17m:15s | 94m:50s | 1023m:39s |
| | Total | 18m:54s | 103m:27s | 1119m:28s |
| 15 | SQP | 00m:58s | 09m:10s | 98m:07s |
| | DI | 05m:40s | 92m:24s | 966m:10s |
| | Total | 06m:38s | 101m:34s | 1064m:17s |
| 16 | SQP | 00m:14s | 01m:42s | 27m:01s |
| | DI | 05m:38s | 05m:19s | 52m:40s |
| | Total | 06m:52s | 07m:01s | 79m:41s |
| 18 | SQP | 00m:55s | 08m:20s | 86m:30s |
| | DI | 08m:41s | 86m:53s | 861m:11s |
| | Total | 09m:36s | 95m:13s | 947m:41s |
| 19 | SQP | 04m:14s | 41m:45s | 411m:12s |
| | DI | 97m:23s | 973m:03s | 9707m:11s |
| | Total | 101m:37s | 1014m:48s | 10118m:23s |

Table 20.2: QAGen Scalability

| Result | TPC-H(Uniform/Zipf) | MIN-Uniform | MAX-Zipf |
|---|---|---|---|
| $R1$ | 1 | 1 | 5 |
| $R2$ | 5 | 1 | 25 |
| $R3$ | $3k$ | 1 | $15k$ |
| $R4$ | $45k$ | 1 | $150k$ |
| $R5$ | $9k$ | 1 | $150k$ |
| $R6$ | $36k$ | 1 | $600k$ |
| $R7$ | 147 | 1 | $20k$ |
| $R8$ | 282 | 1 | $600k$ |
| $R9$ | 282 | 1 | $600k$ |
| $R10$ | 282 | 1 | $600k$ |
| $R11$ | 2 | 1 | 2 |
| Execution Plan | Figure 20.2 (a) | Figure 20.2 (b) | Figure 20.2 (c) |

Table 20.3: Knob Values and Resulting Execution Plans

for TPC-H query 8. Then, we execute query 8 on the four generated databases (on PostgreSQL) and study their physical execution plans. The first database [MIN-Uniform] is automatically generated by the testing framework using the minimum case partition. The database will let query 8 to have the minimum cardinality on each intermediate result during execution. In the [MIN-Uniform] database, the key values between two joining relations have a Uniform distribution. Furthermore, during a grouping operation, tuples will be uniformly distributed into different groups in the [MIN-Uniform] database. The second database [MAX-Zipf] is also generated by the test framework using the maximum case partition with a Zipf distribution. The third database [TPCH-Uniform] is manually added to the test suite and is generated by QAGen using the intermediate result sizes extracted from executing query 8 on TPC-H *dbgen* database (as in the first experiment above). The last database [TPCH-Zipf] is generated by QAGen using the same intermediate result sizes as [TPCH-Uniform] but with a Zipf distribution. Table 20.3 shows the intermediate result sizes of the above set up.

Figure 20.2 shows the physical execution plans of executing TPC-H query 8 on the generated query-aware databases. By controlling the output cardinalities of the operators, it causes PostgreSQL to use different join strategies. For example, when the cardinality of each output is minimum [MIN-Uniform], PostgreSQL tends to use a left-deep-join order (Figure 20.2 (b). When the cardinality of each output is maximum [MAX-Zipf], PostgreSQL tends to use a bushy-tree join order (Figure 20.2 (c). The output cardinalities also strongly influences the choice of physical operators; when the output cardinality is large, PostgreSQL tends to use hash joins (Figure 20.2c). However, when the output cardinality is small, PostgreSQL tends to use fewer hash joins but used sort-merge-joins and nested-loop-joins (Figure 20.2 (a), (b)). The input and output cardinality also influence the choice of physical aggregation operators. When the input to the aggregation (i.e., $R10$ in Table 2) is minimum or same as the TPC-H size, then PostgreSQL tends to use group aggregation (Figure 20.2 (a), (b)). However, when the input to is maximum, then PostgreSQL

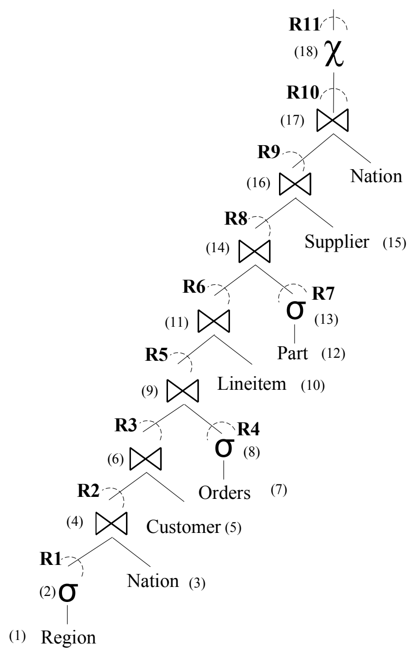


Figure 20.1: TPC-H Query 8: Logical Query Plan

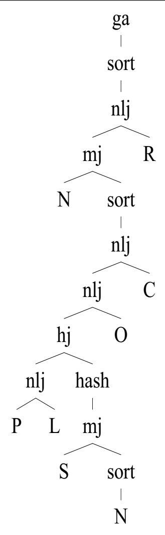
tends to first do a hash aggregation and then sort it (Figure 20.2 (c)).

Controlling the distributions of the query operators shows that the operators in PostgreSQL are less sensitive to the data distribution. For example, when the cardinality is same as TPCH size (Figure 20.2 (a)), the distribution knob does not influence the execution plans. Moreover, the distribution knob also has less influences on the choice of physical operators.

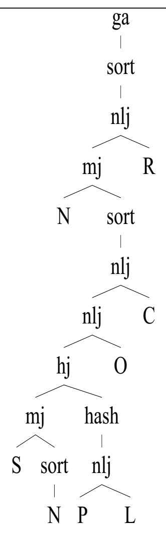
In this experiment, we attempt to use other database generation tools to generate the same set of test databases which can produce the same intermediate query results. We try to run this experiment with two commercial test database generators, DTM Data Generator and IBM DB2 Test Database Generator, and one research prototype [HTW06][1]. However, these tools only allow constraining the base tables properties and we fail to manually control the intermediate result sizes for the purpose of this experiment. Another attempt is to use the query parameter generation tool from [BCT06] to generate query parameters on top of the generated databases. However, that tool can only support select-project-join queries (with single-sided or double-side predicates) which is not suitable for the complex TPC-H queries (which include aggregations and complex predicates) in this experiment.

[1]We also attempt to evaluate DGL from Microsoft [BC05], however their tool is not publicly available.

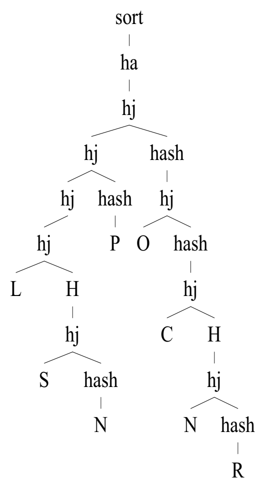
**Base Tables:** $C = Customer$, $N = Nation$, $L = Lineitem$, $O = Orders$
$P = Part$, $R = Region$, $S = Supplier$
**Physical Operators:** $ga$ = *Group Aggregate*, $hash$ = *Hash*, $ha$ = *Hash Aggregate*, $hj$ = *Hash Join*
$mj$ = *Merge Join*, $nlj$ = *Nested Loop Join*, $sort$ = *Sort*



(a) [TPCH-Uniform] and [TPCH-Zipf]



(b) [MIN-Uniform]



(c)[MAX-Zipf]

Figure 20.2: Physical Execution Plans of TPC-H Query 8

# Chapter 21

# Related Work

*Math and music are intimately related.*
*Not necessarily on a conscious level, but sure.*

*– Stephen Sondheim, born 1930 –*

The closest related work in DBMS testing is the work of [BCT06] which studies the generation of query parameters for test queries with given test databases. However, existing database generation tools such as IBM DB2 Database Generator and others (e.g., [GSE$^+$94], [HTW06], [BC05]) were designed to generate general-purpose test databases without any concern for the test queries, and thus the generated databases cannot guarantee sufficient coverage of specific test cases. As a consequence, [BCT06] can hardly find a good database to work on and eventually only a very limited subset of SQL is supported.

QAGen extends symbolic execution [Kin76] and proposes the concept of symbolic query processing (i.e., SQP) to generate query-aware databases. SQP is related to constraint databases (e.g., [Kui02]); however, constraint databases focus on constraints that represent infinite concrete data (e.g., spatial-temporal data) whereas SQP works on finite but abstract data.

The semi-automatic testing framework in this part is related to a number of software testing research work. For example, [AO94] first states the test case selection problem for traditional program testing. Some solutions for the traditional test case selection problem can be found in [AO94; CDFP97; CGMC03; WP01; GOA05].

# Part V

# Summary

Chapter 22

# Conclusions and Future Work

*What is now proved was once only imagined.*

*– William Blake, 1757-1827 –*

This thesis presented two innovative techniques for specifying and generating *test case aware databases* and discussed several applications of these techniques.

Part II presented a new technique called Reverse Query Processing or RQP, for short. RQP combines techniques from traditional query processing (e.g., query rewrite and the iterator model) and model checking (e.g., data instantiation based on constraint formulae of propositional logic). The main application of RQP is the generation of databases for the testing of OLAP applications. It could be shown that a full-fledged RQP engine for SQL (called SPQR) can be built and that it scales linearly with the size of the databases that need to be generated for the TPC-H benchmark.

Part III discussed two more applications of RQP in detail (i.e., the functional testing of OLTP applications as well as the functional testing of a query language) and presented the necessary extensions of RQP. RQP for the functional testing of a query language is currently used in an industrial environment at Microsoft for the testing of the query processing capabilities of the new ADO.Net Entity Framework.

For many other applications of RQP (e.g., Update of Views, Program Verification) significant additional research is needed in order to exploit the potential of RQP. Consequently, the most important avenue for future work is to further explore these applications. Furthermore, additional work is required in order to develop techniques for RQP which guarantee certain properties of the generated data (e.g., minimality). In addition, it is going to be important to leverage recent developments of the model checking community.

Finally, Part IV presented another technique called Symbolic Query Processing or SQP, for short. SQP combines the techniques from traditional query processing (e.g., the iterator model) and symbolic execution from software engineering (e.g., representing concrete data by symbols). The main application of SQP is to generate test case aware databases for the testing of individual DBMS components. A prototype system QAGen which implements a symbolic query processing engine for a subclass of SQL queries was presented. It could be shown that QAGen is able to generate query aware databases for complex SQL queries and that it scales linearly. Moreover, a semi-automatic test case generation framework was proposed, which provides a good starting point for building a fully-automatic DBMS testing framework.

One of the most important avenues for future work is to support more query classes in QAGen. Additionally, in order to support further applications, we also plan to extend SQP to take a set of annotated query plans as input to generate one test database that incorporates all the constraints of these annotated query plans. Alike, it is also important to study the possibility of instantiating many symbolic tuples in parallel during the data instantiation phase in order to increase the efficiency of QAGen. Another interesting future work is to extend the current test case generation framework so that it supports more coverage criteria. For example, it would be interesting if the framework can generate test cases where an operator (e.g., selection) gets a maximum partition input but returns a minimum partition output. Finally, we believe that the work of SQP can be integrated with traditional symbolic execution so as to extend program verification and test case generation techniques to support the testing of database applications as well.

For both frameworks, RQP and SQP, we have shown that they are able generate test case aware databases which satisfy complex constraints and that our prototype implementations already scale well for huge amounts of data. Consequently, we can generate test databases for many practical situations which is a basic requirement for an industrial application of both frameworks. However, both frameworks also have some limitations:

- For a better industrial acceptance it would be helpful to enhance the usability of the frameworks; e.g., by implementing graphical tools which simplify the specification of the constraints on the test databases.

- Another drawback of both frameworks is that initially one test database is generated per test case, which is very expensive and makes it difficult to manage the generated test databases for industrial applications where many thousands of test cases are necessary. Therefore, in this thesis we already sketched several solutions to tackle this problem; e.g., by merging the test databases or by mutating the test cases so that more than one test case can be executed on the same test database. However, we did not analyze this problem in detail.

- Moreover, we also have not considered the evolution of the generated test databases when the test cases that are to be executed on a database application or on a DBMS are modified. Consequently, it would also be an important avenue for future work to support the evolution of the generated test databases without having to regenerate the complete test databases in order to make both frameworks even more interesting for practical applications.

In general, we believe that this thesis is only the first steps into a new research direction.

Appendix A

# Analysis of the Approximate Overweight Subset-Sum Problem

## A.1 Correctness

As explained in the Section 17.2.2 and Figure 17.5, if our algorithm returns an answer after Line 3, the answer must be optimal. Thus, in the following, we shall assume that the algorithm proceeds after Line 3, so that $c_r \leq c$, and $\sum_{i=1}^{r-1} c_i < c \leq \sum_{i=1}^{r} c_i = p$.

Let $OPT$ denote the optimal subset sum. Suppose that $OPT$ is the sum of some subsets of $C$ that consists of $a$ values in $L$ and $b$ values in $S$, say, $\{\ell_1, \ell_2, \ldots, \ell_a\}$ and $\{s_1, s_2, \ldots, s_b\}$. Immediately, we have the following facts:

**Fact A.1** $p/2 \leq c \leq OPT \leq p$.

**Proof A.2** *Since $\sum_{i=1}^{r-1} c_i < c$ and $c_r \leq c$, we have $p \leq 2c$. On the other hand, $OPT$ is the optimal subset sum value, so $c \leq OPT \leq p$.*

**Fact A.3** $a \leq r$.

**Proof A.4** *Since the sum of the smallest $r$ values in $C$ is already at least the target sum $c$, $OPT$ cannot contain more than $r$ values. Thus, $a + b \leq r$ and so $a \leq r$.*

**Fact A.5** $a \leq 2/\epsilon$.

**Proof A.6** *Since each large value $\ell_i$ is at least $(\epsilon/2)p$, we have $\ell_i \geq (\epsilon/2)c$ as $p \geq c$ (by Fact A.1). Thus, the sum of any $2/\epsilon$ large values is at least $c$, so that $OPT$ cannot contain more than $2/\epsilon$ large values. This implies $a \leq 2/\epsilon$.*

Let $L^* = \ell_1 + \ell_2 + \ldots + \ell_a$. Let $v^* = \lceil \ell_1/d \rceil + \lceil \ell_2/d \rceil + \ldots + \lceil \ell_a/d \rceil$ be the sum of the quantized values of each $\ell_i$. Note that $v^*$ is at most $L^*/d + a$, which is at most $\lceil p/d \rceil + a \leq g$. Thus, there is a bucket $B_{v^*}$ and a value $X[v^*]$ corresponding to $v^*$. We now claim that when we execute Line 24 in Figure 17.5, the value $X[v^*]$ satisfies the condition below:

**Claim A.7** $c \leq X[v^*] \leq (1+\epsilon)OPT$.

If the claim is true, then after the execution of Line 24 in Figure 17.5, the set $B_i$ returned must have a value of at least $c$ and at most $(1+\epsilon)OPT$, so that it is a desired approximate solution to the overweight subset-sum problem. So, it remains to prove Claim A.7.

Let $L'$ be the value of $X[v^*]$ at the end of Phase 1 (Line 11-17 in Figure 17.5), so that $L'$ is the sum of the "large" numbers in $B_{v^*}$. By our choice of updating the buckets, it is easy to prove by induction that $L' \geq L^*$. (Without loss of generality, assume $\ell_1 \leq \ell_2 \leq \cdots \leq \ell_a$. Then, inductively, after we have finished processing $\ell_i$ in lines 13–17, the value $X[j]$ with $j = \lceil \ell_1/d \rceil + \lceil \ell_2/d \rceil + \cdots + \lceil \ell_i/d \rceil$ is at least $\ell_1 + \ell_2 + \cdots + \ell_i$.) On the other hand, $L'$ is at most $dv^*$, so that

$$L' \leq (\lceil \ell_1/d \rceil + \lceil \ell_2/d \rceil + \ldots + \lceil \ell_a/d \rceil)d \leq L^* + ad.$$

By Facts A.1, A.3 and A.5, we have $ad \leq (2/\epsilon)d = (\epsilon/2)p \leq \epsilon OPT$. Thus, the value of $X[v^*]$ at the end of Phase 1, which is $L'$, satisfies:

$$L^* \leq L' \leq L^* + \epsilon OPT.$$

Now, there are two cases:
**Case 1:** If $L' \geq c$, then Phase 2 (lines 18–23 in Figure 17.5) will not change the value of $X[v^*]$, so that $X[v^*] \leq L^* + \epsilon OPT \leq (1+\epsilon)OPT$.
**Case 2:** If $L' < c$, after the fine-tuning in Phase 2, the value of $X[v^*]$ must be at least $c$ (otherwise, $L'$ plus the sum of all numbers in the small set $S$ is less than $c$. However, $L' \geq L^*$, so this will contradict the fact that *OPT*, which is $L^*$ plus *some* numbers in the small set $S$, is at least $c$.) and at most $c + (\epsilon/2)p$. Thus, after Phase 2, the value of $X[v^*]$ satisfies $c \leq X[v^*] \leq (1+\epsilon)OPT$.

This completes the proof of the claim, which leads to the following theorem:

**Theorem A.8** *Suppose there exists an optimal solution for the overweight subset sum problem on a set $C$ and a target sum $c$. Suppose further that the optimal solution has sum $OPT$. Then, on given any $\epsilon$, our algorithm always returns a feasible subset of $C$ whose sum is at most $(1+\epsilon)OPT$.*

## A.2 Time and Space Complexities

A breakdown of the time complexities is as follows. Firstly, Line 1-6 and Line 8–9 is done in $O(m)$ time. Then, the value of $g$ is bounded by $1 + 4/\epsilon^2 + 2/\epsilon = O(1/\epsilon^2)$, so Line 7 is done in $O(1/\epsilon^2)$ time. In Phase 1, the loop (lines 11–17) is executed at most $m$ times, and each execution requires an update of at most $g$ values (by careful implementation with a standard trick, so that when we process $c_k \in L$, we store $B_{j,v_i}$ by a triple $(j, k, c_i)$ in Line 16 instead). Thus, Phase 1 in total takes $O(m/\epsilon^2)$ time. In Phase 2, the loop (lines 18–23) is executed $g$ times, and each execution requires $O(m)$ time. So Phase 2 in total also takes $O(m/\epsilon^2)$ time. Therefore, the time complexity of the algorithm is $O(m/\epsilon^2)$.

Next, the algorithm requires two arrays $\mathcal{B}$ and $X$. Each bucket of $\mathcal{B}$ can store up to $m$ numbers, so in total it occupies $O(gm)$ space. Each entry of $X$ stores one integer, so in total it takes $O(g)$ space. Together with the space to store $C$, the space complexity is $O(gm)$. Thus, we have the following:

**Theorem A.9** *The algorithm runs in $O(m/\epsilon^2)$ time, and requires $O(gm)$ space.*

Appendix B

# Complexity of the Group Assignment Problem

**Problem Definition:** The combinatorial problem that we call the *Group Assignment* problem is non-trivial to define. Therefore, we begin by defining the major entities of the input and then define the problem itself.

**Definition B.1 (Group Assignment Problem (GA), input)** *The input $(G, C, m)$ consists of a ground set $G$ of $n$* items *where each item represents a pre-group of the input of an aggregation operator, as well as a* constraint set *$C$ of $m$* group-count constraints *which result from the positive and negative group-count knob values (and $m$ is the output cardinality of the aggregation operator) and of an* instantiation restriction vector $\mathbf{r}$ *which results from domain constraints on the group-by attributes.*

*We first describe the ground set and the associated variable set: Each item $a \in G$ has an* item size *$s(a) \in \mathbb{N}$ and an associated $d$-dimensional* variable vector *$\mathbf{v}(a)$. A variable vector $\mathbf{v} = (v_1, \ldots, v_d)$ is taken from the Cartesian product of $d$ disjoint* variable sets *$\Sigma_1, \ldots, \Sigma_d$. Thus, the variable vector $\mathbf{v}(a)$ represents the symbolic values of the group-by attributes of the pre-group which is represented by $a$ and the size $s(a)$ represents the size of that pre-group. In case that the input is not pre-grouped w.r.t. the group-by attributes, the size is $s(a) = 1$ for all items $a \in G$.*

*The goal of the GA is to partition the items into $m$ groups $(\gamma_1, \ldots, \gamma_m)$, where $\biguplus_i \gamma_i = G$, such that each group $\gamma$ meets the associated group constraint $c(\gamma) \in C$. Each item $a \in G$ is associated with a size value $s(a)$. Each group constraint $c(\gamma_i)$ is of one of the following three types:*

$$\sum_{a\in\gamma_i} s(a) \leq \mathit{rhs}(i) \qquad \sum_{a\in\gamma_i} s(a) = \mathit{rhs}(i) \qquad \sum_{a\in\gamma_i} s(a) \geq \mathit{rhs}(i)$$

*A last part of the input is the instantiation restriction vector $\mathbf{r}$. Part of the problem is to assign values from finite sets to the variables. More precisely, for each dimension $i$,$1 \leq i \leq d$ there is a* *separate finite domain set $D_i = 1, \dots, r_i$ from which the variables in $\Sigma_i$ can be instantiated ($D_i$ can be created from the domain constraints on the $i$-th group-by attribute). The instantiation can be seen as an* instantiation mapping $\mathbf{f} : \Sigma_1 \times, \dots, \times \Sigma_d \rightarrow D_1 \times, \dots, \times D_d$ *such that $\mathbf{f}(\mathbf{v}(a))$ is the instantiated variable vector $\mathbf{v}(a)$. Thus, $\mathbf{r}$ specifies the cardinalities of these domains.*

**Definition B.2 (GA problem)** *Given an input $(G, C, m)$ to the GA problem, partition the ground set into $m$ groups $(\gamma_1, \dots, \gamma_m)$ such that the constraints given by $C$ hold and find an instantiation mapping $\mathbf{f}$ such that the following property holds:*

*Two items are in the same group if and only if they have componentwise equal instantiated variable vectors: $\exists i : a, b \in \gamma_i \Leftrightarrow \mathbf{f}(\mathbf{v}(a)) = \mathbf{f}(\mathbf{v}(b))$.*

It is not surprising that the GA-problem is NP-complete.

**Lemma B.3** *The GA-problem is strongly NP-complete for varying sizes of ground set (single and multiple group-by attributes) and constraint set, an arbitrary single constraint type and a single fixed right hand side for the constraints.*

**Proof B.4** *The problem is obviously in NP, as one can guess and verify a solution. For the reduction we reduce to the 3-partition problem, problem SP15 in [GJ90].*

*3-partition asks for a given set $A$ of $3m$ elements of sizes $\sigma(a')$ for $a' \in A$ and a bound $B$ (with $B/4 < \sigma(a') < B/2$ as well as $\sum_{a' \in A} \sigma(a') = mB$) whether $A$ can be partitioned into $m$ disjoint set $A_1, \dots, A_m$ such that for each $A_i$, $\sum_{a' \in A_i} \sigma(a') = B$. Note that each $A_i$ must therefore contain exactly three elements of $A$.*

*This problem can be formulated by the grouping part of the above problem alone: Each element $a' \in A$ maps to an item $a \in G$ with $s(a) = \sigma(a')$ and $\mathbf{v}(a) = \alpha_a$. In this transformation each item gets a separate variable. The constraint set consists of $m$ constraints that impose $\sum_{a \in \gamma_i} s(a) = B$. Note that it is possible to replace all equal constraints together by lower or greater equal constraints.*

# Bibliography

[ABMM07] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. Anatomy of the ADO.NET Entity Framework. In *SIGMOD*, 2007.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[AO94] Paul Ammann and Jeff Offutt. Using formal methods to derive test frames in category-partition testing. In *Compass*, pages 69–80, 1994.

[Bal06] Melinda-Carol Ballou. Worldwide distributed automated software quality tools 2006-2010 forecast and 2005 vendor shares. *IDC report*, Dec 2006. http://www.idc.com/getdoc.jsp?containerId=204874.

[BC05] Nicolas Bruno and Surajit Chaudhuri. Flexible database generators. In *VLDB*, pages 1097–1107, 2005.

[BCL86] José A. Blakeley, Neil Coburn, and Paul Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *VLDB*, pages 457–466, 1986.

[BCT06] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating Queries with Cardinality Constraints for DBMS Testing. *TKDE*, 2006.

[BGB05] Jesus Bisbal, Jane Grimson, and David Bell. A formal framework for database sampling. *Information & Software Technology*, 47(12):819–828, 2005.

[BGHS07] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A genetic approach for random testing of database systems. In *VLDB*, pages 1243–1251, 2007.

[Bin96] Robert V. Binder. Testing object-oriented software: A survey. *Software Testing, Verification and Reliability*, 6(3/4):125–252, 1996.

[Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BKL06a] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse Query Processing. Technical report, ETH Zurich, 2006.

[BKL06b] Carsten Binnig, Donald Kossmann, and Eric Lo. Testing database applications. In *SIGMOD*, pages 739–741, 2006.

[BKL07a] Carsten Binnig, Donald Kossmann, and Eric Lo. QAGen: Generating Query-Aware Test Databases. Technical report, ETH Zurich, 2007.

[BKL07b] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse Query Processing. In *ICDE*, pages 506–515, 2007.

[BKL08] Carsten Binnig, Donald Kossmann, and Eric Lo. Towards automatic test database generation. *Data Engineering Bulletin*, 31(1), 2008.

[BKLO07] Carsten Binnig, Donald Kossmann, Eric Lo, and Tamer Özsu. QAGen: Generating Query-Aware Test Databases. In *SIGMOD*, pages 341–352, 2007.

[BKLSB08] Carsten Binnig, Donald Kossmann, Eric Lo, and Angel Saenz-Badillos. Automatic Result Verification for the Functional Testing of a Query Language. In *ICDE*, 2008.

[CAA$^+$04] Ramkrishna Chatterjee, Gopalan Arun, Sanjay Agarwal, Ben Speckhard, and Ramesh Vasudevan. Using data versioning in database application development. In *ICSE*, pages 315–325, 2004.

[CDF$^+$04] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 2004.

[CDFP97] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatiorial design. *IEEE Trans. Software Eng.*, 23(7):437–444, 1997.

[CGMC03] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE*, pages 38–48, 2003.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[Che76] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *Proceedings of CAV*, volume 3576, pages 296–300, 2005.

[CN97] Surajit Chaudhuri and V. Narasayya. TPC-D data generation with skew., 1997. ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[CT03] María José Suárez Cabal and Javier Tuya. Improvement of test data by measuring sql statement coverage. In *STEP*, pages 234–240, 2003.

[dbM] dbMonster. http://dbmonster.kernelpanic.pl/.

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

[Elk89] C. Elkan. A decision procedure for conjunctive query disjointness. In *PODS*, 1989.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.

[GLJ01] César Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, pages 571–581, 2001.

[GMUW01] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[GOA05] Mats Grindal, Jeff Offutt, and Sten F. Andler. Combination testing strategies: a survey. *Softw. Test., Verif. Reliab.*, 15(3):167–199, 2005.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[GSE$^+$94] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD*, pages 243–252, 1994.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD*, pages 23–33, 1987.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in starburst. In *SIGMOD*, pages 377–388, 1989.

[HKK05] Florian Haftmann, Donald Kossmann, and Alexander Kreutz. Efficient regression tests for database applications. In *CIDR*, pages 95–106, 2005.

[HKL07] Florian Haftmann, Donald Kossmann, and Eric Lo. A framework for efficient regression tests on database applications. *VLDB J.*, 16(1):145–164, 2007.

[HM05] Tony Hoare and Jay Misra. Verified software: theories, tools, experiments. Vision of a grand challenge project. In *Verified Software: Theories, Tools, Experiments*, 2005.

[HN96] Joseph M. Hellerstein and Jeffrey F. Naughton. Query execution techniques for caching expensive methods. In *SIGMOD*, pages 423–434, 1996.

[HTW06] Kenneth Houkjær, Kristian Torp, and Rico Wind. Simple and realistic data generation. In *VLDB*, pages 1243–1246, 2006.

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

[IK75] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4):463–468, 1975.

[IST06] Standard glossary of terms used in Software Testing, 2006. http://www.istqb.org/downloads/glossary-1.2.pdf.

[IWL83] Tomasz Imielinski and Jr. Witold Lipski. Inverting relational expressions: a uniform and natural technique for various database problems. In *PODS*, pages 305–311, New York, NY, USA, 1983. ACM Press.

[Joh74] David S. Johnson. Fast algorithms for bin packing. *J. Comput. Syst. Sci.*, 8(3):272–314, 1974.

[Kap04] Gregory M. Kapfhammer. *Computer Science Handbook*. CRC Press, June 2004.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[Klu80] A. Klug. Calculating constraints on relational expression. *ACM Trans. Database Syst.*, 5(3):260–290, 1980.

[KMPS03] Hans Kellerer, Renata Mansini, Ulrich Pferschy, and Maria Grazia Speranza. An efficient fully polynomial approximation scheme for the subset-sum problem. *J. Comput. Syst. Sci.*, 66(2):349–370, 2003.

[KS03] Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *ESEC / SIGSOFT FSE*, pages 98–107, 2003.

[Kui02] Bart Kuijpers. Introduction to constraint databases. *SIGMOD Record*, 31(3):35–36, 2002.

[LS93] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In *VLDB*, pages 171–181, 1993.

[MAB07] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling Mappings to Bridge Applications and Databases. In *SIGMOD*, 2007.

[MF02] Sam Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, 2002.

[MR86] Heikki Mannila and Kari-Jouko Räihä. Test data for relational queries. In *PODS*, pages 217–223, 1986.

[MR89] Heikki Mannila and Kari-Jouko Räihä. Automatic generation of test data for relational queries. *J. Comput. Syst. Sci.*, 38(2):240–258, 1989.

[MS77] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley and Sons, New Jersey, USA, 1977.

[MS04] Gerome Miklau and Dan Suciu. A Formal Analysis of Information Disclosure in Data Exchange. In *SIGMOD*, pages 575–586, 2004.

[NML93] Andrea Neufeld, Guido Moerkotte, and Peter C. Lockemann. Generating consistent test data for a variable set of general consistency constraints. *VLDB J.*, 2(2):173–213, 1993.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6), 1988.

[Prz02] Bartosz Przydatek. A fast approximation algorithm for the subset-sum problem. *International Transactions in Operational Research*, 9(4):437–459, August 2002.

[PS04] Meikel Poess and John M. Stephens. Generating Thousand Benchmark Queries in Seconds. In *VLDB*, pages 1045–1053, 2004.

[RDA04] Lihua Ran, Curtis E. Dyreson, and Anneliese Amschler Andrews. Autodbt: A framework for automatic testing of web database applications. In *WISE*, pages 181–192, 2004.

[RSSS94] Kenneth A. Ross, Divesh Srivastava, Peter J. Stuckey, and S. Sudarshan. Foundations of aggregation constraints. In *Principles and Practice of Constraint Programming*, pages 193–204, 1994.

[RTI02] RTI. The economic impacts of inadequate infrastructure for software testing. May 2002.

[Slu98] Donald R. Slutz. Massive Stochastic Testing of SQL. In *VLDB*, pages 618–622, 1998.

[SP04] John M. Stephens and Meikel Poess. Mudd: a multi-dimensional data generator. In *WOSP*, pages 104–109, 2004.

[TCdlR07] Javier Tuya, María José Suárez Cabal, and Claudio de la Riva. Mutating database queries. *Information & Software Technology*, 49(4):398–417, 2007.

[TPCa] TPC-DS benchmark. http://www.tpc.org/tpds.

[TPCb] TPC-H benchmark. http://www.tpc.org/tpch.

[WE06] David Willmor and Suzanne M. Embury. An intensional approach to the specification of test cases for database applications. In *ICSE*, pages 102–111, 2006.

[WP01] Alan W. Williams and Robert L. Probert. A measure for component interaction test coverage. In *AICCSA*, pages 304–312, 2001.

[WSWZ05] Xintao Wu, Chintan Sanghvi, Yongge Wang, and Yuliang Zheng. Privacy aware data generation for testing database applications. In *IDEAS*, pages 317–326, 2005.

[yCC99] Man yee Chan and Shing-Chi Cheung. Testing database applications with sql semantics. In *CODAS*, pages 364–376, 1999.

[Zip49] G. Zipf. *Human Behaviour and the Principle of Least Effort*. 1949.

[ZXC01] Jian Zhang, Chen Xu, and S. C. Cheung. Automatic generation of database instances for white-box testing. In *COMPSAC*, pages 161–165, 2001.

# List of Figures

# List of Tables