Portland State University

PDXScholar

11-26-2008

Graphical User Interfaces as Updatable Views

James Felger Terwilliger
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Let us know how access to this document benefits you.

Recommended Citation

DISSERTATION APPROVAL

The abstract and dissertation of James Felger Terwilliger for the Doctor of Philosophy in Computer Science were presented on November 26, 2008, and accepted by the dissertation committee and the doctoral program.

COMMITTEE APPROVALS:

_____
Lois Delcambre, Chair

_____
David Maier

_____
Leonard Shapiro

_____
Andrew Black

_____
Robert Bertini
Representative of the Office of Graduate Studies

DOCTORAL PROGRAM
APPROVAL:

_____
Wu-Chi Feng, Director
Computer Science Ph.D. Program

ABSTRACT

An abstract of the dissertation of James Felger Terwilliger for the Doctor of Philosophy in Computer Science presented November 26, 2008.

Title: Graphical User Interfaces as Updatable Views

In contrast to a traditional setting where users express queries against the database schema, we assert that the semantics of data can often be understood by viewing the data in the context of the user interface (UI) of the software tool used to enter the data. That is, we believe that users will understand the data in a database by seeing the labels, drop-down menus, tool tips, help text, control contents, and juxtaposition or arrangement of controls that are built in to the user interface. Our goal is to allow domain experts with little technical skill to understand and query data.

In this dissertation, we present our GUi As View (Guava) framework and describe how we use forms-based UIs to generate a conceptual model that represents the information in the user interface. We then describe how we generate a query interface from the conceptual model. We characterize the resulting query language using a subset of relational algebra.

Since most application developers want to craft a physical database to meet desired performance needs independent of the schema used by the user interface, we subsequently present a general-purpose schema mapping tool called a *channel* that can be configured by instantiating a sequence of discrete transformations. Each transformation is an encapsulation of a physical design decision or business logic process. The channel, once configured, automatically transforms queries from our query interface into queries that address the underlying physical database, similar to a view. The channel also transforms data updates, schema updates, and constraint definitions posed against the channel's input schema into equivalent forms against the physical schema. We present formal definitions of each transformation and properties that must be true of transformations, and prove that our definitions respect the properties.

GRAPHICAL USER INTERFACES AS UPDATABLE VIEWS

by

JAMES FELGER TERWILLIGER

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
COMPUTER SCIENCE

Portland State University
2009

DEDICATION

To Allison and Kathleen, who travel the path with me

ACKNOWLEDGEMENTS

My first thanks, of course, go to my advisor Lois Delcambre. She has been everything I could want in a mentor and a friend — helping to take small rough ideas and refine them into something worthwhile, willing to listen to me ramble on through an idea only to realize I've just proven the opposite of my point, able to see my vision almost as fast as it comes into my head. She took a student with a bizarre sense of humor and an uncertain idea of what it means to do research and helped him find faith in himself. With Lois, there is no such thing as failure — only an opportunity: to learn, or to get angry, or to find a new audience, but always to make something better.

To Dave Maier, I give my apologies — no human should have to endure as many puns as I produce. I also give my thanks, for his guidance over the past five years, but most of all, for creating this family that we call Datalab. I can only hope that my work will help carry on this legacy of which we are all so proud.

My profound apologies also go to Andrew Black and Rob Bertini, members of my committee who have my thanks for having to read the many tables of formalism contained in this dissertation, and whose input have been invaluable.

Len Shapiro has a unique ability to ask piercing questions and provide logical counterexamples for what seems like hours on end, even when it becomes clear that he actually fully agrees with you — all because every opportunity to defend an idea is an opportunity to explore its boundaries. I will always be grateful for my conversations

with him.

For five years, I have had the privilege of working with an outstanding team of database researchers. To Dave Archer, Nick Rayner, Susan Price, Jin Li, Bill Howe, Sun Murthy, Vassilis Papadimos, Laura Bright, and Kristin Tufte, I give my thanks for our friendly discussions over reading group and for being a sounding board for ideas. To my good friend and cube mate Rafael de Jesus Fernandez Moctezuma, thanks for showing me that the best way forward is full speed, regardless of whether I must leave a James-shaped hole in the wall to do it.

To the students of Portland State University, thank you for letting me be your teacher. Six times have I had the privilege to work with you, and six times have I had the opportunity to learn what it is like to see the same new idea from so many different vantage points.

To Judy Logan, Jennifer Holub, Nora Mattek, Reid Keil, Chris Newcombe, and the rest of the staff at the Clinical Outcomes Research Initiative, I give my profound thanks for letting me be a part of their team and conduct research by actually performing software development on their live code. A special thanks to Judy for being part advisor, part friend, and part wake-up call if an idea was just too far out there.

Over the nearly three years since the first lines of Guava code were written, I've had the privilege of working with a number of Masters' students. Thanks to Priya Chavan, Sutheera Hengcharoen, Shiyan Tao, Akkshayaa Venkatram, Parvathy Subramanian, Raji Lakshmi, Karthika Kothapally, Supraja Samudrala, Jagriti Agrawal, and Madhura Rama for all of their hard work, and their ability to work near the cutting edge of programming tools. A special thanks goes to Jeremy Steinhauer, whose contributions to Guava are still

growing and whose talents astound all around him.

The biggest thanks of all go to my family, without whom I would have never been born. They also insisted that I reach beyond my grasp and to never be afraid to take risks. Thanks to both my grandmothers, who insisted that I keep music in my life at any expense. Thanks to both my grandfathers, who showed me what incredible things can be built using only your hands, your will, and your imagination. Thanks to my dad, who taught me that the campsite rule is not good enough — to always leave things better than what I find them. Thanks to my mom, who is equal parts whimsy and professional, and is always whichever one is necessary. Thanks to my daughter, Kathleen, through whom I get to see the joy of simple things. And most importantly, thanks to my wife, Allison, who teaches me every day what partnership is, and reminds me that every day is a new chance.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

A popular class of software system available today, both online and on the desktop, is the data-entry application. From electronic health record software to tax software to accounting software, a great deal of effort goes into developing software applications that people use to enter data into a database for later retrieval. Such software generally follows a *forms* paradigm that closely resembles the process of a user filling out data on a paper form [20, 66]. So, tax software would qualify as a data-entry application, while a program that monitors a sensor network would not. Software integrated development environments (IDEs) (e.g., Visual Studio, Eclipse, and XCode) support the forms paradigm by including graphical form-building tools, or a Graphical User Interface (GUI) library whose basic unit of development is the form with graphical widgets that resemble form controls (e.g., text boxes and check boxes).

What separates a data-entry application from other applications is that with a data-entry application, the user typically enters and views data using the same interface. For applications where some parts of the application are data-entry but others are not, we focus here on the data-entry portions.

This dissertation addresses two issues with data-entry applications. First, regardless of the usability of the forms in an application, the schema of the database of that application can be hard to understand, and thus queries are hard to write. If a user wants

to create and issue queries, there are currently three possibilities: (1) have a developer write a specially-designed, user-friendly query interface, (2) write queries using SQL, or (3) specify the query in a generic reporting application. In all three cases, detailed knowledge of the schema and specialized knowledge of the query language is required to write or specify a query. Note that when a developer creates a user-friendly query interface for domain experts, it requires developer time and expertise, as well as its own quality assurance process. Also, such a query interface lacks flexibility in adding new query targets and capabilities.

One reason why query creation can be difficult is that the schema of the database for an application may be different from the conceptual schema. The database schema may have more or fewer tables and may use a generic format, where attribute values are unpivoted to a key-attribute-value representation. The tuple {5, "Name", "Bob"} is an example of a tuple in a key-attribute-value representation, meaning the object with key value 5 has the name Bob. There are many reasons why physical database designers choose a physical schema that is different from the conceptual schema [1]. Our work is focused on providing an appropriate query interface for end-users (who are domain experts) in the presence of an arbitrarily restructured physical schema.

Application forms are designed with a particular type of user in mind: an expert in the underlying domain of the application. For example, a clinical application is designed for people who understand clinical terminology and processes. Accounting software is designed for people who understand accounting practices. Tax software is designed to have an interface that is simple enough for a general user, but with enough depth so that details can be filled in by a professional if necessary. In this dissertation, we describe a

method that exploits the usability of an application user interface to make queries easier to write.

A second issue that we address is application evolution. Changes made to an application may in turn change the requirements of that application's physical database schema (and perhaps data as well). For instance, adding new controls or new forms to an application may mean that the columns, tables, or domains of the database need to change to hold the new data. Or, the requirements of the physical schema may need to change irrespective of the application software, such as to improve the performance of queries. In both of these cases, the common solution is for a developer to write a script that makes the necessary changes to the schema and migrates data from the old schema to the new one. The scripts are written manually, so whether the database upgrade script actually changes the database in the correct way is a problem left to quality assurance.

The thesis of this dissertation is that a data-entry application serves as a view of its underlying database. The user interface of the application serves as the schema of the view, and the middleware of the application acts as the view definition. Furthermore, both data and schema are updatable through this view. This characteristic is noteworthy because determining if an arbitrary view definition is updatable to data is in general a hard problem [19], and determining if a view definition is updatable to schema is rarely if ever considered. If this thesis is true, then we can treat the user interface itself as a view schema, and construct a query interface against it that may be more user-friendly.

We further assert that the view definition derived from the application middleware can be broken down into discrete, algebraic components. If the view is capable of handling schema updates (i.e., an evolving user interface) and can be componentized (i.e.,

is capable of being incrementally changed), then it can support the evolution scenarios described above.

In this chapter, we look at some common attributes of data-entry applications. Next, we address several problems that one encounters when designing such applications, and introduce the specific research goals that we address in this dissertation. We introduce a concrete example of a forms-based data-entry application, one that will serve as a running example throughout the dissertation. Then, we introduce our research and describe how it addresses the query interface problem and the application development and evolution problem. We close the chapter with an outline of the rest of the dissertation.

1.1 DATA-ENTRY APPLICATIONS

Our experience has been that data-entry applications typically have a number of characteristics in common. A survey of a number of business applications both currently available and from our collective work experiences revealed the following characteristics. First, they tend to use a relational database to store their data. Some data-entry applications may use alternative data models such as XML for communication between clients or between tiers of software, but most data-entry applications use a Relational Database Management System (DBMS) for persistent storage of data. Other attributes that we have noted that data-entry applications have in common include:

The user interface of a data-entry application is a conceptual model. Developers may not necessarily create user interfaces with this idea in mind, but the forms of an application, along with the way in which the forms are linked together, are precisely the model of the data that is presented to the user. In our work, we demonstrate

that it is straightforward to extract the conceptual model that is implicit in the UI. This conceptual model is a direct result of various forms in an application and the way they interact. Every form in the application that displays data or allows a user to enter data corresponds to an entity, and each field on a form represents an attribute of that entity. The relationships between forms (where clicking on a button or a link launches another form and hands focus or control to the new form) are represented as relationships among the underlying entities of those forms.

The user interface of a data-entry application is the only conceptual model the user sees. There may be other conceptual models that a developer may create or use during the process of developing an application. The developer may create an Entity-Relationship diagram as a model of the data that belongs in the database. In addition, the developer may create a UML diagram or other object-modeling artifact as a model for underlying classes. However, none of these models is likely to be visible to or understood by the user who is actually using the application; the user's view of the data in the application is heavily influenced by how the data appears in the user interface.

The conceptual model of the user interface of a data-entry application may bear little or no resemblance to the schema of the underlying database. The physical database schema is typically optimized for either space or retrieval time, or organized for easy maintenance and extensibility, and thus may be quite different from the structure of the user interface. Data may be coded, structured in a generic fashion, partitioned, or merged in any way that the physical database designer chooses. These database-oriented decisions are hidden from the user. This situation is an instance of *logical data independence*, where the user is presented with a view of data that is insulated from the

physical (relational) schema, as well as potential changes to that schema. Traditional relational views are the classical example of logical data independence [14, 78].

A typical data-entry application serves as an updatable view of its persistent data. When a user enters data into a user interface, there are some implicit assumptions that the user makes. First, if a user enters information into a form and then brings up the data for the same entity in the same form again, the same data should appear again; in other words, the user makes an implicit assumption that any data entered in the form becomes persistent. Second, if a user enters information into a form, then brings up a different, unrelated form, there should not be any unexpected side effects. For instance, if someone enters a new patient into a software system, then the list of doctors in the same system should not change, unless there is an explicit relationship in the user interface (such as a field asking if the patient is also a doctor). In this way, a GUI acts just like a view of the data in the physical database that is updatable — where one may issue updates against the view as if it were itself a physical database instance, even though it is only virtual.

It is conceivable that a software application would allow update side effects that do not follow some sort of logical pattern, such as introducing random data elements in other fields. In this dissertation, we assume that all side effects of updates are the result of deterministic, logical and user-understood semantic relationships.

One of the benefits of a view definition over a data source (i.e., a sequence of operations defined to draw data from the source) is that one can treat the view as if it were a data source for the purpose of queries, despite the fact that it is only virtual. In general terms, the view-update problem is a decision question: For a given view, determine if

it is possible to translate updates against the extent of the view into updates against the original data source in such a way that, when the view definition is re-executed, the new view extent reflects the update. Research has been done to determine what kinds of operations can be used in view definitions and still have the view be updatable [5, 8, 9, 49]; however, the vast majority of view definitions are not updatable because there is no way to identify a way to update the base source to effect the correct updated view extent [19]. In short, if our thesis holds, a user interface offers by default and by necessity an answer to the classic view-update problem.

1.2 RESEARCH GOALS AND CONTRIBUTIONS

In this dissertation, we address a number of different research questions motivated by forms-based data-entry applications, though several of our research contributions have implications beyond this class of application. In each of the following subsections, we describe an opportunity to assist developers when constructing forms applications, or users when using forms applications. For each opportunity, we present the specific research goals that we address in later chapters, and a description of the contribution that we make to address each goal.

1.2.1 Query Interfaces

As mentioned in Section 1.1, the user interface for a data capture tool is typically designed to be easy to use by users who are knowledgeable in the application domain. For example, a great deal of effort goes into making sure that the user interface of medical software can be understood by clinicians and other medical staff. It is common for a

clinical software user to be well-versed in clinical terminology and medical procedures, but not have the skill to use SQL or the experience to understand the semantics of a complex query.

In addition to being a popular design paradigm for applications, a form is also a popular design paradigm for query interfaces. Software applications and web interfaces often contain "search forms" that allow a user to specify some simple search parameters and then build a potentially complex query and execute it behind the scenes. Natural Forms Query Language (NFQL) [22] is a language for constructing forms that act as a query interface over a database. Recently, Jayapandian and Jagadish [38] created tools that generate forms-based query interfaces based on the schema of the database, and the profiles of queries that have been executed.

All of these form-based query interfaces share two characteristics in common, besides their form-based nature. First, they are all intended to hide some complex query operations from the user. In most cases, the complex query operation in question is the join operator. Second, these techniques typically expose only a subset of the data available in a database. For instance, an application may have a search form for building complex queries to find medical providers in a database, but that form was custom-built by a developer anticipating a certain class of query; one cannot then use the same query form to search for patients. In Jayapandian's research, one of the metrics that they use to evaluate their generated search forms is coverage of schema elements (tables and foreign keys) because they generate forms only for the most frequently accessed tables and the most frequently issued queries (to keep the number of generated forms low).

In this dissertation, we describe how to create a query interface that does not require

users to specify complex operators such as joins and that is therefore potentially easier to use. Unlike existing form-based query interfaces, our query interface exposes all the data that is available in the user interface of the application. The usability of our query interface comes at the cost of limitation in the kinds of queries that it can express relative to SQL. This tradeoff between usability and functionality is common in query interfaces and is present in both NFQL and Jayapandian's automatic query forms (whose query capabilities are comparable to ours).

Research Goal 1: Develop an automatic method for constructing query interfaces that employs the conceptual model inherent in the user interface and draws usability features from the user interface, and is complete with respect to that conceptual model. We have developed such a method for application user interfaces written using a graphical widget library that we created by extending an existing popular widget library, namely the Windows Forms library that comes with the Microsoft Visual Studio IDE. By creating a query interface from the user interface of an application, we preserve and then exploit the contextual clues to the meaning of individual data elements in the user interface, such as the help text and tool-tip text of each form widget that provides guidance to the user of an application.

1.2.2 Application Middleware

Any system that satisfies Research Goal 1 will be able to generate an application-specific query interface. The challenge for such system is how to translate the queries from said query interface into queries that address the schema of the application's physical database. We require that the query interface and application UI communicate with the

database using the same mechanisms; if they do not, there can be no guarantee that the query interface is retrieving the same data that is available in the UI. For this part of the research, we create a tool that can serve as that communication component — often called *middleware* — and that is flexible enough to handle arbitrary queries sent from the query interface.

Between an application and its database, there are two database schemas: the schema that the application needs based on its user interface and business logic (which we call the *natural schema*), and the schema that the database actually has (which we call the *physical schema*). As mentioned above, the structure of and data in the physical schema may bear little resemblance to the structure and data in the natural schema. There are two fundamental reasons why this difference may occur. The first is physical database design. A database developer may construct a physical schema that optimizes space utilization or disk accesses based on the developer's understanding of what kinds of queries and updates may be issued against the database.

Some of the physical design decisions that a developer makes involves choosing physical structures, like indexes and views. Additionally, the design decisions will often involve one or more of the following structural transformations:

- The developer may choose to distribute the columns or rows of a table into several tables. This transformation, called *partitioning*, is typically used if the developer notices that many queries being issued against the database refer to only part of a table. For instance, if 99 percent of all queries issued that involve table T refer only to three columns in T, it makes sense to partition T into two tables: one table with the three commonly-used columns, and one table with the rest of the

columns.

- Conversely, the developer may choose to combine the columns of several tables together using joins, or the rows of several tables together using unions. This transformation, called *merging*, is typically used if the developer notices that tables frequently appear in queries together. Thus, pre-computed joins or unions are stored at the physical level.

- The developer may choose to encode data into a smaller or different representation. For instance, data that represents long strings of text, such as "patient unconscious after procedure", may be reduced to a small integer, such as "0", representing the index of the data item in a radio or drop-down list. For another example, the database may store a single integer that is the bitmask of the values of many Boolean values, such as the answers to a collection of checkboxes, which would normally be represented in individual columns. For this physical design choice, the developer writes an invertible function that can be applied to each row's data values to encode them on the way into the database and whose inverse can be applied to values retrieved from the database. We call this transformation *function application*.

- The developer may choose to store data in column-per-attribute rows when the data is originally represented as key-attribute-value triples, sometimes called a generic format. This transformation is called *pivoting* data; pivoting refers to the process of taking data out of a generic layout (where each row stores a key, an attribute, and a single data value) and into the more familiar structure where

attribute names are part of the schema (columns) rather than part of the data. This transformation is frequently used in data warehousing, but can also be used in physical design. For instance, queries that involve conditions on multiple columns run very slowly on data in a generic format because evaluating the query involves many self-joins; pivoting the data effectively pre-computes these joins.

- The developer may choose to store data in "generic" key-attribute-value triples when the data is originally stored in column-per-attribute rows. Since this process is the opposite of pivoting, this transformation is called *unpivoting*. One reason why a developer may prefer to unpivot a table is to eliminate the need to store null values in the database; if most of the values held in non-key columns of a table are null, storing the table in a generic form may save a significant amount of space. Another reason why a developer may prefer a generic triple store is extensibility; one can add new attributes just by adding new rows to the unpivoted table rather than having to add any new columns to the schema, thus avoiding having to change the schema.

A second reason why a physical database may have different data or structure than the application's user interface is because of the transformations included in the business logic. The business rules in a software application may introduce new data or alter data as it is sent to the database. For instance, a company's auditing policy may require that all data be stamped with the name of the user that entered it.

For applications whose physical schema is different from its natural schema, or whose physical schema contains additional information beyond what is present in its

natural schema, a middleware layer typically handles communication between the application and database. Middleware comes in many shapes and sizes, from simple developer-provided, application-specific class libraries that generate SQL statements, to robust commercially-available packages such as object-relational mappers (ORMs) [33, 39, 49]. Application-specific class libraries suffer the same problem that application-specific query interfaces do: They are specific to the queries and updates that the developer anticipates. Because they are hard-coded, though, they accommodate both the physical design decisions and business logic. ORMs and other middleware tools have generic translation capabilities that translate queries and updates against some input schema (similar to our concept of a natural schema) and transforms those statements into equivalent statements over the physical schema. ORMs are typically limited in the kinds of transformations that they can accomplish. For instance, developers cannot use any currently available ORM to establish a relationship between two schemas that have a pivot or unpivot relationship between them. As a result, tools that use ORMs as application middleware frequently also require some sort of hard-coded transformation as well.

Establishing a relationship between an application's user interface and its database schema using middleware is a special case of a more general problem called *schema mapping*. There are two primary activities involved in schema mapping: identifying items in common between two schemas, and providing a mechanism for translating queries and updates expressed against one of the schemas into equivalent queries and updates against the other schema. Mappings can be expressed all at once using correspondences [49, 72] or incrementally using discrete transformations [9, 30, 47, 80].

Mappings can also be uni-directional [57] or bi-directional [49] in their ability to transform queries or updates. There are a wide variety of techniques and approaches reported in the database literature dedicated to representing, creating, or deriving relationships between schemas. Information integration, view derivation, data model conversion, and many other major areas of database research all relate to schema mapping.

Application middleware is a special case of schema mapping because it has specific requirements on the mapping between application schema and physical schema. As already mentioned, the mapping between schemas may include structural transformations like pivots and business-logic-like transformations. The mapping must also be guaranteed to be information preserving, that is to say, the mapping must be lossless (no information originating from the application is lost) and free of unmotivated side effects (no information appears in the application when it is not expected).

We go into depth in our analysis of existing mapping languages and their capabilities in Chapter 3; the conclusion of our analysis is that no existing mapping language can express all of the physical design transformations listed above, as well as business logic rules, without escaping from the mapping system and adding hard-coded logic. Since the application developer must issue data manipulation language (DML) statements — as well as queries — against the physical schema, we see an opportunity here to extend the capabilities of the mapping from the conceptual model of the user interface to the physical schema to handle transformation of DML statements.

Research Goal 2: Develop an information-preserving schema mapping language that is expressive enough to handle physical design decisions, and whose operational capabilities include transforming queries, data updates, and schema

updates. We define a general-purpose mapping language that meets the special requirement of information preservation. Our language supports pivoting, unpivoting, function application, partitioning, and merging. The mapping language is also able to accommodate schema updates so that we can also handle the major requirement in the next section, schema evolution. The developer can use a mapping to construct a physical database without requiring any changes to the application code, and without changing the user experience. The mapping language thus supports physical database independence. Since our mapping language allows developers to write code against the natural schema derived from the UI, we believe that our mapping language will simplify application development by reducing the cognitive load on the developer. Because the mapping language is general-purpose, it can operate on any query expressed in extended relational algebra issued against the natural schema, not just the limited query language supported by the query interface that we generate from the user interface (as described in Research Goal 1).

Research Goal 3: Demonstrate that our mapping language is extensible. Currently, no mapping language in the literature is expressive enough to accommodate business logic. For instance, mappings may describe the fact that a column was added or dropped [17], but not be expressive enough to describe *why* the column was added or dropped, e.g., if a dropped column was redundant and can be reconstructed, or that the added column is always populated with environment data. Because our mapping language needs to be able to describe the relationship between an application and its database fully, we must allow a developer to add new constructs to the mapping language as needs arise, e.g., to represent data added by business logic.

Research Goal 4: Build a formal framework within which we can prove properties of our mapping language. To ensure that the query interface from Section 1.2.1 always returns correct data, we must prove that our mapping language is in fact information preserving. We also prove equivalences and other formal properties of our mapping language, such as commutativity and invertibility, to support optimization.

1.2.3 Application Evolution

When a new version of a database-backed application is released, the new version is almost invariably accompanied by a script that must be run against the database to update its data and schema to work with the new version of the application. In this section, we discuss how this script is created, and how we make creating this script easier.

In an ideal world, whenever a developer makes a change to an application, the appropriate changes required for the database to be compatible with the new version of the application — and to update the data already present in the database to be compatible with the new application version — would be generated and processed. And, conversely, whenever a database professional makes a change to the database schema, the queries and updates contained in the application code would be automatically modified if necessary to address the new schema. For example, in an IDE, one would like to be able to select a database column and rename it (an example of a *refactoring* [25]) and have the environment automatically detect that all other references to that column in queries and updates in the application need to be renamed as well.

Some modern IDE's allow developers to view or even edit database schemas in the same environment as application code [54]. However, if a field in a database is renamed,

no tools or IDE's exist yet that can automatically alter every query in an application to reference the new column name. Recent research tries to recognize which application queries or updates need to change if a database evolves in specific ways [46], but it has a high rate of false positives, and cannot yet describe *how* the queries or updates should actually change.

From the application perspective, if a developer were to add a new control to a form in the application's user interface, the database will not automatically create a new column in an appropriate table to hold the control's data. Changes to the application's natural schema do not automatically propagate to the physical schema. Therefore, developers must manually keep track of these changes and manually propagate them, which is a potentially error-prone process.

Recent research improves the relationship between application development and database development. For instance, Chaudhuri et al. [12] have created a framework that can correlate system log entries from an application with log entries from a database, so that a developer can associate individual function calls and errors in an application with queries, updates, and errors in a database. The paper associated with this research acknowledges the disconnect between applications and databases, and how little research there is in unifying database development with application development.

On the database side of application evolution, there is a great deal of research investigating the evolution of database schemas [63]. This research often intersects with the schema mapping research mentioned in Section 1.2.2, since one way to think about schema evolution is to consider two versions of the same database schema separately, and to create a mapping between them [6, 17]. However, very little of this research has

made its way into available products that developers can use.

Some tools allow a developer to specify the way that they would like to change the schema of a database and then generate a corresponding upgrade script of DDL statements (e.g., Add Column or Drop Table) that can then be deployed to installations of that database [54]. These tools do not typically support atomic data-level transformations (such as function application) or higher-level transformations that involve moving data (such as partitioning or merging). The developer still needs to manually update any applications that connect to the evolved database by changing any hard-coded queries and updates in the code, or by altering any programs that automatically generate queries and updates.

Most tools that have a schema evolution component follow the model of Ruby on Rails [67] or SQL Alchemy [73]. In these tools, schema evolution for an application is encapsulated into discrete modules called *migrations*; each migration represents all of the actions necessary to migrate a database schema and instance from one version of the application to another adjacent version (one revision higher or lower). One can compose migrations to upgrade or downgrade the version of a database to match the application. Ruby on Rails and SQL Alchemy both apply migrations automatically at runtime to ensure that the application and database have version parity. However, the construction of a migration is not automatic; the DDL or DML statements that constitute a migration must still be provided by the developer, who must create and test them to ensure that they match any changes at the application level.

Research Goal 5: Develop a scheme for handling application evolution, both of the user interface of the application and of the mapping between the user interface

and database, such that the resulting database upgrade script is automatically generated. In our research, we create a unified framework that can handle both evolution of application components and database components. We treat the database as a function of the user interface and the mapping. In other words, rather than alter a database directly, a developer using our tools alters the application's database mapping, which results in automatically generated database upgrade statements in DDL and DML corresponding to the mapping changes. Changes to the UI result in generated database statements as well, meaning that the developer need not manually generate upgrade scripts between versions of an application.

1.3   CASE STUDY

This work was originally motivated by our work with software developed at the Clinical Outcomes Research Initiative (CORI) [13]. CORI seeks to improve the practice of endoscopy by conducting retrospective studies on de-identified patient data (i.e., endoscopy reports). To this end, CORI develops and distributes a software reporting tool that allows the clinician to enter data that describes endoscopic procedures and then generates endoscopy reports suitable for inclusion in the patient medical record. Endoscopy reports from nearly 70 sites across the US are being compiled by CORI in a data warehouse on an ongoing basis. Figure 1.1 shows a screenshot of a form from the CORI user interface.

CORI supports a number of data analysts who conduct various retrospective studies. A retrospective study is an attempt to study data that has already been collected for other purposes — in this case, the reports that have been collected through the CORI

Figure 1.1: A screenshot of a form from the CORI software, version 4.0.23

software by physicians over the course of patient care. Each study requires that the analyst select an appropriate subset of the reports in the warehouse, classify the source data into categories of interest in the study, as appropriate, and then hand off the selected data, post-classification, for analysis in a statistical package.

We use CORI as our primary case study throughout this dissertation because it exemplifies the issues and opportunities that we introduced in Section 1.2. The data analysts' job is to construct valid, correct queries against a database schema that is not easy to understand. CORI data analysts can benefit greatly from a query interface derived from the original user interface because they have experience with clinical data and terminology. The analysts need a comprehensive understanding of the contents of the data warehouse, but they suffer from the following problems that affect the efficiency of their work and the reliability of their studies:

- The primary data repository has a generic data structure and encoded data values, which makes querying difficult, if not impossible,

- both the data and the schema of the data warehouse are arcane because the explanation of how the data is encoded and how it has been restructured is not available to the analyst in a manner that they can understand, and

- fixing either of the previous problems requires time from and communication with the development staff. Furthermore, having the developers write queries puts decisions about query creation that affect the semantics of the data in the hands of people who do not themselves run studies and may not appreciate how an apparently insignificant change in a query may affect the semantics of the associated
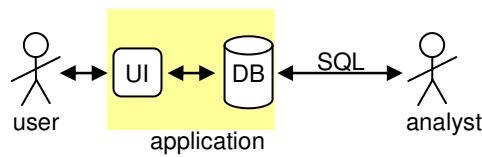
(a) Traditional approach: analyst writes queries against (physical) DB
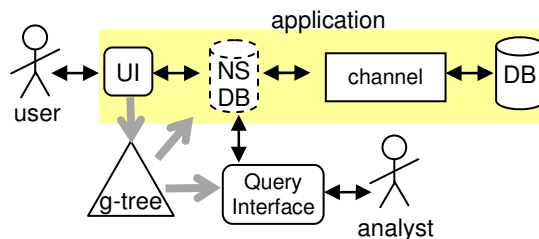
(b) GUAVA: The g-tree is generated from UI, then natural schema and query interface are generated from g-tree. Analyst uses query interface.

Figure 1.2: The GUi As View (Guava) software engineering framework study.

The relationship between the CORI application and its database exhibits many of the features described in Section 1.2.2 as physical design decisions and business logic. The database has been unpivoted, merged, and encoded for space efficiency. It has also been augmented based on the business logic to support security and reporting. Finally, the CORI development staff is also in the process of upgrading to a new version of their application and are encountering the prospect of manual database upgrades. Therefore, CORI serves as a useful case study for each of the research goals mentioned in the previous section.

1.4   GUAVA: GUI-AS-VIEW

This section introduces our GUi As View (Guava) framework, the primary contribution of this dissertation. Guava can be thought of as a tool to support user-interface-centric

software design. Given a description of a user interface, Guava generates a number of other artifacts automatically that are usually created manually by developers.

As opposed to the typical scenario such as that in CORI (Figure 1.2(a)) where users must either use SQL or a custom, separately-designed query interface to access the data, the query interface in Guava (Figure 1.2(b)) is automatically generated from the user interface. First, the complete structure of the user interface is represented in a conceptual model — a hierarchical structure called a Guava-tree (*g-tree*). Guava automatically generates a g-tree from the user interface controls based on our extensions to an integrated development environment. Next, Guava translates a g-tree into a simple relational table structure in what we call a *natural schema*. Finally, a database designer can transform the natural schema into the underlying physical database schema by instantiating the database transformations that comprise our *channel* mapping language. The channel is a flexible, information-preserving mapping for relational schemas that transforms the natural schema into the desired physical schema (at DB design time) and transforms simple queries from the application-UI-based query interface (as well as data and schema update statements) from the natural schema to the physical schema (at run time). The channel supports the Guava query interface, but also presents an interface that allows the application developer to write queries and DML statements directly against the natural schema as well. In Chapter 2, we will describe the structure shown in Figure 1.2(b) in more detail.

In Guava, users are presented with a query interface that leverages all of the effort put into creating a good user interface. Because the query interface accesses data through the same mechanisms as the user interface, it is guaranteed to respect the semantics of

the UI. Query results may be displayed in a number of ways, including a simple table that can then be loaded into other tools such as statistical analysis packages, or a mockup of the user interface showing the data in-place. The key component of the Guava architecture is the channel, without which the Guava query interface could not send queries to the physical schema. However, one can use channels outside the context of Guava as a general-purpose relational schema mapping tool.

Collectively, the natural schema and the Guava query interface serve to satisfy Research Goal 1. The channel has operational characteristics and expressive power sufficient to satisfy Research Goal 2, and is extensible, thus satisfying Research Goal 3. Channel transformations are provably information-preserving, i.e., satisfying the formal framework set out by Research Goal 4. Because a channel instance can accommodate updates to the schema of the natural schema, thus supporting incremental evolution of the natural schema and the components that generated it, the channel serves to satisfy Research Goal 5.

1.5 OUTLINE

The rest of this dissertation proceeds as follows:

Chapter 2 addresses Research Goal 1 by going into detail on how Guava produces a query interface that closely resembles the look and feel (and semantics) of a user interface. We define and formalize our intermediate data structure called a *g-tree*, describe how it is generated, and show how to take a g-tree and produce a default (natural) relational schema for the application.

In Chapter 3, we address Research Goal 2 by defining and formalizing the channel

artifact introduced in Section 1.4. A channel comprises a list of discrete transformations, each of which represents a physical design decision on the part of a database developer. We define seven transformations that encapsulate common physical design decisions, and formalize how each transformation acts on statements posed against the channel's input schema. Also, we introduce algebraic relationships between transformations that define an equivalence relation among channels.

Chapter 4 addresses Research Goal 3 by extending the channel transformation language in three different ways. First, we generalize the definitions of three of the transformations so they are more expressive. Second, we introduce a new class of transformation that corresponds to business-logic decisions rather than physical-design decisions. Finally, we introduce a class of transformation that can express relationships between schema elements in a relational schema and eliminate redundancy that may exist as a result.

Chapter 5 addresses Research Goal 4 by formally proving the properties that were introduced in Chapters 3 and 4. We introduce the techniques that are required to prove the correctness of our transformation definitions; we then prove the correctness of representative samples of the transformation definitions from previous chapters.

Chapter 6 addresses Research Goal 5 — the problem of schema and program evolution — by breaking up the problem into two parts: changes to the user interface and changes to the channel. We consider each class of changes separately, providing a solution for each.

Discussions of related work appear in Chapters 2, 3, 4, and 6 in related work sections specifically for the material introduced in that chapter. We also describe the state of our

prototype implementation and identify implementation issues that we found noteworthy. We use CORI as a case study throughout the chapters to demonstrate the efficacy of our tools; we also point out examples from other applications in Chapters 2 and 3.

Chapter 7 provides a summary of the contributions of our research and describes several avenues for further research opportunities that extend our research on Guava.

Chapter 2

CREATING A RELATIONAL SCHEMA AND A QUERY INTERFACE FROM A USER INTERFACE

Many software applications that are designed to capture data use forms as a visual metaphor. Form-based data entry is a familiar paradigm because of the ubiquitous nature of paper forms in the real world. The forms metaphor is also well-known and well-studied in software engineering [20], and serves as the foundation of every major GUI widget programming library (e.g. Swing, Windows Forms, and Motif). This chapter extends the state of the practice by demonstrating how to use forms as a visual metaphor for queries as well. We do so by first articulating the conceptual model that is inherent in the forms of the user interface. We then use the user interface of the software tool that creates the data as a guide to creating a query interface that closely resembles the original UI. Finally, we describe how to express queries using our interface and characterize the semantics of those queries.

In this chapter, we make the following research contributions:

- We demonstrate that it is possible to create a query interface from the user interface, and evaluate its expressive power.

- We show a way to derive a complete relational schema from a forms-based user interface (complete in the sense that all data in the user interface can be found in

the schema), against which the query interface expresses its queries.

- Characterize what restrictions, if any, exist on the kinds of user interfaces that one can translate into query interfaces.

- Evaluate the applicability of these techniques through two case studies.

Figure 2.1 demonstrates how the client-side components of a UI interact with a database, both in the dominant development paradigm (a) and with our proposed framework (b). The status quo, without using Guava, comprises four components: the user interface, a business logic layer, middleware, and the physical database. A user enters data into the user interface, which is then processed by business logic (to perform functions such as validation). The data then flows into a middleware layer (which is sometimes combined with the business logic), where it is further transformed to match the schema of the physical database. Such transformation may be substantial, since the tables in the physical database may bear little resemblance to the structure of the data as it entered the middleware. When the user requests information through the user interface, a similar process happens: the middleware retrieves data from the database, transforms it, and sends it to the user's screen. Applications typically support the so-called CRUD operations: create data (C), retrieve data via simple pre-defined queries (R), update data (U), and delete data (D).

There is no query interface present in the application stack shown in Figure 2.1(a). If users want to run queries over the data that is captured by this software, there are generally only two options available: have a developer write a special query interface, or use one of a variety of tools (such as visual query builders, report generators, or
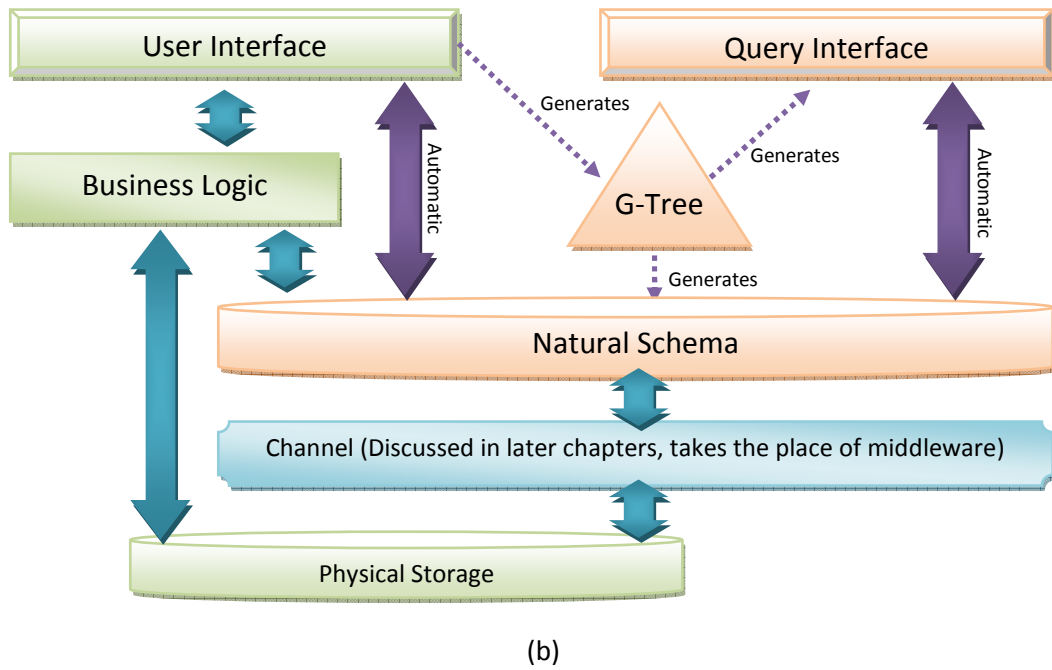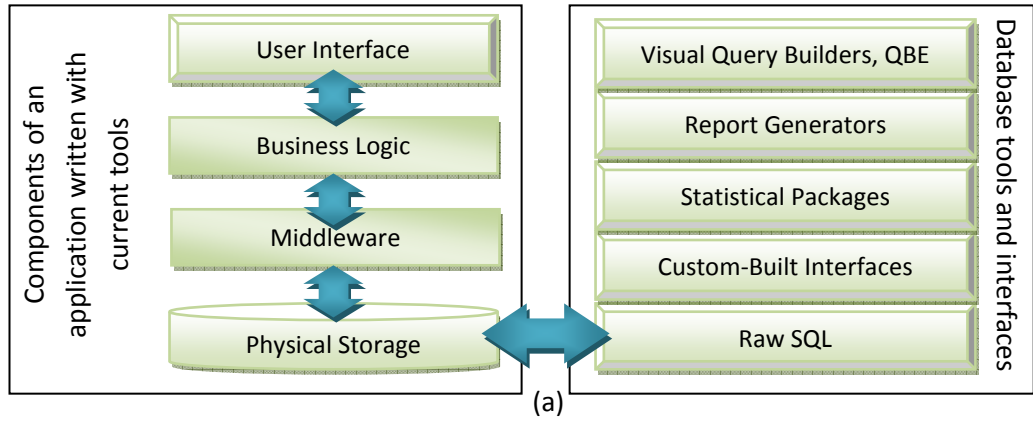
Figure 2.1: An overview of how components in an application written using currently available tools (a), compared to how the UI components of Guava interact and are generated (b)

raw SQL) to express queries against the database schema. The first option requires developer time and effort, and results in a query interface that is statically coded to meet the expected query workload from the users and the schema of the data, both of which can change. The second option requires the user to have both mastery of a query language and a comprehensive knowledge semantics of the data based on the structure and names that appear in the schema (perhaps with additional documentation, e.g., in a data dictionary).

The middleware layer is what prevents the query interface from participating in the application stack. An application's middleware is often manually coded due to the complexity of the transformation required to conform data from the user interface to match the physical schema. Tools exist that can automatically generate middleware (such as object-relational mappers, discussed in Chapter 3), but the developer typically must write code as well, to enforce or uphold semantics or perform transformations on the data that are specific to the application. Manually coded middleware generally does not support translating arbitrary queries; middleware typically only supports CRUD operations. Automatically-generated middleware tools sometimes provide an interface that allows a developer to issue a wide range of object-oriented or SQL-like queries and translate them into queries against the database [33]. However, the casual user cannot access such features without the aid of a developer-written query interface. The query translation features also cannot generally work with additional application-specific data transformations hard-coded by the developer.

Even if a developer manually implements a query-translation feature in middleware, and the translation is proven to be correct, the user still needs to understand SQL and

the relational schema. Plus, there is an additional problem; we now need to know what the relational schema is of the input to the middleware, not just the output (the physical schema). The input schema may be implicit and undocumented, since it only exists in code and lacks the robust tool support available for schema discovery in databases.

Guava takes the user interface and generates an artifact called a g-tree (Section 2.1); Guava then in turn generates two additional artifacts from the g-tree, a query interface and a natural schema (Section 2.2). The natural schema is a relational schema that resembles the structure of the UI rather than the physical schema. The query interface also resembles the original UI of the application, presenting the same kinds of contextual information to the user that, in the UI, assisted the user in determining the meaning of data elements (e.g., leading text, help text, or proximity to other controls). The dotted arrows in the figure indicate the generation of artifacts. The large solid arrows in the diagram indicate the flow of data between components.

The intent of Guava is for the user interface to perform CRUD operations through the natural schema, thereby accessing the physical database through the same mechanism as the query interface. Thus, Guava ensures that the query interface is faithful to the data-access semantics of the original software, which in turn ensures that queries written in the Guava query interface return the expected results. Whether the application employs additional business logic is immaterial, so long as any logic that transforms the data does so between the natural schema and physical database, rather than between the application and the natural schema. Chapter 4 discusses how to encode business logic in such a way. The natural schema also serves as an API to the database, as a developer can issue queries or updates against it. If necessary, the application can ignore the Guava

components entirely and connect directly to the database. However, any interactions that bypass the Guava components are not reflected in the query interface, since the structure of the underlying database is not represented in the g-tree for the application.

Tools that can automatically generate middleware present an object or relational interface with which the application can communicate, but the developer must still write the glue code between the application and the middleware that, for instance, displays data on a form or sends data back to the interface when the 'OK' button is clicked. This functionality is called a *controller*, and is part of a design pattern called Model-View-Controller (MVC) [26]. There are software tools that build controllers automatically or semi-automatically [67, 73], but rarely are they integrated with the same tool that generates the middleware [33]. Guava generates these connections between the user interface and the natural schema as well.

We begin the rest of the chapter by introducing and formalizing the g-tree and natural schema (Section 2.1). We then show how to generate a query interface from the g-tree, and describe the kinds of queries that the interface can produce (Section 2.2). Next, we present notes about our prototype implementation of these tools (Section 2.3), followed by two case studies: an attempt to re-create part of a commercially-available software product using Guava (Section 2.4) and an alternative use of Guava that allows external applications to use Guava forms to uniquely identify data in a database (Section 2.5). The chapter concludes with an analysis of related work and a summary (Sections 2.6 and 2.7).

Figure 2.2: A simple forms-based application with two forms. The second form provides additional details for the same person represented by the first form, and the second form appears by clicking the first form's 'Details' button

2.1   G-TREES AND NATURAL SCHEMAS

With Guava, we seek to exploit the hierarchical nature of forms-based user interfaces to provide a simple representation of the user interface's information. Figure 2.2 shows an example user interface; the widgets on each screen form a hierarchy based on the "contains" relationship, and the forms of an application are structured as a hierarchy because each form is launched from an event on another, save for the form that appears at application launch that serves as the root. In Figure 2.2, the form on the right is launched by the details button of the form on the left.

The rest of this section explains the steps that Guava uses to generate a g-tree. A Guava-tree (*g-tree*) represents the information present on a user interface, including the relationships between forms. Also of interest are the *context elements* for the widgets, such as the widget's type (e.g., text box or checkbox), its default value, and its text. A

widget's text may be simple to find for checkboxes and group boxes where the text is simply part of the widget, but is often harder to find for text boxes and drop-down lists where the text is actually in an adjacent label. These context elements are informative for anyone using the application and for users that want to query the data.

Formally, a g-tree is a rooted directed tree with a set of nodes N and a set of directed edges E such that:

- Each $n \in N$ is labeled with one of the values *Entity*, *Attribute*, *Container*, or *Control*.

- Each $n \in N$ has a property *Name* whose value is unique in the tree.

- Each $n \in N$ has a partial function $h : String \rightarrow String$ that associates context element names with the value of that context element for the control.

- Each $e \in E$ is labeled with one of the values *Contains*, *Single-launch*, or *Multiple-launch*.

The node labels refer to the kind of control the node represents:

- A node marked with the value Entity (also called an *entity node*) refers to a form.

- A node marked with the value Attribute (also called an *attribute node*) refers to a graphical widget on a form that holds data that is saved to or retrieved from the database.

- A node marked with the value Container (also called an *container node*) refers to a graphical widget that is not itself a form and does not present data, but contains other form widgets.

- A node marked with the value Control refers to any graphical widget that cannot be categorized as one of the above.

The edge labels describe the relationships that can exist between two widgets. A "Contains" edge from *A* to *B* indicates that *B* represents a widget that is spatially contained by *A*'s widget. "Single-launch" means that the parent node's control launches the child's control, e.g., by clicking or selecting an item, and that the relationship between the child form's data and the form containing the parent is one-to-one. A common example of the single-launch relationship is shown in Figure 2.2, where the form on the right is just a details window for the form on the left. A "Multiple-launch" relationship is similar to the "Single-launch" relationship, except that the relationship between the two forms is one-to-many. One example of a multiple-launch relationship is the relationship between a drop-down list that holds the times and dates of a person's appointments and the window with the details of that appointment that is displayed when an appointment is selected from the drop-down list. The relationship is multiple-launch because a person can have any number of appointments.

To illustrate the function *h* in action, consider the check box on the first form in Figure 2.2. It has several obvious context elements, such as control text, default value, size, and location. So, for the g-tree node associated with that check box, the function *h* is defined as follows:

- *h*(*"Control Text"*) = "Procedure Completed"

- *h*(*"Default Value"*) = "False"

- *h*(*"Size"*) = "(100, 10)"

- *h*(*"Location"*) = *"*(10, 90)*"*

Each type of control supports its own set of context elements. Some context elements, such as size and location, will exist for all control types. Others, such as text length, will be supported only by certain control types (in this case, text boxes). Context elements include both the elements that have a human-readable description of data (e.g., control text, help text, and domain values) and elements that are not necessarily useful to a human user, but necessary for reconstructing the query interface at a later time (e.g., control size and location, to describe where those elements will appear in the interface).

In addition, every attribute node *a* in a g-tree has a domain, denoted as *Domain*(*a*). The domain of *a* can be one of the following:

- Any subset of one of the standard atomic data type domains, including Boolean, Integer, Real, and String.

- A reference to an entity node *e*, coupled with an optional view expression *v*. This relationship is denoted as *Domain*(*a*) = *EntitySet*(*e*), *Domain*(*a*) = *EntitySet*(*e*, *v*) in the presence of a view expression, and is the user interface equivalent of a relational foreign key.

The view expression *v* describes how a reference to an entity *e* is displayed on the screen, and produces a string representation of an entity. Formally, *v* is a function *v* :*Nodes*→*String* that takes as input a subset of the attribute nodes that are immediate descendants of an entity node (without going through another entity node) and produces a user-friendly representation of the referenced entity. For instance, if the view expression is the function *v*(*FirstName, LastName*) = LastName + "," + FirstName where + is

the string concatenation operator, then in that control you will see values that look like "Thomas, Bob" when in fact the control stores an arcane object ID value referencing a row in another table.

We also define the following useful functions over g-trees:

- For any g-tree *g*, *rootnode*(*g*) is the root node of the g-tree.

- For any node *n*, *Entity*(*n*) is the nearest entity node to *n* above it in the g-tree, including *n* itself. Conceptually, when *n* represents a control on a form, regardless of the node type of *n*, *Entity*(*n*) is the node corresponding to that form. By definition, *Entity*(*n*) = *n* if *n* is an entity node.

- For any entity node *e*, *Attributes*(*e*) is the collection of attribute nodes that are descendants of *e* following a path that does not include another entity node. Conceptually, since *e* represents a form, *Attributes*(*e*) represents all of the widgets on that form that display data elements, e.g., widgets that correspond to attribute nodes.

- For any entity node *e*, *Parent*(*e*) is the nearest entity node to *n* strictly above it in the g-tree. Conceptually, since *n* represents a form, *Parent*(*n*) represents the form that launched it. By convention, *Parent*(*e*) = *null* if *e* is the root node of a g-tree.

Translating a user interface into a g-tree is straightforward. Each form in the user interface becomes an entity node, each data-bound widget that holds values from an *atomic domain* (i.e., one of the base types supported by the programming language, such as a string or integer) becomes an attribute node, each container widget (such as a group box) becomes a container node. Widgets that display data with non-atomic type

break down into nodes according to Algorithm 2.1, introduced momentarily. Any other graphical widget becomes a control node. Our Guava user interface translator derives the name of each node from the name the developer gives the graphical widget in the application code. If one form or control c_1 contains another control c_2 (e.g., when a group box contains a text box), the translator draws a *Contains* edge from the node for c_1 to the node for c_2. If a control launches another form, but the new form merely contains more details about the first form, the translator draws a *Single-Launch* edge from the control to the form. If, instead, the new form allows creation of several instances for each instance of the first form (evidenced by the presence of new-edit-delete functionality on a form to manage child instances), the translator draws a *Multiple-Launch* edge.

For each widget that holds non-atomic data of type T, we assume that the structure underlying type T (e.g., the constraints governing the structure and values that are stored in the widget) can be represented recursively in the following language:

- $A(D, n, h)$, an atomic data type, where D is an atomic domain (including any reference domain *EntitySet*(e)), n is a name, and h is a context function

- $C(t_0, t_1, \ldots, t_k, n, h)$, representing a tuple type, where each t_i is another expression in the language, n is a name, and h is a context function

- $E(t, n, h)$, representing a set type, where t is another expression in the language, n is a name, and h is a context function (other aggregate types such as lists or bags can be represented by creating a new type $t' = C(t, p, n', \{\})$ for some new name n' and where p is a integer type representing position or repetition number)

The context functions associated with types may be empty (and are often empty in

practice). If a type in the underlying language can be represented as $A(D, n, h)$, then the widget is in fact atomic, and Algorithm 2.1 need not be run to begin with.

Algorithm 2.1: If type T can be represented in the language, translate it into nodes in the following way:

- Translate $E(t, n, h)$ into an Entity node with name n and context h, connected to the tree for type t by a *Contains* edge.

- Translate $C(t_0, t_1, \ldots, t_k, n, h)$ into a Container node with name n and context h, connected to the trees for types t_i by *Contains* edges.

- Translate $A(D, n, h)$ into an Attribute node with name n, context h, and domain D.

  □

Note that the type language and algorithm are effectively building a miniature g-tree to represent the data type, if possible. For example, the grid control in Figure 2.3 will be represented by an Entity node, a Container node, and three Attribute nodes, with domains String, String, and Boolean as also shown in Figure 2.3. If type T cannot be represented in the language, we treat the type T as if it were atomic by translating the control into an attribute node whose domain is String, meaning that the control will serialize its contents to a string.

The context functions for controls involved in Algorithm 2.1 describe the attributes of the various data structures buried within a compound-value control, and are no different from a context function for an atomic data control (e.g., a text box). For instance, the context function for the "Specialty" grid column in Figure 2.3 may be defined on "Default value", "Tool tip", and "Is required".

(a)   (b)
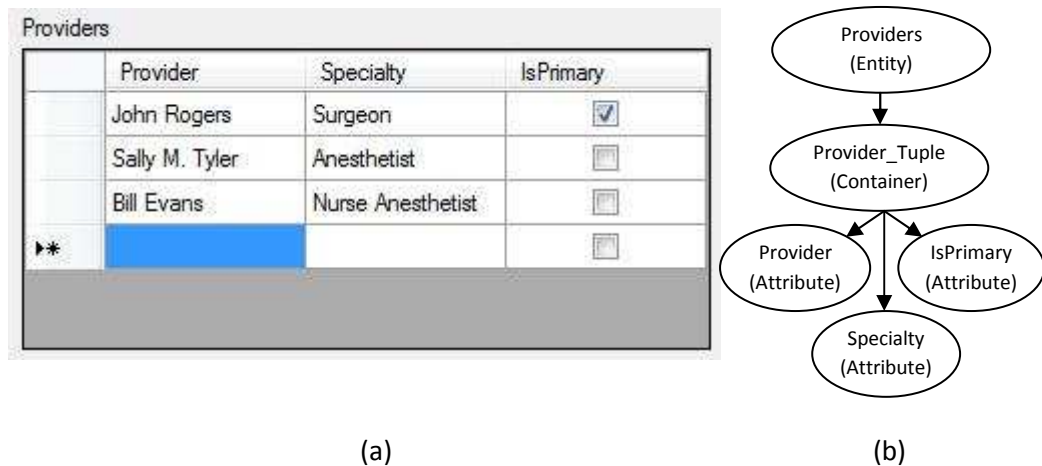
$T_{Provider} = A(String, Provider, \{Default=""\})$
$T_{Specialty} = A(EntitySet(Specialty, V_{Specialty}), Provider, \{Default=""\})$
$V_{Specialty}(SpecialtyName) = "SpecialtyName"$
$T_{IsPrimary} = A(Boolean, IsPrimary, \{Default=False\})$
$T_{Provider\_Tuple} = C(T_{Provider}, T_{Specialty}, T_{IsPrimary}, Provider\_Tuple, \{\})$
$T_{Providers} = E(T_{Provider\_Tuple}, Providers, \{HasMaxNumber=False\})$

(c)

Figure 2.3: A grid control (a), how it breaks down into a g-tree fragment (b), and the underlying type of the grid control ($T_{Providers}$) expressed in our type language (c)
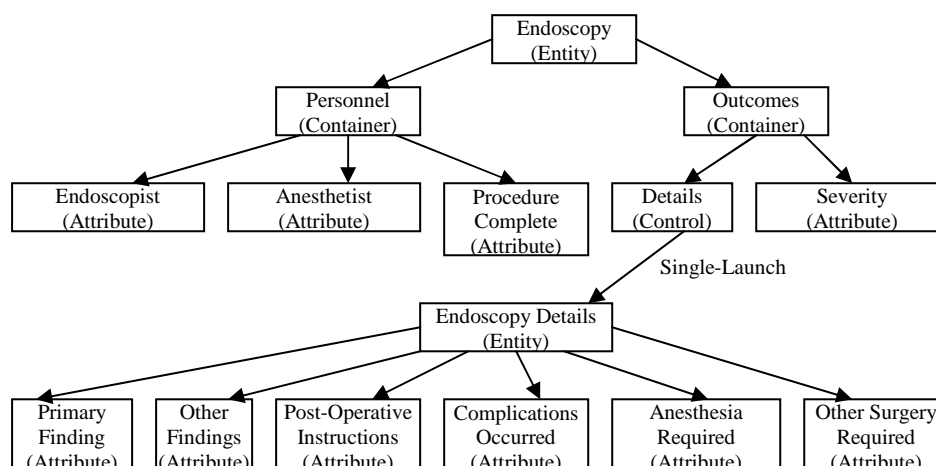
Figure 2.4: An example g-tree corresponding to the application in Figure 2.2. Any edge that is not labeled is a *Contains* edge

Each data-bound widget, whether or not it has atomic-typed data, must get its data from the database through the middleware (and, thus, the natural schema, once Guava is introduced) and put its data back into the database through the same means. Therefore, we assert that each non-atomic data-bound control must already have some sort of data structure that the middleware understands how to work with, and mechanisms for reading from and writing to that structure. If that data structure is relational in nature, then it can be easily described using the language introduced in Algorithm 2.1. If the data in the control is natively stored as a more complex structure that cannot be described using the language above, such as a bitmap image or untyped XML document, then (as mentioned earlier) we treat the control as an atomic-valued control. To store data in such a format in a relational database, one needs to write serialization and deserialization routines (if they do not exist natively for the complex type) so that its data can be stored in a single column.

Figure 2.4 shows the g-tree that corresponds to the UI in Figure 2.2. The entity node for the parent form is the root of the tree, the container nodes representing the group boxes are its children, and the controls in the group boxes become child nodes of the group box nodes. The child form also becomes an entity node, which is a child node of the control that launched it. The edge is labeled as Single-Launch because the child and parent forms share a one-to-one relationship, since there is no new-edit-delete functionality in the parent form and a single button launches the form.

Notice that not every g-tree corresponds to a working user interface. For instance, a single-launch edge leading to an attribute node does not make sense, because that implies clicking a button would launch a text box or a checkbox, not another form. We define a g-tree to be *valid* if it satisfies these properties:

- The root node of the tree is of type *Entity*.

- The in-edge for all non-*Entity* nodes is of type *Contains*.

In Guava, we generate what we call a *natural schema*, a relational schema where each form corresponds to a single table using the following algorithm:

Algorithm 2.2: To translate a valid g-tree (N, E) into its natural database schema:

- For each *Entity* node $n \in N$, create a table with name $n.Name$, and add a column called *id*, an artificially generated primary key.

- For each *Entity* node $n \in N$ that is not the root node, let $p = Parent(n)$. If n's in-edge is of type *Single-launch*, create a foreign key from ($n.Name$).id to ($p.Name$).id. If n's in-edge is of type *Multiple-launch* or *Contains*, create a new column ($n.Name$).fk and a foreign key from the new column to ($p.Name$).id.

- For each *Attribute* node $a \in N$, let $p = Entity(a)$. Create a column named *a.Name* in table *p.Name*. If *a* has domain *EntitySet(e)* for some node *e*, then set the new column's domain to be the domain of artificially-generated keys, and create a foreign key from *a* to *e.id*. Otherwise, set the domain of the new column to *Domain(a)*. □

Figure 2.5 shows the result of running Algorithm 2.2 on the g-tree in Figure 2.4, and is consequently the natural schema corresponding to the forms in Figure 2.2. The natural schema can serve as the schema of the underlying physical database; however, it is more likely that the schema on disk will be significantly different, to accommodate physical design issues such as retrieval and update speed. These issues are covered in the next chapter, Chapter 3.

Given a form G, it is possible that multiple other forms F_1, F_2, and F_3 can launch G. This idea of *multiple form provenance* means that a given instance of an entity of G may be associated with any one of the parent forms. Standard foreign keys can only reference a single parent table; thus, there is no single construct in the relational model to support this possibility of multiple parents without using an active construct such as a trigger.

The underlying data model in Guava is the relational model, but we extend the model with a generalized notion of foreign keys to support the possibility of multiple parent forms. We allow multiple foreign keys to be defined on a set of source columns with 'OR' semantics, which accommodates multiple form provenance. In the standard relational model, a table with two foreign keys from the same columns uses 'AND' semantics.
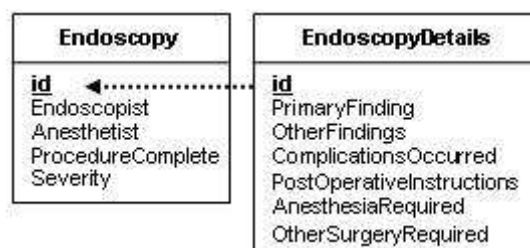
Figure 2.5: The natural schema that represents the data in Figure 2.4

We make one restriction, which is that a single form will not participate as a child in both one-to-one and many-to-one relationships. This restriction requires that if there is a foreign key from *G.id* to some other table, then *G* will only be the source of one-to-one foreign keys, i.e., all the foreign keys are from *G.id* to the *id* attribute in other tables. These multiple foreign keys from *G.id* have and 'OR' semantics. On the other hand, if there is a foreign key from *G.fk* to some other table, then *G* will only be the source of many-to-one foreign keys, i.e., all of the foreign keys will be from *G.fk* to the *id* attribute in other tables. These multiple foreign keys also have 'OR' semantics. This restriction on foreign keys, coupled with Algorithm 2.2, means that if a table has a column called *fk* at all, it will participate in only many-to-one foreign keys. In the universe of forms, this restriction means that if any form is launched in a single-launch relationship with its parent, that form must always be launched using a single-launch; the same can be said for multiple-launch relationships.

In the discussion so far, we have described Guava in a setting with a single UI that addresses a single underlying database. Consider the case where a developer or company creates an entire suite of applications to run on a single database. Each application may be considered individually, with its own g-tree, query interface, and natural schema.

We can also consider all of the applications at once, resulting in a forest of g-trees, a single unified query interface, and a single natural schema. This holistic approach has the benefit that application A could issue queries against tables that only exist in application B to retrieve data through the natural schema. It is also possible that, since the multiple applications are constructed by the same institution, a single form may be used in multiple applications. We assume that the names of forms are distinct between applications unless the forms are identical.

The multiple application scenario is not a major focus of our research. However, we note that there are three interesting cases that may emerge when considering a collection of applications at one time:

- The sets of forms between the applications are disjoint, and the applications do not overlap in terms of real-world entities covered. In this case, there is no interesting interaction between the applications, and no reason to consider the applications together, unless there is a development reason to do so. In this case, Guava applies without any new features.

- The sets of forms between the applications are not disjoint. Considering the set of applications at one time would allow the natural schema to reflect the interrelationships between the applications.

- Two applications each have a form that covers real-world entity set X, but the forms are not the same. This case presents an integration challenge. The two forms would become distinct tables in the natural schema with no connection. We present a potential solution to this case in Chapter 4.

(a)

(b)

Figure 2.6: Two possibilities for the Guava query interface. One interface mocks up each form (a), where each data entry control now becomes a place to enter print and filter statements. An alternative interface with the same functionality creates a tree-structure (b) that mimics the structure of the g-tree

2.2 QUERIES IN GUAVA

The *Guava Native Query Interface* is graphical in nature, based on the original interface, as shown in Figure 2.6(a). Guava uses context information to generate the query interface for the application. To specify a query using the interface in Figure 2.6(a), the user navigates through the forms just as they would with the original user interface. The user specifies which attributes to return in the query answer by right clicking on a control and selecting the "print" option. The user also specifies the conditions on which to filter rows by filling in data into the various data fields, similar to QBE [87]. For example, the query in Figure 2.6(a) prints the endoscopist of all completed procedures where the anesthetist was Bob, the severity was Abnormal, and complications occurred. Unlike QBE, the user does not need to specify any join conditions; Guava only supports the joins that are implied by the relationships between forms.

Other query interfaces are also possible, such as one that exposes the controls in an entire application in a single tree as shown in Figure 2.6(b). The interface in Figure 2.6(b) is similar in functionality to the previous interface, except that the user prints a data control by checking the box next to that control's name in the tree. The context of each control, as well as the interface to specify conditions, can be viewed in a separate window. The user can also search the tree for data controls: The interface takes a search term and finds all occurrences of the term in the context elements for each control. The interface highlights and expands each matching control, and lists the context elements that matched the term.

Guava takes the query as specified by the user and translates it into relational algebra against the natural schema. The first step in the translation is to accumulate the query

into a data structure.

Definition: A *decorated g-tree* is a g-tree where, to each entity and attribute node, we attach two additional pieces of data:

- A Boolean value *Print* representing the user's decision to return the value of the attribute (or ID for the entity), similar to the print flag in QBE. The default value for *Print* is false.

- A Boolean function *Condition* in the form of *Node θ Value* where *Value* is any constant value, *Node* is the current node, and *θ* is any of the standard six comparison operators. The function can also be a conjunction of such expressions, or *True* to indicate that there is no filter for that node.

A decorated g-tree may also have replicated entity children. In other words, if an entity node *e* appears as a child of control *c* in a g-tree, the decorated version of the g-tree may have multiple, distinct instances of *e* (and the entire tree beneath *e*) as children of *c*. For instance, one may want to express the query "find all patients who have seen both doctors Bob and Alice", where physician is a child node of patient; in this query, there are two physician child nodes of the patient node.

Intuitively, Guava constructs a decorated g-tree by taking the specified print statements and filtering conditions from the query interface and attaching them to the node in the g-tree representing the control on which they were found. We define a query in the Guava architecture as follows:

Definition: A node *n* in a decorated g-tree is *non-trivial* if its *n.Print = True* or its boolean condition expression is anything other than *True*.

Definition: A *g-subtree* of a g-tree g' is a subtree of g where the root node is an entity node. A *pruned g-tree* is the smallest g-subtree of a decorated g-tree that contains all of the non-trivial query nodes of the subtree. This corresponds to the idea that a query may be able to start with a sub-form of the application, and not have to start with the root form. That is, a pruned g-tree has an entity node at its root that may be different from the root of the original tree.

Definition: A *Guava query* over a g-tree g is a forest of g-subtrees of g where all of the following are true:

1. One of the g-subtrees g_0 is set aside as a "distinguished" subtree, and g_0 is pruned.

2. Every non-distinguished subtree t in the forest has an entity node e that can be associated with an attribute node a in another tree where $Domain(a) = EntitySet(e)$.

3. No two non-distinguished subtrees are associated with the same attribute node in the same subtree.

4. The associations between subtrees form a tree. In other words, if we were to consider each association to be an additional edge, the result of putting all of the the subtrees together is itself a tree.

5. At least one of the nodes in the forest has a boolean print value set to *True*.

In addition, the query provides a function U that takes an entity node in the forest and maps it to a unique name. We need this function because there may be entity nodes in common between subtrees in the query forest (corresponding to the case where the same form is instantiated more than once over the course of constructing a query), so we need some way to disambiguate different instances of the same entity nodes.
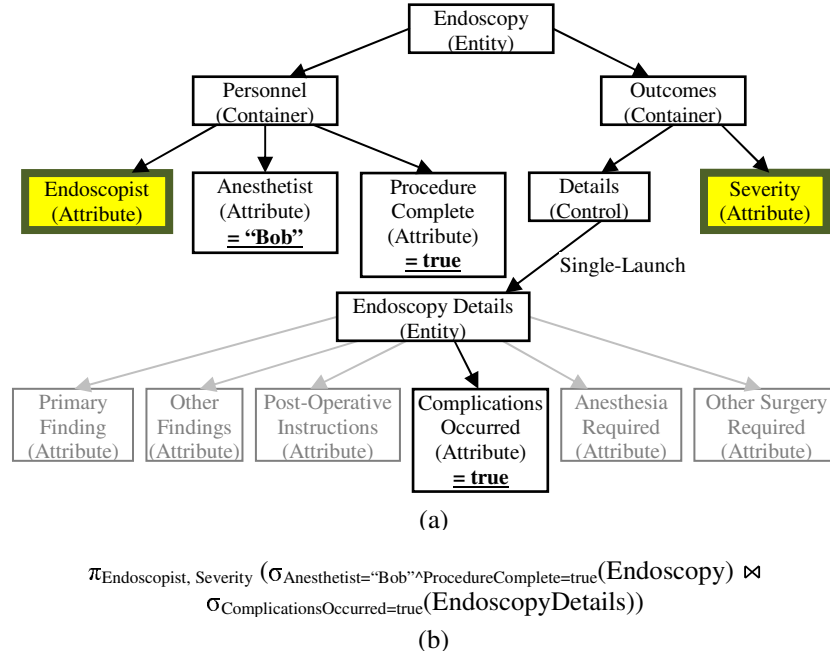
Figure 2.7: The pruned, decorated g-tree (a) corresponding to the query in Figure 2.6(a), and the relational algebra query that results from running Algorithm 2.3 on it (b)

Finally, we define $SingleTableQuery(e) = \rho_{e.Name \to U(e)}(\sigma_V(e.Name))$, where e is an Entity node in a decorated g-tree, ρ is the relational renaming operator and V is the conjunction of all boolean condition expressions in $Attributes(e)$ and that of the entity node e itself. With this machinery in place, we translate a Guava query into relational algebra.

Algorithm 2.3: To translate a Guava query q over a decorated g-tree g into relational algebra expressed against the natural schema of g:

1. Begin with the root node r of the distinguished g-subtree in the query. Set $TQ = SingleTableQuery(r)$.

2. Also, if $Print(r) = True$, set $TP = \{U(r).ID\}$. Otherwise, set $TP = \varnothing$.

3. Traverse the subtree in depth-first, pre-order fashion.

4. For each entity node e found with a *Single-Launch* in-edge, set

$$TQ = TQ \bowtie_{U(Parent(e)).ID=U(e).ID} SingleTableQuery(e).$$

5. For each entity node e found with a *Multiple-Launch* or *Contains* in-edge, set

$$TQ = TQ \bowtie_{U(Parent(e)).ID=U(e).FK} SingleTableQuery(e).$$

6. For each entity node e found with $Print(e) = True$, set $TP = TP \cup \{U(e).ID\}$.

7. For each attribute node a found that is associated with another g-subtree g' with associated entity node e and root node e_{root} by relationship $Domain(a) = EntitySet(e)$, construct the query TQ' associated with g' (running the algorithm from the beginning). Then set $TQ = TQ \bowtie_{U(Entity(a)).a=U(e).ID} TQ'$.

8. For each attribute node a found with $Print(a) = True$, set

$$TP = TP \cup \{U(Parent(a)).(a.Name)\}.$$

9. Once traversal of the entire forest is complete, return $\pi_{TP}(TQ)$. □

Figure 2.7(a) shows the Guava query associated with the query shown in the query interface of Figure 2.6(a). Pruned nodes are grayed-out in the tree. The result of running Algorithm 2.3 on the Guava query in Figure 2.7(a) is the relational algebra expression shown in Figure 2.7(b).

By construction, we see that the resulting Guava query language is a subset of the relational algebra restricted to the following:

- *T* for any table in the natural schema

- $\pi_{\vec{C}}(Exp)$ for any collection of columns $\vec{C}$

- $\sigma_{C\theta V}(Exp)$ for any column C, value V, and comparator θ

- $Exp1 \bowtie Exp2$ only if there is a foreign key from $Exp2$ to $Exp1$

From the recursive construction of the above subset of relational algebra, we also see that any expression in the language can also be represented as a Guava query, making the two languages equivalent:

- For any table T in the natural schema, one can construct a Guava query by finding an Entity node in a g-tree corresponding to the form for T and decorating the nodes in $Attributes(e)$ with the $Print$ decorator.

- To represent $\pi_{\vec{C}}(Exp)$ for any collection of columns $\vec{C}$, start with the Guava query for Exp and remove the $Print$ decorator from any node not in $\vec{C}$.

- To represent $\sigma_{C\theta V}(Exp)$ for any column C, value V, and comparator θ, start with the Guava query for Exp and add the appropriate condition decorator to the node corresponding to C in the query.

- To represent $Exp1 \bowtie Exp2$ where there is a foreign key from $Exp2$ to $Exp1$, first construct the Guava queries for $Exp1$ and $Exp2$. Then, consider whether the foreign key corresponds to an edge in the g-tree or a Lookup association. If the foreign key comes from an association, create a new Guava query out of the union of the original queries' subtrees. Set the distinguished subtree of the new query to be the distinguished subtree from the query for $Exp1$. If the foreign key comes from a g-tree edge, do the same thing, except take the subtrees from the $Exp1$ and

*Exp*2 that participate in the foreign key and connect them into the same subtree along the edge in question.

Therefore, the Guava query language is equivalent in expressive power to single-statement conjunctive queries where joins are restricted to foreign keys and selection can use any of the six comparators to relate a column against a constant, but not a column against another column.

Using Algorithm 2.3, all joins in the language are inner joins, so any query that spans multiple forms will only return results that matches data that is present in all of the forms.

2.3 IMPLEMENTATION NOTES

We have implemented portions of the functionality above to demonstrate proof of concept of the query interface in the Microsoft Visual Studio 2008 Integrated Development Environment (IDE) using the C# programming language. We provide a library of GUI widgets by subclassing the standard widgets that come with Visual Studio. We have implemented a framework that can take any application written using those widgets and generate all of the artifacts in Figure 2.1. One can use the prototype framework with any language that runs on the .Net framework, e.g., C# or Python, allowing the developer the freedom of several possible programming languages. One need only to implement the UI widgets in another language such as Java to extend to yet more languages. In this section, we describe characteristics of our implementation that are noteworthy.

2.3.1 Reflection

Reflection is a feature present in many modern programming languages that allows a program to examine an object at runtime and determine what properties it has and what values are in those properties. Using reflection, a program can determine an object's type and any interfaces it implements; discover what properties, methods, and fields exist on the object; and retrieve pointers to methods and properties that can be called, all without the benefit of knowing the object's type ahead of time (i.e., without requiring static typing). Reflection can also be used to generate objects at runtime when the type of the object is not known a priori. The developer takes the name of a class stored as a character string and can use reflection to find the constructor for that class and instantiate it to create an object.

We use reflection in C# to reify the relationships between forms. Suppose that a developer defines a button on form "foo" that launches form "bar" when clicked. Normally, to bring about this effect, the developer manually writes code for "foo" associated with an event that fires when the button is clicked. That event code cannot be queried at run time, so there is no way to ask of the button questions like "which form do you launch?" Guava's version of a button provides a property called "LaunchForm" that holds the name of the form that the button should launch if clicked. Behind the scenes, Guava uses reflection to take the value of this property and create an object of that type.

2.3.2 Interfaces

Though reflection can provide the names of methods and properties, it provides no information about the semantics. Suppose we want to get the current value in a control.

If the control is a TextBox, the property that holds the current value in the box is called "Text". However, if we want the value in a CheckBox, the property that holds the current value is called "Value". What is worse is that the CheckBox control also has a property called "Text", but it returns the leading text of the control — in effect, schema information rather than data. Reflection would only allow the programmer to find out that both controls have a property called "Text", leading one to incorrectly infer that they serve the same purpose.

We use a different programming language feature — an interface — to document the uniform semantic meaning of methods and properties across widgets. An interface is a contract that an object must meet in terms of properties and methods that the object supports; the interface provides no implementation of those properties or methods, so a developer must provide the code. Our interface, called "Guava", contains many properties, including the following:

- `GName`, the name of the control

- `GText`, the leading text of the control

- `GToolTip`, the tool-tip text that appears when the pointer hovers over a control

- `GDomain`, the domain of allowable values for a control

Each of these properties must be connected to existing properties or functionality of a control, if they exist for the control; otherwise, the programmer must introduce the new functionality required by the property (which is a very rare occurrence). For instance, the `GText` property is set to the "Text" property for a CheckBox control. For a

TextBox control, the `GText` property is set to the text from the label that appears before the TextBox. Therefore, `GText` is unambiguously treated as schema, not data.

We subclassed each of the available data-bound controls in the Visual Studio environment (e.g., text boxes, check boxes, and grids) so that there are analogs of each of those controls that implement the Guava interface. For instance, the class "GTextBox" is precisely a TextBox that has been extended to implement the Guava interface. All that an existing implementation of a user interface must do to use Guava is include our code library and change the types of the graphical widget objects to their "G" counterparts. We talk about implementing custom-built controls to match the Guava interface in the case study in Section 2.4.1, below.

2.3.3 Query Results and Query Interface

The query language supported by Guava is a subset of relational algebra. Thus, query results are relations. Therefore, results can be viewed in a grid or table. The current implementation of our query interface (shown in 2.6(b)) displays results as a grid, and allows the results to be exported to a spreadsheet. However, since queries can be expressed in an environment that mimics the original user interface, it makes sense that one might want to view the query results in that same environment. It would be straightforward to implement a browser for the query answer that shows each query result, in turn, in a display based on the original UI. We would simply show, for each row, a mockup of each form referenced by the query with each of the "printed" fields filled in.

In a meeting with CORI's analysts on September 14th, 2006, the existing query interface and grid layout of query results were presented. Though the analysts clearly

were impressed by the prospect of a fully-graphical query interface, they said that the existing state of the implementation was "exciting" and "very useful". Though they thought that viewing results in context might be helpful, their primary use for query results would be loading them into a statistical package for further analysis, and that the spreadsheet form of the results were sufficient.

2.4 CASE STUDY 1: MODELING AN EXISTING USER INTERFACE USING GUAVA

To demonstrate the feasibility of using Guava, we took the existing CORI application (version 4.0) and converted a significant subset of it to use our Guava implementation. We did not convert the entire CORI application into the Guava model because a vast majority of the forms in CORI are small forms with commonly-used controls on them, such as checkboxes, buttons, and text fields. Thus, there was no compelling reason to implement all of them in Guava. However, we did implement enough of the CORI application to demonstrate the following:

- All of the available types of graphical controls used by CORI are available in our implementation, including both standard Visual Studio data-bound components and custom controls.

- All of the different ways to launch forms or represent entities in the CORI application are available in our implementation. The primary entities in CORI are patients, staff, procedures, and findings; a complete coverage of CORI for our purposes needed to include all of these entities.

- The procedure query screen in CORI is at least partially implemented, in the sense that it can cover a similar set of queries as the page in the original application. CORI includes a screen that allows a user to find any procedure for any patient. Normally, a procedure is a weak entity of a patient and is accessed through the patient forms. We implemented a custom procedure search page in our case study because we felt that it was important to cover at least one case where the application UI needs to issue an ad hoc query against the database through the natural schema.

The next subsections describe the parts of the CORI application that we implemented with respect to the items above, and our experiences doing so.

2.4.1 Adapting Controls to Work With Guava

Many of the controls used in the CORI application are standard forms widgets that come with Visual Studio, and therefore are included in the set of controls that we subclassed, as described in the previous section. Another set of controls used by CORI are functionally equivalent to existing controls; for instance, CORI procedure screens have a columns of labels lining the left edge of the screen, as in Figure 2.8. Clicking on one of the labels loads a panel of controls into the main area of the form. This situation is functionally identical to a TabControl, such as the one seen across the top of Figure 2.6(b). So, our implementation of the CORI application in Guava uses a TabControl instead of the list of vertical labels. When there is an application with a functionally equivalent control that already exists in Guava, we document it and use the existing control.
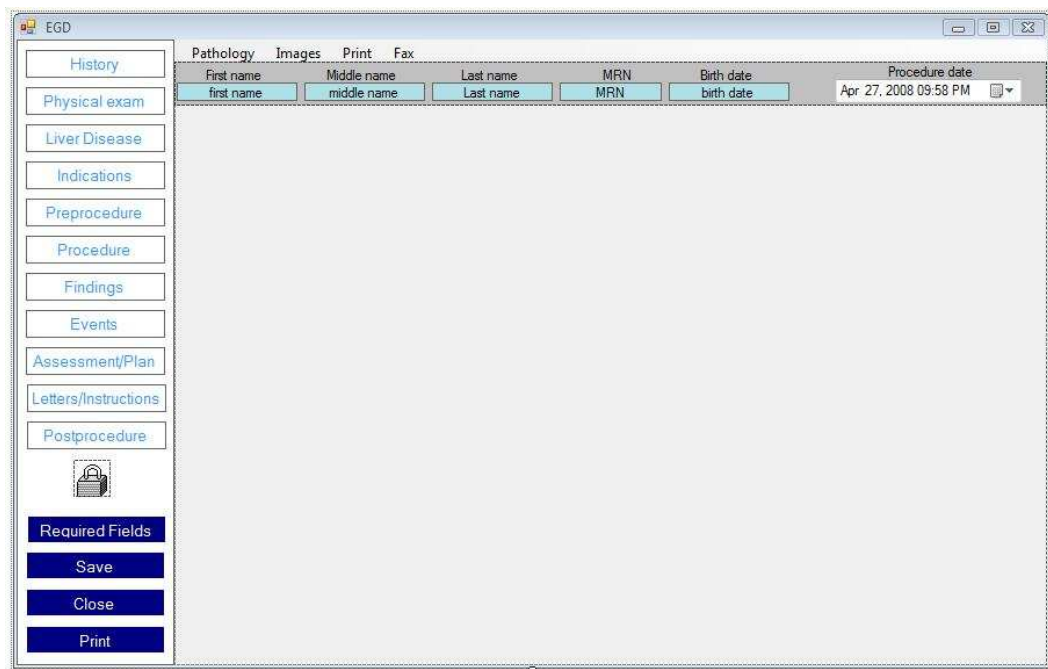
Figure 2.8: One of the main procedure screens in CORI; clicking on a label on the left will load a panel of controls into the empty space in the lower right

Figure 2.9: A custom checkbox control that exists in CORI that we have modified to work in Guava

There are two controls that are custom to CORI that we could not mimic with existing controls. The first is called a "CheckFlag", as shown in Figure 2.9. The control is similar to a standard checkbox control, except that clicking it may launch a new form if the developer has designated a target form, similar to clicking a standard button control. The color of the CheckFlag depends on whether there is any data contained in the launched form. If the launched form is empty, the control's check mark has the standard white background; if the launched form contains data, the background is amber, as shown in the figure. For this custom control, we connected the various properties of the Guava interface to existing properties without needing to add new functionality; in other words, we were able to implement all of the `GName`, `GText`, `GDomain`, etc. properties in one or two lines of code that reference other existing property values of the control.

The second custom control in CORI is called "DrawFinding", and is shown in Figure 2.10(a). The control is a graphical representation of anatomy; a user can click on a location within the anatomy (or draw a small region) to demonstrate that a clinical finding was discovered at that location. Clicking on the location brings up a list of possible findings for the current procedure; selecting a finding launches a form associated with that finding.

The biggest problem to overcome when migrating the DrawFinding control to Guava was a lack of encapsulation. The code of the DrawFinding control was actually spread between the control itself and each form that could possibly contain the control. For instance, the code that runs when the user clicks on the control is passed in from the enclosing form, such as a Colonoscopy form, as procedures that are called as an event.

That way, when the user clicks on a DrawFinding control when the application is running, it brings up a list of procedure-specific findings. Here is part of the code that is passed into a the DrawFinding control from a Colonoscopy procedure:

```
case "Arteriovenous Malformation (AVM)":
cform = new CorUI.COL.FindingDetails.ColAVM();
break;


case "Diverticulosis":
cform = new CorUI.COL.FindingDetails.ColDivertic();
break;


case "Fissure / Fistula":
cform = new CorUI.COL.FindingDetails.ColFissureFistula();
break;
```

The code above associates each finding with the appropriate form to launch if the finding is chosen.

Because the Guava implementation uses reflection, the Guava-ized version of this control does not need to have any code passed in. The "GDrawFinding" control has a single property, "Forms", that is a list of associations. For instance, one item in the list might be "Arteriovenous Malformation (AVM)—CorUI.COL.FindingDetails.ColAVM", to associate the AVM finding with the ColAVM form.

Other than drawing outside code into the control or into properties whose value can be set on the designer window (such as the "Forms" property), the remaining code of
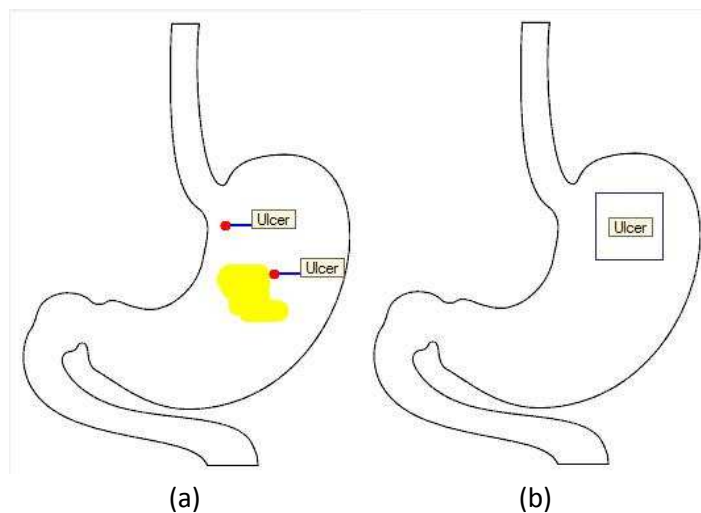
Figure 2.10: A custom graphical control that exists in CORI that we have modified to work in Guava; in data entry mode (a), the user can specify the locations of findings, and in query mode (b), the user can query for findings based on location

the DrawFindings control was left untouched, which is an important characteristic: In both of these custom controls, the existing graphical and behavioral code of the controls is left undisturbed.

Most controls, especially those that come with Visual Studio, have a clear analog in a query-by-example framework. For instance, a text box in "query mode" acts as a filter based on what is in the text box (entering "Bob" in the box limits query results to those that have "Bob" in that field). The choice of how to implement a query interface using a DrawFindings control is not so clear; we developed one simple query version of the control. As shown in Figure 2.10(b), in query mode, the user draws a boundary box rather than drawing individual points; the query results are then filtered for findings that are located within that box. We do not assert that there is only one way to alter

the control to support visual queries; rather, we demonstrate by proof-of-concept that custom graphical controls can work in the graphical query metaphor. It took less than 20 minutes to add this query mode extension to DrawFindings.

2.4.2 Adapting an Entire UI to Use Guava

The vast majority of the CORI application is a hierarchical arrangement of data-entry forms with standard controls, which matches the Guava paradigm. We describe here the cases where the CORI application has some functionality that diverges from this paradigm. All of these cases occur on the main screen of CORI that appears when the application is launched, also known as the Lobby screen.

The first major issue with the Lobby screen is how patients are handled. Figure 2.11 shows the Patient form in the Lobby screen. On this screen, the user provides search criteria for patients in the upper-left corner of the form. The retrieved patients appear in a grid in the lower-left corner (position A in the figure). When the user selects a patient, that patient's details appear in the upper-right corner (position B in the figure). A list of the patient's procedures appears in the lower-right corner (position C in the figure). From the lower-right corner, the user can open an existing procedure or create a new one.

This layout is unlike the rest of the application because it does not subscribe to a one-entity-per-form paradigm. Such a paradigm would have the Lobby screen (its own entity) launch a new form specifically for a Patient when an item in the grid (position A) is selected. Then, in that new form, a Procedure could be selected. We mocked up this part of the UI in our implementation of CORI using a one-entity-per-form style. Because this

Figure 2.11: The screen in CORI for finding patients and entering new ones

setup is functionally equivalent to the situation where Lobby-Patient-Procedure are flattened into one form, we assert that it would be simple to Guava-ize the compound Lobby screen without decomposing; we can combine our Lobby-Patient-Procedure forms into a single, flattened form where the boundary around each nested entity is marked as an entity node rather than a container node.

The other major difference is the Procedure screen, shown in Figure 2.12(a). This form (also part of the CORI Lobby) allows a user to query for existing procedures, which is a departure from the normal Guava paradigm because, within the CORI database, a procedure is a weak entity of a patient. In other words, procedures are found or added by looking up a patient, then looking at procedures. The form shown in Figure 2.12(a) packages up a query over procedure data and returns the list of all matching procedures,

irrespective of patient. Procedures are still weak entities; the Lobby merely presents an alternative access path for finding them.

Figure 2.12(b) shows the same form in our implementation. Our proof-of-concept version of the procedure query form has fewer displayed options (adding more options would have been time and labor intensive, but not difficult), but similar functionality: The user chooses some criteria, the software generates a query, and the list of matching procedures appears. Clicking on a procedure launches the Procedure form for that entity. So, this form packages a query in relational algebra and sends it to the natural schema, rather than sending raw SQL to the database. This situation illustrates how an application's business logic can still operate on top of the natural schema without requiring access to the physical database.

2.4.3 Additional Results

According to team member Jeremy Steinhauer, who handled converting CORI forms to use Guava:

> Having not much experience with gui development it took me longer than expected to figure out what was needed as well as the best approach to converting the forms to use guava. After trying a couple different ways of migrating the forms I found that the best way to maintain data requirements, conserve look and feel, and convert quickly was to copy and paste directly from the code the form elements, then adjust the properties by deleting extraneous and renaming the similar ones. In the end I was able to get the conversion time for a single form down to about 30 mins for about 97% of

(a) Procedure selector from the CORI application

(b) Procedure selector from our mock-up of the CORI application

Figure 2.12: Forms in the CORI app (a) and our converted app (b) that find procedures based on specified criteria

the code. The other 3% of components required more time but it was noth-

ing that could not be mimicked with more time under the Guava architecture

as far as I could tell. (05/12/2008)

Jeremy's task was more difficult than would normally be expected because he did not

have access to a running installation of CORI, let alone any familiarity with the source

code. Even in this setting, his experience was simple and positive.

An analysis of the CORI design in Guava yielded the following numbers:

- 124 widgets were non-data related and trivial to switch to Guava.

- 205 widgets (86% of the data-bound widgets) were either built-in Visual Studio
  widgets, CheckFlags, or DrawFindings and thus had direct Guava analogs.

- 24 widgets (10% of the data-bound widgets) had functional equivalents that were
  already Guava-ized. We assert that explicit, Guava-ized versions of these widgets
  could be implemented with a minimum amount of work, given our experience
  adapting CheckFlag and DrawFindings.

- 6 widgets were data-bound, but not connected to persistent storage. These con-
  trols, such as a list box that lists the "checked" items on a different form, just
  derive data from other controls and are simple to code. Because they are not data
  bound, they are not in the g-tree of the application and are trivial to switch to
  Guava.

- 1 widget required a base data type of binary-large-object because it is an image
  widget. Our Guava implementation does not yet support this data type. Support-
  ing this data type in the implementation would be simple, so long as we restrict

the user from specifying conditions on image types in the query interface (it is unclear what kinds of conditions could be specified, or even valid to begin with, on images in standard SQL).

- 1 widget deals with Pathology data. Pathology data comes from entities that do not have a corresponding form (i.e., there is no single form in the application to enter Pathology items). Therefore, this is a control that intentionally bypasses the natural schema and connects directly to the database (which corresponds to its existing behavior anyway).

From these results, we conclude that the user interface of the CORI application can be brought into the Guava architecture solely by changing the types of objects from standard controls to our Guava equivalents and making a small number of custom controls conform to the Guava object interface.

2.5 CASE STUDY 2: GUAVA AS AN ADDRESSING SCHEME

Context is an idea that is central to Guava; one of the key features of the Guava query interface is that it provides contextual information to users, so that the user may better understand schema and data elements when constructing queries. Guava is not the only research project that seeks to leverage contextual information for the benefit of users. The SPARCE project (Superimposed Pluggable ARchitecture for Context and Excerpts) [60] provides an architecture for managing small excerpts of information (called *marks*) drawn from base information sources. Each mark contains a number of pieces of information, including a link back to the document from which the excerpt was drawn.

A mark can also be used to draw contextual information about an excerpt, so for information drawn from a word processing document, SPARCE can determine the text surrounding the excerpt, the formatting used for the excerpt, the excerpt's position in a document outline, etc.

SPARCE can access data in a number of different formats, including word processing documents, spreadsheets, PDF documents, and web pages. It is also an extensible architecture in the sense that a developer can write a component called a *context agent* that allows SPARCE to interact with a new type of data. In this section, we describe our effort to provide interoperability between the Guava and SPARCE projects by making the Guava architecture itself act as a context agent, thereby giving SPARCE access to data that is present on forms from any application built using Guava. Our work demonstrates an alternative use for our g-tree artifact, where g-tree nodes can be used as an addressing scheme for data components in an application.

A SPARCE *base application* is any application that can manage data and has an associated context agent. For instance, Microsoft Word is a base application because there is a context agent written that can interact with Word and generate excerpts from a word-processing document open in the application. A *superimposed application* is an application that can store and manage marks from base applications by receiving marks from context agents. Through a context agent, a superimposed application and a base application can communicate in three fundamental ways:

- A user can create a mark in a base application and place it in the superimposed application. For instance, suppose a user has Microsoft Word open with a document open. The user can select a segment of text, mark it (the Word context agent

is a plug-in into the application, so one can simply click a button in a toolbar to mark the text), and have the mark appear in the superimposed application. For a Guava-enabled base application, this functionality means being able to right-click on any data-bound control in a UI (e.g., a text box), have a "mark" option available in the resulting pop-up context menu, and have the contents of that control (for the data currently displayed in the form) made available in the superimposed application.

- A user can select a mark in a superimposed application and see contextual information about the mark. For instance, in a superimposed application, the user may right-click on our mark from Word and select "view context". The superimposed application will invoke the context agent for Word, and the context agent will then display contextual information about the mark, including information about the excerpt's position in the document and all of the other context items mentioned above. For a Guava-enabled application, this functionality means that one can view the context information of a marked control (help text, label, etc.) from the superimposed application, without launching the original application.

- A user can open a mark in a superimposed application and view it in its original context. For instance, in a superimposed application, the user may right-click on our mark from Word and select "open base document". Microsoft Word would launch, the proper document would open in Word, and document will scroll to the excerpted text and highlight it. For a Guava-enabled application, this functionality means that the superimposed application can launch the Guava-enabled application and navigate through the forms of the application, launching them

each in turn, to display the form that contains the marked control, and with the form displaying the same entity's data that was displayed when the control was marked.

As a motivating example, consider a situation where a clinic is using the CORI procedure software from the case study in Section 2.4, and the physicians in the clinic would like to do rounds. Over a period of time, a physician can collect excerpts from a patient's chart (possibly from both CORI and other software applications) into a summary document. When presenting the summary document at rounds, the physician can then address questions about a finding in the summary document by displaying the finding in the software that had recorded it, with the patient's chart open for context and the finding highlighted.

To extend Guava to function as a SPARCE context agent, we needed to add the following features to our Guava prototype to enable it to participate in the three functions above:

- A mechanism to allow a user to select a particular control on a form, excerpt it (i.e., create a mark), and make it available to an outside application — in the case of a Guava application, an excerpt means the data contained in the control, combined with a way to describe the location of the control in the application.

- A mechanism to allow an outside application to access the context information for a particular control. Since each base application can determine its own definition of context, we expose the contextual information that is exposed by each control (and is thus available in each g-tree node).

- A mechanism to allow an outside application to launch a Guava-enabled application (e.g., CORI) and navigate to a particular place in the form hierarchy.

All three of these items share two base requirements: an addressing scheme for an arbitrary Guava-enabled application that can uniquely identify a piece of information, and a navigation scheme to describe, given an address, how to find that address in the application. We can leverage our g-tree artifact to provide both such schemes. The granularity of our addressing scheme is at the level of attribute nodes, since those are the atomic structures in a g-tree that represent the presence of data.

A *Guava seed*, or g-seed, is a node in a g-tree, combined with the following:

- The path of g-tree nodes from the root of the g-tree to the node.

- A key value for every entity node in the path.

To demonstrate the utility of a g-seed, consider the case of the Other Findings control in the Endoscopy Details form in Figure 2.2. In this application, we may want to excerpt the information, "Other Findings for Endoscopy Details number 104". The address of this information is the Other Findings textbox on the Endoscopy Details form, when the form is viewing the details information for key value 104. This address is similar to, for instance, identifying a field in a database by providing the table name (Endoscopy Details), column name (Other Findings), and primary key value (104). In a software application, it is not possible to simply launch the Endoscopy Details form for that particular key value without navigating through other forms first; rather, one launches the root form of the application, then navigates through other forms (selecting a particular
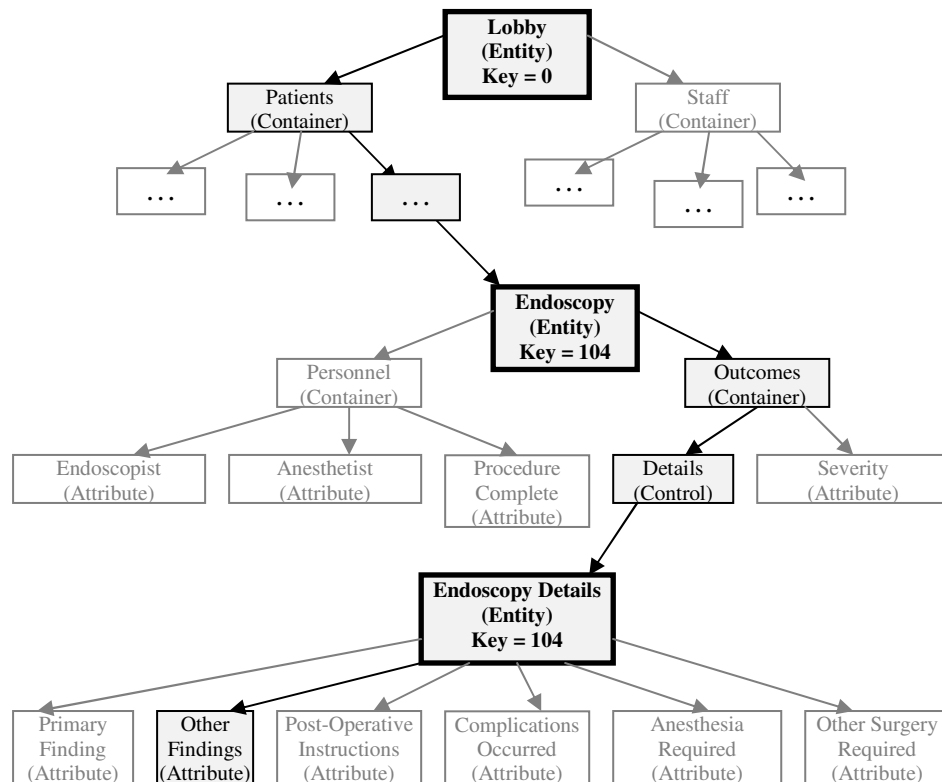
Figure 2.13: A g-seed for the Other Findings text box control from Figure 2.2; the darkened path of nodes through the tree describe the path of forms necessary to reach the Other Findings control in the hierarchy, and in particular, the Other Findings value for Endoscopy Details number 104

entity of interest for each form along the way) until one arrives at the Endoscopy Details form for value 104.

A g-seed encapsulates both the addressing scheme and navigation scheme described above; the specific g-seed describing "Other Findings for Endoscopy Details number 104" is shown in Figure 2.12. The address of the information corresponds to an attribute node in the tree. The navigation path to that attribute is the path from the root node of the g-tree to the attribute node, with key values attached to each entity node along the way, representing which entity's information is loaded into each form while navigating to the information of interest.

Solving the addressing and navigation issues using g-seeds was the only interesting research issue that we encountered in our Guava/SPARCE integration work. Once we implemented g-seeds as part of our prototype, all of the remaining work was on developing interfaces so that the two software implementations could interact, and a small component that can launch Guava-enabled applications and forms.

2.6 RELATED WORK

There are several approaches that model a user interface as a tree structure and view the associated data as an XML document, including XAML [84], XUL [86], and XForms [85]. These XML-based approaches are similar in spirit to Guava but they are limited to describing a single form at a time; there is no automated support for describing the relationship among forms other than by using a programming language. These technologies do not have a native query language, though XQuery can be used to query documents in each format.

Several projects have studied the relationship between forms and data. For instance, Rollinson and Roberts [66] describe how to represent the semantics of a forms-based interface in a conceptual modeling language. The applications they consider are limited to ones where the UI and the database are closely related, perhaps even where the UI is semi-automatically generated from the database. Other studies of forms, such as that provided by Draheim and Weber [20], extensively study the methodology behind forms and form-based modeling, but do not expressly describe how to connect forms to persistent storage nor how to pose queries. We describe our mechanism for connecting to persistent storage, a channel, in Chapter 3.

The Natural Forms Query Language [22] allows users to write forms that serve as the interface to an underlying database, both for updates and for queries. The relationship between NFQL forms and the underlying database is established using name matching, vocabulary analysis and user-guided heuristics. With Guava, the difference between the structure of the forms and the database can be much greater than NFQL (by using a channel, introduced in Chapter 3). NFQL forms must use standard widgets such as check boxes and text boxes; the Guava query interface can use both standard and custom widgets.

Several available commercial software packages are dedicated to managing forms for data entry and coordinating them with a database back-end. IBM has two different tools [35, 36] that allow a user to create data-entry forms, and have the database automatically generated and connected to the forms. Microsoft Access [50] has a similar feature, where a developer creates a form and the underlying tables and columns are generated based on the types of the controls on the forms and the relationships between

the controls. These tools focus on individual forms in isolation and do not consider entire applications. Also, the forms in these applications cannot double as query interfaces — one must run queries directly against the database.

Microsoft InfoPath [51] and Crystal Reports [15] are tools that allow a user to create a form for viewing query results, then connect the form to an existing data store. In both of these tools, the predominant way to connect a form to data is using SQL. With InfoPath, one can also enter data into the forms, similarly to the IBM tools and Microsoft Access mentioned earlier, if the mapping between the form and the database supports update. These two tools are marketed as reporting tools that can generate "executive summaries" of large amounts of data from possibly multiple sources. Their connections to the underlying databases are as flexible as the mapping language; however, once the mapping is specified, the connection between form and database is fixed. Thus, one uses a form to view the result of a pre-determined query rather than to specify a query on the fly.

One of the key features of Guava's native query interface is providing contextual information to the user. Another system that leverages context is the Context Interchange project (COIN) [70], which provides contextual information alongside data in a database. The concept of context within COIN can include the same kinds of context as Guava, such as caption or help text information, but context in COIN is primarily used for converting data from one context to another. For instance, in COIN one can say that the value '5' has context "Units = cm"; when compared against the value "10, Units = ft", the '5' is converted to feet so the comparison can occur in the same context. The context information of the data in COIN is unavailable until a query is processed, and

is intended to be implicit until needed rather than be explicit for the benefit of the user. Also, COIN requires infrastructure within the database to handle the context information, where Guava artifacts are entirely in application space.

The traditional artifact used to discover and explore the schema in a database is a *data dictionary*. One example of a data dictionary is described in documentation that accompanies the Logician electronic medical record software (now called Centricity EMR [11]). This document [44] contains a listing of all of the tables in the database, the columns in each of the tables, and whether each column participates in a key or a foreign key. Each table and column is accompanied by a detailed description of its use. The data dictionary also contains database diagrams in Entity-Relationship format describing the relationships between the tables. Finally, there is some front matter describing the naming scheme behind the columns; for instance, the column "pid" always means "patient id", while "xid" always means "expiration id". This documentation was assembled by a full-time technical writing team, in tandem with information provided by developers. All of this documentation is intended to help a user execute queries using SQL against a database, without using any kind of domain-specific interface.

A data dictionary is an example of a broader category of artifact called a *metadata repository*. Literature on metadata management and metadata repositories describes in detail the kinds of information to be tracked. For instance, Hay describes a wide variety of metadata that can be captured about an information system [31]. On page 9, Hay lays out a variety of kinds of information to be included in his conception of a metadata repository, including security, event processing, rules, tactics, use cases, state transitions, and the usual physical data layout information (tables and columns). In

particular, Hay's table includes database design and user interface design, which are the primary components of a Guava query interface. Hay describes in detail the kinds of information to track, but provides no insight into gathering this information.

Tools such as SAS Metadata Server [69] can store and track changes to the kinds of metadata mentioned above. In addition, more modern database systems can generate reports and diagrams using catalog information on tables, columns, and relationships. SAS can connect to databases and automatically gather this catalog information. However, SAS (or any other metadata management tool) will not have any provenance information regarding the relationship between an application and its database unless it is manually entered.

One defining characteristic of Guava's query interface is that it is graphical. Both the syntax and expressive power of Guava's query interface are inspired by the graphical query language Query-By-Example (QBE) [87]. What Guava's query interface offers beyond these tools is integrated data dictionary (context) information, automatically provided from the UI, as well as a look and feel that is familiar to users of the application.

There are a variety of visual query languages whose visual metaphors reflects their respective underlying data models. GQL [61] is a graphical language with constructs that appear as repeated application of functions, since it operates on the functional data model. In Query By Diagram [4], the data model is the ER model, so queries in QBD look very similar to small ER diagrams. For XML data, the XGL language [24] offers a graphical interface that resembles small trees of nodes. Guava's query interface has a clear visual metaphor as well — a hierarchy of forms — that conforms to Guava's modified relational model.

Another defining characteristic of Guava's native query interface is that it is as tailored to a domain expert's needs as the original application. There are many query environments, visual or textual, that are designed to be domain-specific. Like Guava's query interface, these query environments are designed to be understandable to domain users. For example, NeuroQL [77] is a textual query language over a data model designed for neurological data. Queries in NeuroQL can refer to connections between neurons, molecular structures, and electrical patterns, all of which are concepts in the underlying data model and also comprehensible to a user knowledgable in neural biology.

2.7 SUMMARY

In this chapter, we have presented a way to formalize the data content in a forms-based user interface. These user interfaces have a natural relational representation, which we can automatically generate. The forms metaphor also lends itself to a QBE-type query interface; we described how a graphical version of this interface would appear, and demonstrated how we implemented a query interface with a tree-like look and feel. We demonstrated, using the CORI application, that converting an existing user interface to use Guava is feasible with reasonable programming effort.

Chapter 3

TRANSFORMATIONS AND THE CHANNEL

Chapter 2, among other things, outlines a method by which one can derive a relational schema called the natural schema from an application's user interface. In many situations, the database designers may prefer not to use the natural schema as the schema for the physical database. It is also possible that the application must use an existing database with a schema that is quite different in structure from the natural schema. In this chapter, we introduce a tool that connects the natural schema of an application to its physical schema; this tool is called a *channel*.

Our channel is, in a way, another solution to the classic view update problem [19]; namely, given a set of view definitions $\vec{V}$ (usually expressed in a query language), determine whether it is possible to translate update statements issued against $\vec{V}$ unambiguously into statements against the base schema such that, when the queries in $\vec{V}$ are re-executed, it appears to the user that the view schema was updated directly. Often, the view update problem is simplified using syntactic restrictions on the query definitions. In the SQL:1999 standard, and in most relational DBMS systems, a view is determined to be updatable if it is limited to using only certain syntactic features. Other approaches to solving the view update problem focus on identifying a subset of a query language (e.g., conjunctive queries) and focus on ways of providing proofs of view updatability for that subset, possibly with directives specified by a developer that give hints as to how

to remove ambiguity [9]. Generally, approaches to the view update problem focus on conjunctive queries, because unambiguously updating through joins, inner and outer, is a significant problem.

The thesis of this dissertation is that any forms-based software application with database access is already supporting a view of the data in that database. Moreover, if the application is largely a data-entry application (introduced in Chapter 2), that application already serves as an updatable view. What is interesting about this observation is that the relationship between an application's natural schema and its physical database can rarely be expressed using the limited expressive power of known updatable views. For instance, in the CORI application, the physical storage for clinical procedures is in a generic layout — in other words, as key-attribute-value triples. The relationship between data in this layout and data in a more standard, one-attribute-per-column layout is called a *pivot*. No current solution to the view update problem is expressive enough to handle pivots. In CORI's case, the pivot operation is hard-coded in the application middleware.

Our research addressed the view update problem in the opposite direction from the existing view-update literature. Rather than considering how to make views expressed in SQL updatable, we start with transformations that we know to be invertible, such as pivots, and allow the database designer to compose them to form a mapping from the natural schema to the desired physical schema. The result is a mapping language with significantly different expressive power from existing updatable view languages. A channel is a composition of these transformations. Queries, DML updates, or even schema changes issued by the application against the natural schema are pushed through

these transformations one at a time; at each step, the transformation takes each statement and translates it into an equivalent statement (or set of statements) that is valid over the transformation's output schema.

In this chapter, we make the following research contributions:

- We define an extension to the standard relational model that includes an extended relational algebra, a generalized notion of referential integrity, and incremental evolution of both foreign keys and column domains.

- We define a mapping language of transformations that is expressive enough to cover pivoting, unpivoting (the reverse transformation for pivoting), invertible function application, horizonal and vertical partitioning, and horizontal and vertical merging. The mapping language uses our extended relational model as a basis, and is closed under the operations that the extended model supports.

- We define the unambiguous action of each transformation on statements (queries, data updates, and schema updates) expressed against the transformation's input schema into a set of equivalent expressions against the transformation's output schema.

- We calculate the approximate effect of each transformation on the kinds of physical data characteristics utilized by contemporary database tuning tools, e.g., row counts and histograms of data value distributions (so as to allow channels to assist database designers in database optimization and physical design).

- We evaluate our mapping language by examining an existing database-backed application, and determining if our mapping language is expressive enough to

map the application's natural schema to its physical schema.

- We further evaluate the expressive power of our mapping language by comparing it against an alternative transformation language.

- We evaluate the performance of an implementation of channel transformations.

This chapter introduces transformations designed to handle physical design decisions. Chapter 4 will expand the menu of available transformations in several ways, including transformations intended for purposes other than physical design. One final note: We envision the channel as a general-purpose database transformation tool, not just a component of the Guava architecture. In other words, one can design a channel between two arbitrary database instance, not just between the Guava natural schema and its physical storage layer.

3.1   THE GUAVA DATA MODEL

The Guava GUI tools in Chapter 2 operate on the relational model, except that referential integrity (i.e., foreign keys) is extended to allow disjunction. In this chapter, we use a data model, the Guava Data Model (GDM), that is more expressive in terms of query language and available data and schema constructs than the model used in Chapter 2.

In this section, we outline the ways in which the GDM extends the relational model. In particular, we consider the expressiveness of the query language and the allowed DML and DDL operations, and we describe classes of integrity constraints expressed as primary and foreign keys.

3.1.1 Queries

A channel transforms queries expressed in extended relational algebra. The term "extended relational algebra" has been used to refer to a number of different languages in different settings. In general, extended relational algebra includes the standard eight-operator relational algebra (σ, π, $\times$, $\bowtie$, $\cup$, $\cap$, $-$, and $\div$), an optional rename operator (ρ), plus additional operators introduced to serve some specific need, such as aggregation. For our purposes, the operators we introduce beyond standard relational algebra are:

- Left outer join ($\sqsupset\!\bowtie$) and left antisemijoin ($\overline{\ltimes}$)

- Pivot ($\boxslash^{\vec{C}}_{C,A,V}$), for a set of values $\vec{C}$ on which to pivot, pivot column A, and pivot-value column V (translates a relation from a key-attribute-value triple form into a normalized, column-per-attribute form)

- Unpivot ($\boxslash_{\vec{C},A,V}$), the inverse operation to pivot

- Function application ($\alpha_{\vec{I},\vec{O},f}$), applying the function f on input columns $\vec{I}$ and placing the result in output columns $\vec{O}$

- Table and row constants

All of the operators above (except function application) can be expressed using standard relational algebra operators. However, much like the join operator, which can be expressed in terms of the relational algebra operators cross-product and select, each of these operators has associated algorithms that are significantly faster than the execution of an equivalent expression using standard operators. To demonstrate that the operators

above can be expressed in relational algebra, consider the following equivalence for the pivot operator:

$$\vec{\varnothing}_{\vec{C},A,V}Q \equiv (\pi_{columns(Q)-\{A,V\}}Q) \rhd\!\!\lhd (\rho_{V\to C_1}\pi_{columns(Q)-\{A\}}\sigma_{A=C_1}Q)$$

$$\rhd\!\!\lhd \ldots \rhd\!\!\lhd (\rho_{V\to C_n}\pi_{columns(Q)-\{A\}}\sigma_{A=C_n}Q) \text{ for } C_1,\ldots,C_n = \vec{C}$$

Likewise, the unpivot operator may be expressed as follows:

$$\varnothing_{\vec{C},A,V}Q \equiv \bigcup_{C\in\vec{C}}(\rho_{C\to V}\pi_{columns(Q)-(\vec{C}-\{C\})}\sigma_{C<>null}(Q) \times \rho_{1\to A}(name(C)))$$

Examples of these operators acting on instances of data can be found in Figure 3.6(b) (in Section 3.3). Evaluating either of these operators simply by running the equivalent query in relational algebra shown here would be very slow; for instance, pivot requires a great many outer joins, which are costly. Both pivot and unpivot have associated algorithms that can perform the same task in a single pass of the data, provided that the data is sorted on the key for the pivot operator. Therefore, unpivot can be evaluated in $O(n)$ I/O's (compared to $O(mn)$ I/O's for its union definition, where m is the number of pivot column), and pivot can be evaluated in $O(n\log n)$ I/O's, dominated by the cost of sorting the data (compared to $O(m(n\log n))$ for its left-outer-join definition). Commercial database systems include syntax in their proprietary dialect of the SQL language that can handle pivoting and unpivoting that are evaluated using these optimized algorithms [16, 53].

3.1.2 Updates to Data and Schema

The full list of update statements for our data model, both DML and DDL, is shown in Table 3.1. We support the three common DML statements (insert, update, and delete),

Table 3.1: The DML and DDL statements that channels support.

Statement	Explanation of Variables
Insert $I(T,\vec{C},Q)$	Table to insert rows into (T), columns of the table that will hold the values ($\vec{C}$), and values of $\vec{C}$ of the new rows (Q). Q may be a constant or a query result.
Update $U(T,\vec{F},\vec{C},Q)$	Table to update rows in (T), list of equality conditions on key columns (with "AND" semantics) to identify rows ($\vec{F}$), non-key columns of the table that will hold the new values ($\vec{C}$), and new values of the rows specified by query (or constant) (Q). Query Q may refer to the current (pre-update) row values as constants. Not all key columns need to have a condition.
Delete $D(T,\vec{F})$	Table to delete rows from (T) and list of conditions (with "AND" semantics) to identify rows ($\vec{F}$). Each condition in $\vec{F}$ is an equality condition on a key column. Not all key columns need to have a condition.
Add Table $AT(T,\vec{C},\vec{D},\vec{K})$	Name of the new table (T), names of the table's columns ($\vec{C}$), column domains ($\vec{D}$), and key columns ($\vec{K}$).
Rename Table $RT(T_o,T_n)$	Name of the old table (T_o) and new name for the table (T_n). Throws error if T_n causes a schema conflict.
Drop Table $DT(T)$	Name of the table to drop (T)
Add Column $AC(T,C,D)$	Name of the table (T), and the new column's name (C) and domain (D).
Rename Column $RC(T,C_o,C_n)$	Name of the table (T), the column's old name (C_o) and new name (C_n). Throws error if C_n causes a schema conflict.
Drop Column $DC(T,C)$	Name of the table (T) and the (non-key) column to drop (C).
Add Element $AE(T,C,E)$	Table (T) and column (C) whose domain is being edited, with the new domain value (E).
Rename Element $RE(T,C,E_o,E_n)$	Table (T) and column (C) whose domain is being edited, with the old and new domain values (E_o, E_n). Throws error if E_n conflicts with an existing element.
Drop Element $DE(T,C,E)$	Table (T) and column (C) whose domain is being edited, with the dropped domain value (E).

with the restriction that conditions in update and delete statements are equality conditions on key attributes. So, a channel will accept the statements `DELETE FROM T` and `DELETE FROM T WHERE A=1 AND B=3` for key columns A and B. A channel will not accept the statements `DELETE FROM T WHERE C=5` for non-key column C, or `DELETE FROM T WHERE A>2` for any column A, for example. We also do not allow updates on key attributes, assuming that the application will instead issue a delete followed by an insert if this is necessary, though the DDL Rename Element statement — introduced shortly — performs a similar task. These restrictions are simplifying assumptions for the benefit of simpler definitions of the channel transformations. All of the DML updates that are typically generated by a forms-based application satisfy these restrictions, since they will always update or delete rows based on the key of a selected or visible entity. One can mimic the action of an arbitrary conditions on an update or delete statement by using a query to retrieve all of the key values for rows that match the statement's conditions, then issue an update or delete for each of the qualifying rows using a loop construct (to be introduced in Section 3.1.4).

Our model also supports statements that add, rename, or drop tables; add, rename, or drop non-key columns; and add, rename, or drop domain elements — statements that incrementally evolve a schema. Statements that affect tables and columns are the same as in standard SQL. The domain element DDL statements are unique to our model and have some special semantics. If a domain element E in a non-key column C is dropped, then any row that had a C value of E will have that value set to null. However, if C is a key column, then any such row will be deleted. In addition, the Rename Element DDL statement will automatically update an old domain value to the new one. Since renaming

an element can happen on any column, key or non-key, renaming elements is a way to update key values. If an element is being added, renamed, or dropped from a column that participates in a foreign key, the system verifies that there is a companion statement in the current transaction modifying the other column participating in the foreign key, without which the system will throw an error.

3.1.3 Generalized Referential Integrity Constraints

In the GDM, we support three levels (or "tiers") of referential integrity. The three tiers offer a trade-off between expressive power and efficiency. Two of these tiers are strictly more expressive than relational referential integrity. We require additional expressive power with referential integrity because ordinary referential integrity constraints cannot be passed through our channel transformations and still be expressed as ordinary referential integrity constraints over the transformation's output.

Tier 1 Foreign Keys

A Tier 1 foreign key is a foreign key in the traditional relational model. A Tier 1 foreign key from a table T to a second table B is expressed as $FK(T.\vec{X} \rightarrow B.\vec{Y})$, where $\vec{Y}$ (a set of columns) is the primary key of B, and $\vec{X}$ is some set of columns in T with the same cardinality as $\vec{Y}$. If the foreign key is in place in a relational database, then the following logical expression is true:

$$\forall_{t \in T} t[X] \neq null \rightarrow \exists_{t' \in B} t[X] = t'[Y].$$

In plain English, every tuple t in table T must have a corresponding tuple b in table B where t's values for columns $\vec{X}$ are the values in the primary key columns of b, provided

that $t.\vec{X}$ contains a value. Figure 3.1(a) shows an example of a Tier 1 foreign key, and the statement used to define it.

Tier 2 Foreign Keys

A Tier 2 foreign key is a generalization of the standard relational foreign key in several ways. First of all, the foreign key can reference a subset of the key columns in the target table (called B previously). Second, the foreign key may reference a subset of rows in either the source or the target table; a standard foreign key refers to the entire source table (T) and the entire target table (B). Finally, we allow multiple foreign keys to be defined on the same source columns with disjunctive semantics. With standard relational foreign keys, one cannot declare more than one foreign key on the same set of source table columns because it creates ambiguity with respect to cascading updates and deletes. With Tier 2 foreign keys, one can express the notion, "the values in columns $\vec{X}$ must be found in $\vec{Y_1}$ OR $\vec{Y_2}$". This notion of disjunction in foreign keys subsumes the extended foreign keys introduced in Chapter 2.

A Tier 2 foreign key can be expressed as a sequence of statements, also called *foreign key fragments*:

$$FK(\vec{F}|T.\vec{X} \rightarrow \vec{G_1}|B_1.\vec{Y_1}), FK(\vec{F}|T.\vec{X} \rightarrow \vec{G_2}|B_2.\vec{Y_2}), \ldots, FK(\vec{F}|T.\vec{X} \rightarrow \vec{G_n}|B_n.\vec{Y_n})$$

where each $\vec{Y_i}$ is a (not necessarily proper) subset of the primary key columns of table B_i. $\vec{F}$ and each $\vec{G_i}$ are collections of conditions on key columns with AND semantics. The sequence above is defined to be equivalent to the following logical expression:

$$\forall_{t \in T} t \models \vec{F} \wedge t[X] \neq null \longrightarrow ((\exists_{t' \in B_1} t[X] = t'[Y_1] \wedge t' \models \vec{G_1})$$

$$\lor (\exists_{t' \in B_2} t[X] = t'[Y_2] \land t' \models \vec{G_2}) \lor \ldots \lor (\exists_{t' \in B_n} t[X] = t'[Y_n] \land t' \models \vec{G_n})).$$

Figure 3.1(b) shows an example of a Tier 2 foreign key enforced on table instances, and the statements used to create the foreign key.

Any Tier 1 foreign key can be expressed as a Tier 2 foreign key of the following form:

$$FK(true|T.\vec{X} \rightarrow true|B.\vec{Y})$$

where $\vec{Y}$ is the entire primary key for table B, and no other foreign key is expressed from columns $T.\vec{X}$. Conversely, any Tier 2 foreign key that can be expressed in this form is also a Tier 1 foreign key.

Any Tier 2 foreign key that cannot be expressed as Tier 1 also cannot be expressed as a standard foreign key in a relational database. However, a Tier 2 foreign key can be enforced in a standard relational database using triggers — specifically, insert and update triggers on the source table T and a delete trigger on each target table B_i. Translating a Tier 2 foreign key into triggers is straightforward. The following pseudo-code describes the algorithms for the triggers generated from the sample Tier 2 above:

```
begin insert trigger (T)
    if new tuple satisfies conditions F
        for each i in 1..n
            for each tuple t in Bi
                if t[Yi] = new tuple[X] and t satisfies Gi
                    accept insert
    reject insert
end trigger
```

```
(update trigger follows the same pattern as insert trigger)


begin delete trigger (Bi)
    if deleted tuple satisfies conditions Gi
        for each tuple t in T
            if t[X] = deleted tuple[Yi] and t satisfies F
                found = false
                for each j in 1..n
                    for each tuple t' in Bj
                        if t'[Yj] = t[X] and t satisfies Gj
                            found = true
                            break
                    if found
                        break
                if not found
                    delete tuple t
end trigger
```

The second trigger above is a generalized version of cascading deletes. When a row is deleted from table B_i, the trigger checks to find all tuples in T that match the deleted row. Then, for each matched row, the trigger checks all of the target tables of the foreign key to see if the row can match a new target. If not, the row is deleted as part of the cascade.

Tier 2 foreign keys are likely to have an efficient implementation because of the high

ChildTable:

ID	ColA	ColB	ColC
1	R	a	x
1	S	b	z

FK(true | ChildTable(ColB, ColC)
$\rightarrow$ true | ParentTable(ColB, ColC))

ParentTable:

ColB	ColC	ColD	Val
a	x	45	19
b	z	33	44
a	z	48	95

(a)

ChildTable:

ID	ColA	ColB	ColC
1	R	a	x
1	S	b	z
2	R	c	z

FK
(ID=1| ChildTable(ColB, ColC)
$\rightarrow$ true | ParentTable1(ColB, ColC))

FK
(ID=1| ChildTable(ColB, ColC)
$\rightarrow$ true | ParentTable2(ColF, ColG))

ParentTable1:

ColB	ColC	Prov	Val
a	x	N	19
d	y	M	44
a	x	M	52

ParentTable2:

ColF	ColG	Prov	Val
b	z	T	44

(b)

ChildTable:

ID	ColA	ColB	ColC
1	R	a	x
1	S	b	z

PivotTable:

ColB	x	y	z
a	19	NULL	95
b	NULL	NULL	44

Check ($\pi_{ColB,ColC}$ ChildTable $\subseteq \pi_{ColB,ColC} \bowtie\!\!\!\!/_{\{x,y,z\},ColC,Val}$ PivotTable)

(c)

Figure 3.1: Examples of Tier 1 (a), Tier 2 (b), and Tier 3 (c) foreign keys

probability of index usage. The worst-case scenario is that each of the tables T, B_1, ..., B_n must be scanned once. The best-case scenario is that there is an index on $T.\vec{X}$; if there are already indexes on all of the primary keys, which is highly probable, then the triggers may be able to operate using only index scans.

Since a single Tier 2 foreign key comprises a sequence of statements rather than a single statement, a Tier 2 foreign key can be constructed one statement at a time. For instance, one can issue the following statement:

$$FK(\vec{F}|T.\vec{X} \rightarrow \vec{G_1}|B_1.\vec{Y_1})$$

One can subsequently issue the following statement:

$$FK(\vec{F}|T.\vec{X} \rightarrow \vec{G_2}|B_2.\vec{Y_2})$$

The result effectively relaxes the foreign key, allowing values in $T.\vec{X}$ to be found in either table B_1 or table B_2 rather than simply table B_1.

In addition to incrementally adding new foreign key statements, one can also incrementally drop portions of a Tier 2 foreign key. The drop foreign key statement has the following syntax:

$$DFK(\vec{F}|T.\vec{X} \rightarrow \vec{G_1}|B_1.\vec{Y_1})$$

In our running example, after this statement is issued, the values in $T.\vec{X}$ must be found in table B_2, since values are no longer allowed to be found in B_1. The parameters to the alter and drop foreign key statements have identical syntax as the create foreign key statement.

Whenever a foreign key is updated by a DFK statement, the source table of the foreign key must be checked to see if any rows now violate the new foreign key definition.

Any rows in table T that satisfy conditions $\vec{F}$ but no longer satisfy the foreign key are transformed using the same semantics as the Drop Element statement mentioned in Section 3.1.2: If any of the columns in $T.\vec{X}$ (the foreign key columns of the source table) are key columns, the violating row is dropped; otherwise, the values of those columns are set to null.

Similar to referential integrity constraint checking in relational database systems, referential integrity checks occur at the end of the enclosing transaction. One can issue several foreign key fragments in the same transaction and not have referential integrity enforced until all statements have been processed. Therefore, one can simulate the action of "updating" a foreign key by adding and deleting fragments in the same transaction.

Tier 3 Foreign Keys

A Tier 3 foreign key is a generic containment constraint between two queries. One can express a Tier 3 foreign key as follows:

$$Check(Q_1 \subseteq Q_2)$$

where Q_1 and Q_2 are any queries expressed in extended relational algebra. A Tier 3 foreign key has clear semantics: At all points in time, the results of query Q_1 must be a subset of the results of query Q_2. Figure 3.1(c) gives an example of a Tier 3 foreign key, where the target of the foreign key is a pivoted table. The foreign key in the figure cannot be neatly drawn as arrows in the diagram as could be done with the other tiers of foreign keys, since the target data of the foreign key exists only in schema.

Just like Tier 2 foreign keys, one can specify multiple Tier 3 foreign key statements.

If multiple Tier 3 foreign keys share the same head query, they have disjunctive seman-
tics. In other words, the statement sequence:

$$Check(Q_1 \subseteq Q_2), Check(Q_1 \subseteq Q_3), Check(Q_1 \subseteq Q_4)$$

is defined to be equivalent to the statement:

$$Check(Q_1 \subseteq (Q_2 \cup Q_3 \cup Q_4))$$

Also similar to a Tier 2 foreign key, one can incrementally alter parts of the sequence
that defines the key. The statement that drops a Tier 3 component is:

$$DCheck(Q_1 \subseteq Q_2)$$

Any Tier 2 foreign key can be expressed as a Tier 3 foreign key. The statement:

$$FK(\vec{F}|T.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$$

is equivalent to the statement:

$$Check(\pi_{\vec{X}}\sigma_{\vec{F}}T \subseteq \pi_{\vec{Y}}\sigma_{\vec{G}}B)$$

Any Tier 3 foreign key that can be written as an expression in the form above can
also be expressed as a Tier 2 foreign key. If Tier 2 and Tier 3 foreign keys are defined
on the same set of source columns, the composition of the keys is treated as a Tier 3 key.
For example, if one issues the following statement sequence:

$$FK(\vec{F}|T.\vec{X} \rightarrow \vec{G}|B.\vec{Y}), Check(\pi_{\vec{X}}\sigma_{\vec{F}}T \subseteq \pi_{\vec{Z}}Q)$$

the result is equivalent to the statement:

$$Check(\pi_{\vec{X}}\sigma_{\vec{F}}T \subseteq ((\pi_{\vec{Y}}\sigma_{\vec{G}}B) \cup \pi_{\vec{Z}}Q))$$

Tier 3 foreign keys differ from Tier 2 foreign keys in two additional ways. First, while Tier 3 foreign keys can be implemented as triggers in relational database systems, Tier 3 foreign keys are far less efficient than Tier 2 foreign keys. Whereas Tier 2 foreign keys operate as single table scans at worst or index scans at best, Tier 3 keys may involve arbitrarily complicated relational algebra queries and may incur a high cost in terms of disk access and frequency of trigger firing. For instance, the insert trigger for a Tier 3 foreign key must fire each time a tuple is inserted into any of the tables referenced by the foreign key's head query.

The final difference between Tier 3 and Tier 2 keys is the effect of incremental modifications on data instances. Like Tier 2 foreign keys, if a fragment of a Tier 3 foreign key definition is dropped, the applicable source instance for the foreign key must be checked to make sure it is still valid. However, since the source of a Tier 3 foreign key Q_1 is potentially an arbitrary query, it may not be possible to identify the specific rows of the source instance that cause the foreign key violation if one occurs. Therefore, if query Q_1 is of the same form as the source of a Tier 2 foreign key ($\pi_{\vec{C}}\sigma_{\vec{F}}T$ for columns $\vec{C}$, equality conditions $\vec{F}$, and table T), then violating rows are updated in the same way as a Tier 2 foreign key. Otherwise, the system issues an error that referential integrity is violated and aborts the statement, a similar severity of error as an ordinary referential integrity violation.

DDL Statements and Their Effect on Foreign Key Fragments

Whenever DDL statements alter schema elements, the foreign key fragments that reference the altered elements must in turn be altered. For instance, for any schema renaming

statement (table, column, or element), all references to the renamed object in the foreign key fragments defined over the schema must be updated. For instance, if the fragment $FK(\vec{F}|S.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ is defined on a schema, and the statement $RT(S, S')$ is executed on the database to rename the table S to S', the fragment must change to be defined as $FK(\vec{F}|S'.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$. We can accomplish this effect by dropping the old fragment and adding the new one within the same transaction.

Dropping schema elements has a more profound impact on foreign key fragments. Foreign key fragments must be altered so that all references to the dropped elements are also eliminated, even if that means the fragment itself is dropped. For instance, consider the Drop Table statement $DT(T)$. Any Tier 2 foreign key fragment $FK(\vec{F}|S.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ will be dropped if the table $S = T$ or if the table $B = T$. For a Tier 3 foreign key fragment $Check(Q_1 \subseteq Q_2)$, all references to the table T in Q_1 and Q_2 are replaced with a constant representing an empty table; if either Q_1 or Q_2 becomes equivalent to $\mathcal{N}$ (the query that always returns an empty table) as determined by using relational equivalences, we drop the fragment.

The Drop Column DDL statement removes all references to the column from foreign key fragments. There are three cases to consider for the statement $DC(T, C)$:

- For each Tier 2 foreign key fragment $FK(\vec{F}|S.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ where $S = T$ and $\vec{X} = \{C\}$, drop the fragment, as all of the columns of the foreign key have been dropped. We need not check to see if the dropped column is in the target of the fragment, since key columns cannot be dropped in our framework.

- For each Tier 2 foreign key fragment $FK(\vec{F}|S.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ where $S = T$ and $C \in \vec{X}$, $|\vec{X}| > 1$, if C' is the name of column that C references in $B.\vec{Y}$, replace the

fragment with $FK(\vec{F}|S.\vec{X'} \rightarrow \vec{G}|B.\vec{Y'})$, where $\vec{X'} = \vec{X} - \{C\}$ and $\vec{Y'} = \vec{Y} - \{C'\}$. In essence, remove the column from both the source and target of the fragment. Again, we need not check the target of the fragment.

- For each Tier 3 foreign key fragment $Check(Q_1 \subseteq Q_2)$, replace all instances of table T in both Q_1 and Q_2 with $\pi_{\vec{C}}T$, where $\vec{C}$ is all of the columns of T after removing C. If either query becomes invalid because of an operator referencing an invalid column or any subexpression of the query having a result with zero columns (determined using relational equivalences), we throw an error, alerting the system that a Tier 3 constraint definition would be invalid if the statement commits and identifying the offending fragment. If, instead, either Q_1 or Q_2 becomes equivalent to $\mathcal{N}$ as a result, we drop the fragment. If queries Q_1 and Q_2 are no longer union-compatible, for each column that was eliminated from Q_1, introduce a projection operator to Q_2 to remove the corresponding column by position in the query result, and vice versa.

The Drop Element DDL statement eliminates any equality references to that element in foreign key fragments. There are two cases to be considered for the statement $DE(T, C, E)$:

- For each Tier 2 foreign key fragment $FK(\vec{F}|S.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$, if $S = T$ and there exists an equality condition $< C, E > \in \vec{F}$, or $B = T$ and there exists an equality condition $< C, E > \in \vec{G}$, we drop the fragment.

- For each Tier 3 foreign key fragment $Check(Q_1 \subseteq Q_2)$, replace all instances of table T in both Q_1 and Q_2 with $\sigma_{C \neq E}T$. If either query becomes equivalent to $\mathcal{N}$ as

a result (again, determined using relational equivalences), we drop the fragment.

We assume that some entity will interpret DDL statements when they arrive at the database and automatically adjust the set of foreign key fragments accordingly. Such an entity may be either the database management system, or some interface above it such as a provider, introduced in Section 3.2.4.

3.1.4 Additional Statements

Our model supports an error statement $Error(Q)$. The error statement throws an error and aborts the current transaction if the query Q returns any rows; in other words, the query Q searches for "bad" rows or rows that may cause some sort of conflict with statements that may occur after the error check.

Also, our model supports a looping construct, denoted as $Loop(t, Q, \vec{S})$. The semantics of this statement is similar to a cursor: t is declared as a row variable that loops through the rows of the result of Q; for each value t takes on, the sequence of statements $\vec{S}$ execute. Statements in $\vec{S}$ may be any of the statements shown in Table 3.1, an error statement, or another loop construct, and may use the variable t as a row constant.

3.2 THE CHANNEL

In this section, we introduce the channel, the artifact in Guava that handles data and schema transformation between the natural schema and the physical schema.

Definition: A *channel transformation* is a mapping between two database schemas D and D' in our extended relational model, associated with a collection of translations

that convert statements posed against D into statements against D'. The statement translation algorithms for a transformation T must be *information preserving*, meaning that they must satisfy the following properties:

- The statements supported by the GDM are closed under T, meaning that when T translates a GDM statement, the resulting statements are also in the GDM, as described in the previous section.

- If one considers fully-materialized valid instances of both database D and database D', where D is materialized by issuing the query "SELECT * FROM A" through the channel for each table A in the natural schema, then when translating the query Q against D into a query Q' against D', the result of executing Q' against D' will always be equal to the result of executing Q against D ($Q(D) = Q'(D')$).

- For any table A in the original schema, if one pushes the query "SELECT * FROM A" through the channel to get result R, and then pushes a DML or DDL update through the channel, and finally pushes the same query through the channel to get result R', the differences between R and R' are exactly those that correspond to the update (as if the update had been executed against the query result directly).

This last condition is a round-tripping condition; Figure 3.2(a) shows its commutativity diagram for transformation T, table A, single-table query Q_A = "SELECT * FROM A", and DML or DDL statement Δ. Figure 3.2(b) describes the effect of the round-trip condition in informal terms. The condition ensures that no updates to the data are ever lost by the channel, and that there are no unforseen side effects. Note that the condition does not specify that the update pushed through the channel is to table A;

(a)

(b)
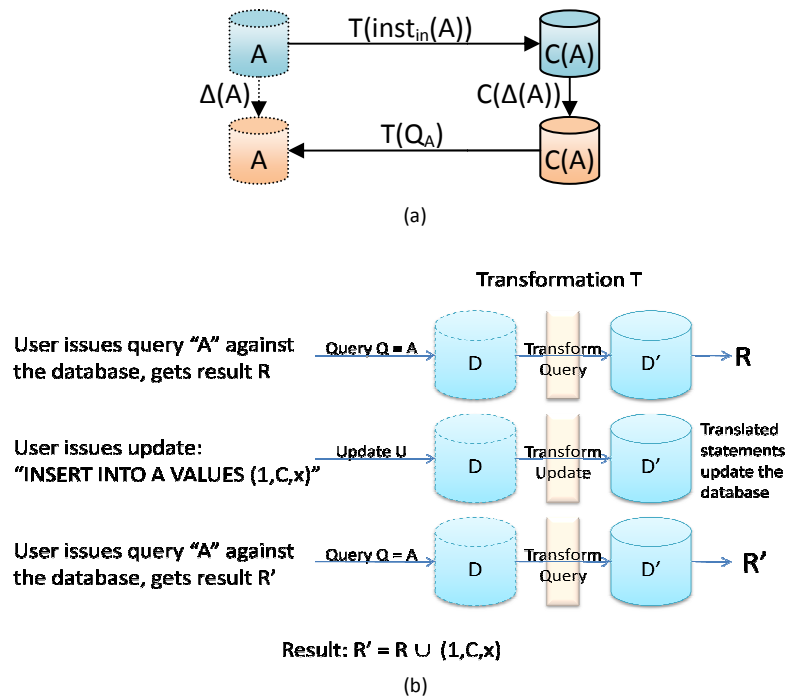
Figure 3.2: The commuting diagram for the information preservation properties of transformation T operating on table A (a), and an example of the channel transformation round-tripping condition (b)

any update to table *B*, $A \neq B$, will have no effect on the results of the single-table query *A* (unless effects are expected as a result of a cascading delete). In this way, the information preservation property is much like the concept of *constant complement* in view update literature [5]; a relational view *V* satisfies the constant complement property if, whenever the view is updated, any other view *V'* whose data is mutually-exclusive with *V* will remain unchanged. In Chapter 5, we formalize these properties and prove that the channel transformations introduced in this chapter are information preserving.

A *channel* is a sequence of channel transformations $[T_1, T_2, T_3, ...]$ with composition semantics. Because channel transformations satisfy the information-preservation properties, any channel with multiple transformations will also be information preserving by induction. The order of transformations is significant, given the sequential operation of transformations on statements, so channels $[T_1, T_2]$ and $[T_2, T_1]$ are not necessarily the same, though they may have identical results on identical input. We define a notion of equivalence on channels in Section 3.3.2.

Figure 3.2(a) shows an example of a channel with six transformations. Each of the transformations in the channel is parametrized to operate on specific schema elements, e.g., tables and columns (the parameters are not shown in the figure, but must be specified for each instance of a transformation). For instance, the first Horizontal Merge transformation in the channel in the figure operates on tables T1, T2, and T3, while the second Horizontal Merge operates on T4 and T5. The database developer specifies these parameters when designing a channel. Though a channel is a sequence of operators, one may think about a channel in a more graphical form based on its parametrization. Figure 3.2(b) is the same channel as the one shown in Figure 3.2(a) shown in a more

user-friendly way that is similar to a workflow display seen in graphical ETL tools, such as SQL Server Integration Services [52]. It is also true that Figure 3.2(a) is a serialized representation of Figure 3.2(b) (one of several possible serializations).

One way to define and use channel transformations is to take a database instance as input, transform it, and produce a database instance as output, which would be a similar workflow to ETL. Instead, we use the channel dynamically. At run time, the developer, query writer, or application is able to use the natural schema as if it were the actual database instance. The channel then translates each query, insert, update, delete, or schema modification that addresses the natural schema into a corresponding set of operations that address the physical database (Figure 3.4). Thus, the channel supports the natural schema as a virtual database, just like a traditional view is a virtual table defined over the existing tables in the physical database.

A channel is, in a way, a view defined backwards. One starts with the schema that the application or the user sees (e.g., the natural schema in Guava), and applies transformations one at a time until the desired physical schema is achieved. The physical schema may belong to a pre-existing database instance, or the physical database might not exist yet, in which case the channel would create a default instance with the appropriate schema. Specifying channel transformations that will be applied to the natural schema in order is not the only way to build channels, but it explains the naming conventions of the transformations. For instance, Horizontal Merge describes a horizontal merging of tables from the natural schema into a table in the physical schema, not the other way around.
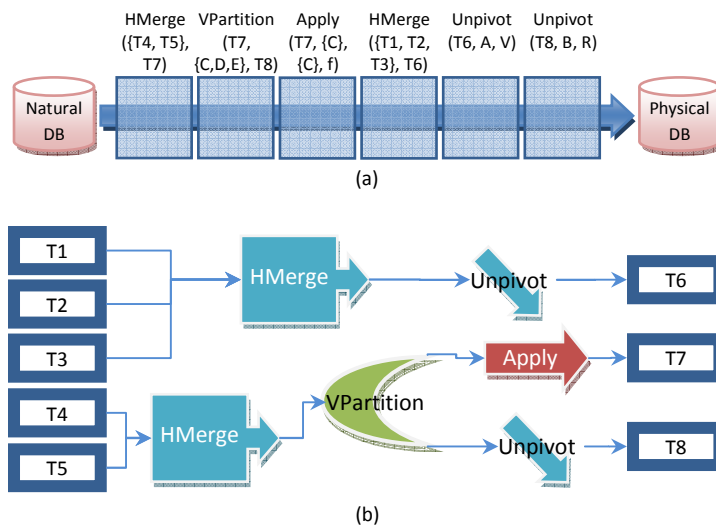
Figure 3.3: An example of a channel with six transformations (a), and a graphical representation of the same channel (b)
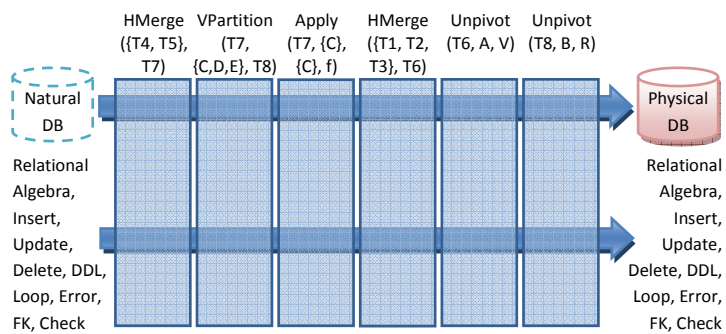
Figure 3.4: The same channel as in Figure 3.3(a), but when the natural schema is a virtual instance (i.e., a view over the physical database)

3.2.1 Seven Channel Transformations

Our language of seven transformations includes pivots, unpivots, function application, and the partitioning and merging operations typically found in physical database design [1] and translations of object class hierarchies to relations [59]. Our seven physical design transformations are listed in Table 3.2, with a short description of each. Informally, we can describe each transformation in terms of how it acts on a concrete instance of the transformation's input schema, as follows:

- $Apply(T_a, \vec{C}_{in}, \vec{C}_{out}, f)$, which applies an invertible function f to each of the rows in the table T_a. The function input is taken from columns $\vec{C}_{in}$, and output is placed in columns $\vec{C}_{out}$. We show an example of the Apply transformation applied to a concrete instance in Figure 3.5(a).

- $VPartition(T_a, \vec{C}s, T_n)$, which distributes the columns of table T_a into two tables, T_a and T_n. The columns in $\vec{C}s$ are the ones that stay in T_a. We show an example of the VPartition transformation applied to a concrete instance in Figure 3.5(b), reading from left to right.

- $VMerge(T_a, T_n)$, which vertically merges two tables T_a and T_n related by a one-to-one foreign key. We show an example of the VMerge transformation applied to a concrete instance in Figure 3.5(b), reading from right to left (VMerge and VPartition are inverses of one another on our sample instance).

- $HPartition(T_a, C_{in})$, which horizontally partitions the table T_a based on the valued in column C_{in}. We show an example of the HPartition transformation applied to a concrete instance in Figure 3.6(a), reading from left to right.

Table 3.2: Seven channel transformations, their descriptions, and their effect on relational queries.

Transformation	Description of Transformation	Effect on Query
VPartition $(T_a, \vec{C_s}, T_n)$	Vertically partition a table T_a into two tables T_a and T_n. Leave columns $\vec{C_s}$ in T_a, place all others in T_n. Copy key columns of T into both T_a and T_n.	$T_a \implies T_a \;\supset\!\!\!\bowtie\; T_n$
VMerge (T_a, T_n)	Vertically join two tables T_a and T_n along a foreign key. The foreign key must be from T_n to T_a on like-named columns and must be unconditional, but may be partial. The foreign key columns of T_n must be exactly the key columns of T_n. The resulting table is T_a.	$T_a \implies \pi_{\mathbf{Cols}(T_a)} T_a$ $T_n \implies \pi_{\mathbf{Cols}(T_n)} \sigma_{NN} T_a$ (NN is the condition that at least one value in $(\mathbf{Cols}(T_n) - \mathbf{Keys}(T_n))$ is non-null) Optimized: $T_a \;\supset\!\!\!\bowtie\; T_n \implies T_a$ Optimized: $T_a \bowtie T_n \implies \sigma_{(\mathbf{Cols}(T_n) - \mathbf{Keys}(T_n)) \neq null} T_a$
HPartition (T_a, C_{in})	Horizontally partition a table T_a into a collection of tables. The destination of each row of T_a is determined by its C_{in} value (C_{in} is a part of the key).	$T_a \implies \cup_{t \in \mathbf{Dom}(C_{in})}(t \times \{name(t)\})$ Optimized: $\sigma_{C_{in}=T} T_a \implies T \times \{name(T)\}$
HMerge $(\vec{T_s}, T_a, C_{out})$	Take the outer union T_a of a collection of tables $\vec{T_s}$, and record each row's table of origin in column C_{out}. Tables in $\vec{T_s}$ must have union-compatible key columns, and like-named columns must have the same domain.	$T(T \in \vec{T_s}) \implies \pi_{\mathbf{Cols}(T)} \sigma_{C_{out}=T} T_a$
Apply $(T_a, \vec{C_{in}}, \vec{C_{out}}, f)$	Apply an invertible function f to the $\vec{C_{in}}$ columns of each row of table T_a, placing the result in the columns $\vec{C_{out}}$ and eliminating columns C_{in}. $\vec{C_{in}}$ must be non-key attributes.	$T_a \implies \alpha_{C_{out}, C_{in}, f^{-1}} T_a$ Optimized: $\pi_{\vec{C_s}} T_a(\vec{C_s} \cap \vec{C_{in}} = \varnothing) \implies \pi_{\vec{C_s}} T_a$
Pivot (T_a, A, V)	Pivot a table T_a out of Key-Attribute-Value triples, resulting in one row per key value. Take column names from column A and values from column V.	$T_a \implies \boxtimes_{\mathbf{Dom}(A),A,V} T_a$
Unpivot (T_a, A, V)	Translate table T_a into Key-Attribute-Value form, with attribute column A and value column V. Non-key columns of T_a must have the same domain.	$T_a \implies \boxtimes_{\mathbf{Cols}(T_a)-\mathbf{Keys}(T_a),A,V} T_a$

- *HMerge*($\vec{T}s, T_a, C_{out}$), which horizontally merges the union-compatible tables in the set $\vec{T}s$ into a new table T_a, adding a new column C_{out} that holds the name of the table that each row came from. We show an example of the HMerge transformation applied to a concrete instance in Figure 3.6(a), reading from right to left (HMerge and HPartition are inverses of one another on our sample instance).

- *Unpivot*(T_a, A, V), which transforms a table T_a from a standard one column per attribute form into a key-attribute-value generic form, effectively moving column names into data values in new column A. Data values are placed in column V. We show an example of the Unpivot transformation applied to a concrete instance in Figure 3.6(b), reading from left to right.

- *Pivot*(T_a, A, V), which transforms a table T_a that is in generic key-attribute-value form into a form with one column per attribute. We show an example of the Pivot transformation applied to a concrete instance in Figure 3.6(b), reading from right to left (Unpivot and Pivot are inverses of one another on our sample instance).

Table 3.2 also describes how each transformation operates on queries as they are passed through the channel. A query posed against the channel's input schema is pushed through the channel to produce an equivalent query against the output schema. Each transformation translates a query in a similar fashion to view unfolding — the transformation looks for all references to tables in the query and translates them in-place as necessary [29]. For instance, the transformation HPartition(T_a, C_{in}) looks for all instances of table T_a in the query and replaces each with the expression $\cup_{t \in \textbf{Dom}(C_{in})}(t \times \{name(t)\})$. This expression rebuilds the original table T from the data in the output schema of the

transformation.

To simplify expressions and possibly save the DBMS's query optimizer some effort, each transformation also looks for common patterns in the relational algebra expressions. For example, if HPartition(T_a, C_{in}) finds both a reference to T_a and the operator $\sigma_{C_{in}=T}$ above it in the query tree, it will apply an equivalence on the fly and produce $T \times \{name(T)\}$ (where T is the same value found in the σ operator). Table 3.2 shows how each transformation translates references to tables in a query. The table also shows several algebraic equivalences by showing the patterns the transformations look for, and what each pattern is replaced with when found.

Tables 3.3 through 3.9 fully define the action of each transformation on each supported statement (how each transformation works on queries was described in Table 3.2). The tables use the following notation and terminology:

- Any function in **boldface** returns a list

- $\mathcal{T}$ refers to the current transaction

- **Cols**(T) refers to the set of columns of table T; if T exists both before and after the transformation, this expression refers to the "before" state

- **Keys**(T) refers to the key columns of table T

- **Domains**(T) refers to the domains of the columns of table T

- **Dom**(C) refers to the domain of the column with name C

- **Inputdomains**(f) and **Outputdomains**(f) refer to the set of domains and codomains respectively of the function f

Patient:

ID	Name	HeightInch	Country
1	Bob	72.0	USA
2	Ted	63.0	Canada
3	Sally	66.0	USA

Patient:

ID	Name	HeightFeet	Country
1	Bob	6.0	USA
2	Ted	5.25	Canada
3	Sally	5.5	USA

Apply (Patient, {HeightInch}, {HeightFeet}, f(x) = x/12)

(a)

Patient:

ID	Name
1	Bob
2	Ted
3	Sally

Patient:

ID	Name	HeightInch	Country
1	Bob	72.0	USA
2	Ted	63.0	Canada
3	Sally	NULL	NULL

FK

PatientDetails:

ID	HeightInch	Country
1	72.0	USA
2	63.0	Canada

VPartition (Patient, {Name}, PatientDetails)

VMerge (Patient, PatientDetails)

(b)

Figure 3.5: An example of the Apply (a), Vertical Partition (b), and Vertical Merge (b, in the reverse direction) transformations acting on concrete instances

Patient:

ID	Country	Name	HeightInch
1	USA	Bob	72.0
2	Canada	Ted	63.0

USA:

ID	Name	HeightFeet
1	Bob	6.0

Canada:

ID	Name	HeightFeet
2	Ted	5.25

HPartition (Patient, Country)

HMerge ({USA, Canada}, Patient, Country)

(a)

Patient:

ID	Name	City	Country
1	Bob	Boston	USA
2	Ted	NULL	Canada

Patient:

ID	Attr	Value
1	Name	Bob
1	City	Boston
1	Country	USA
2	Name	Ted
2	Country	Canada

Unpivot (Patient, Attr, Value)

Pivot (Patient, Attr, Value)

(b)

Patient:

ID	Attr	Value
1	Name	Bob
1	City	Boston
1	Country	USA
2	Name	Ted
2	Country	Canada
2	City	Toronto
3	Name	Sally

Patient:

ID	Name	City	Country
1	Bob	Boston	USA
2	Ted	Toronto	Canada
3	Sally	NULL	NULL

(c)

Figure 3.6: An example of the Horizontal Partition and Merge transformations (a), the Pivot and Unpivot transformations (b) acting on concrete instances, and an example of pushing inserts through a Pivot (c)

- $< c, v >$ refers to a condition for update or delete: that the value in the column with name c equals value v (a shorthand used occasionally in the tables is to represent multiple conditions at once by having multiple columns in c and values in v, with "AND" semantics)

- *col(D)* refers to the column name associated with domain D (we assume that each domain is augmented with an idea of identity when associated with a column in a table, in case multiple columns have identical domains, so that the *col* function is well defined)

- *name(e)* represents the name of a column, table, or element

- *AdjustForeignKeys(T,C,E)*, which represents the output of running Algorithm 3.1 (algorithm discussed and defined momentarily)

Not all supported statements appear in the tables that define the action of a transformation. If a statement S does not appear in a table for transformation T, there are four possible explanations:

- T does not have any effect on statement S — in other words, S passes through transformation T unaffected.

- S is a drop foreign-key statement DFK. These statements are not included because transformations operate on DFK statements in the identical fashion as FK statements. If the output of a transformation definition includes a *Check* statement, that statement is replaced with a *DCheck* statement.

- S is a Tier 3 foreign key statement $Check(Q_1 \subseteq Q_2)$, a Tier 3 drop foreign key statement $DCheck(Q_1 \subseteq Q_2)$, an Error statement $Error(Q)$, or a Loop statement $Loop(t, Q, \vec{S})$. The transformation simply operates in-place on the parameters of each statement (e.g., Q_1, Q_2, Q, and $\vec{S}$) because T already knows how to operate on queries and statements.

This list of transformations is not complete in the sense that there are other possible transformations that are information preserving. Here is one example: consider the *rotate* transformation, which re-assigns the keys in a table. Specifically, the transformation considers the rows in a table as an ordered list, ordered by the key of the table. The transformation takes the k'th row of the input, for all possible k, and produces a row in the output with the key of row k and the data of row $k + 1$ modulo the row count. In other words, the rotate transformation rotates the keys of a table by one place. One can define the action of the rotate transformation on each of the supported statements unambiguously. The transformation is also closed under the supported statements, and is information preserving. However, this transformation is rarely (if ever) used in practical software applications. We discuss the process a developer can use to create new, application-specific channel transformations in Chapter 4.

Element-Level DDL Statements and Foreign Key Evolution

In Tables 3.5 and 3.8, the translation of Add Element statements includes the statement *AdjustForeignKeys(T,C,E)*. The responsibilities of *AdjustForeignKeys(T,C,E)* are to re-transform the referential integrity constraints that are defined on the transformation's input schema, determine if there has been a change to the constraints on the output

schema, and generate the appropriate statements (*FK*, *DFK*, *Check*, or *DCheck*) as necessary. These foreign key adjustments are required whenever a channel transformation translates an Add Element DDL statement into a column-level or table-level DDL statement, because these transformations will also translate foreign key fragments in such a way that the output fragments are dependent on the domain of an input column. For the seven channel transformations defined in this chapter, Horizontal Partition and Pivot are the only transformations with this property and thus the only transformations that require this check.

For example, consider the channel transformation $O = HPartition(T_a, C_{in})$, input schema $T_a(\underline{id}, C_{in}, A, B)$, and foreign key fragment $\mathcal{F} = FK(true|S.fk \rightarrow true|T_a.id)$. If the domain of column C_{in} is $\{X, Y\}$, then according to Table 3.5, the result of applying O to $\mathcal{F}$ is two foreign key fragments $FK(true|S.fk \rightarrow true|X.id)$ and $FK(true|S.fk \rightarrow true|Y.id)$ on the output schema. Note that the number of fragments in the output is the same as the number of elements in the domain of C_{in}, supporting the intuition that changing the number of elements in the domain will change the number of fragments in the output.

If we subsequently add a new element Z to the domain of the input column $T_a.C_{in}$ using the statement $AE(T_a, C_{in}, Z)$, we recognize that, if we were to push the same foreign key fragment $\mathcal{F}$ through O again, the result would now be three fragments in the output schema: $FK(true|S.fk \rightarrow true|X.id)$, $FK(true|S.fk \rightarrow true|Y.id)$, and $FK(true|S.fk \rightarrow true|Z.id)$.

The net effect of the Add Element statement with respect to the foreign key fragments in the output schema is to add a new fragment, $\mathcal{F}_{out} = FK(true|S.fk \rightarrow true|Z.id)$,

which is the difference between the sets of generated foreign key fragments before and after the Add Element has been processed. Even though fragment $\mathcal{F}$ did not change in the input schema, the transformation O must add the new fragment $\mathcal{F}_{out}$ to the output as part of the processing of $AE(T_a, C_{in}, Z)$.

Algorithm 3.1: To evaluate *AdjustForeignKeys(T,C,E))*, for each Tier 1 or Tier 2 foreign key fragment $\mathcal{F} = FK(\vec{F}|S.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ defined over the transformation's input schema:

- If $S \neq T$ and $B \neq T$, ignore $\mathcal{F}$ and move on to the next fragment. (The transformation does not affect $\mathcal{F}$.)

- If $S = T$ and there exists some equality condition $< C, v > \in \vec{F}$, $v \neq E$, or if $B = T$ and there exists some equality condition $< C, v > \in \vec{G}$, $v \neq E$, ignore $\mathcal{F}$ and move on to the next fragment. (The transformation may affect this fragment, but the equality condition means that the fragment's translation by the transformation is unaffected by the addition of a new domain element.)

- Otherwise, re-translate $\mathcal{F}$ through the current channel transformation both pre- and post-insertion of the new domain element. Compare the two outputs, and create the appropriate foreign key fragment modification statements against the output to account for the differences.

For each Tier 3 foreign key fragment $\mathcal{F} = Check(Q_1 \subseteq Q_2)$ defined over the the transformation's input schema:

- If neither query Q_1 nor query Q_2 contains a reference to table T, ignore $\mathcal{F}$ and move on to the next fragment. (The transformation does not affect $\mathcal{F}$.)

- If $\sigma_{C=E}Q_1 \equiv \mathcal{N}$ or $\sigma_{C=E}Q_2 \equiv \mathcal{N}$ (where $\mathcal{N}$ is the query whose result is always empty), determined by using relational algebra equivalences, ignore the fragment and move on to the next one. This case will most commonly happen if there is another operator $\sigma_{C=E'}$ somewhere else in the query tree, $E \neq E'$. (The transformation may affect this fragment, but the fragment's translation by the transformation is unaffected by the addition of a new domain element.)

- Otherwise, re-translate $\mathcal{F}$ through the current channel transformation both pre- and post-insertion of the new domain element. Compare the two outputs, and create enough foreign key fragment modification statements against the output to account for the differences. □

In the previous example, the statement $\mathcal{F}_{out} = FK(true|S.fk \rightarrow true|Z.id)$ is the difference between the set of fragments before the element translation and the set of fragments afterward. Therefore, $\mathcal{F}_{out}$ is the result of running Algorithm 3.1 in this case.

In the remainder of this section, we explain the formalism of three of the cells in Tables 3.3 through 3.9.

Example: Rename Column pushed through the HMerge transformation

Consider the Rename Column statement $RC(T, C_o, C_n)$, which renames the C_o column of table T to the new name C_n. The $HMerge(\vec{T}s, T_a, C_{out})$ transformation (Table 3.6) assumes that if two tables in $\vec{T}s$ have columns with the same name, then those two columns also have the same domain and will be merged into the same column in the output table T_a. So, when a rename statement is passed through the HMerge, there are two pieces of information to gather: whether any of the tables $\vec{T}s - T$ have a column

with the old name C_o, and whether any tables in $\vec{T}s - T$ have a column with the new name C_n. The answer to either question may be yes or no, so there are four cases to consider.

Case 1: None of $\vec{T}s - \{T\}$ has columns called C_o or C_n. In this case, simply rename the column, since there are no other side effects.

Case 2: None of $\vec{T}s - \{T\}$ has a column called C_o, but at least one other table in $\vec{T}s - T$ has a column called C_n. This situation means that there is already a column in the output called C_n, so we cannot just rename the existing column. So, we move all of the data from the old column C_o to the new one with an update statement. Also, because no other table has a column called C_o, we can drop it from the result.

Case 3: None of $\vec{T}s - \{T\}$ has a column called C_n, but at least one other table in $\vec{T}s - T$ has a column called C_o. We need to add a new column C_n to the output, since one did not exist before. Then, move the data from C_o to C_n for rows that came from T. Do not move any data that comes from other tables. We also do not drop the old column C_o, since at least one table in the merge has active data in it.

Case 4: At least one table in $\vec{T}s - \{T\}$ has a column called C_o, and at least one other table in $\vec{T}s - T$ has a column called C_n. In this case, move the data in C_o over to C_n for those rows that came from T with an update statement and set the old values to null. We do not need to add a new column (since it already existed), and we do not need to drop a column afterward (since at least one other merged table uses it).

Example: Insert pushed through the Pivot transformation

In this section, we consider the insert statement processing of Table 3.8. Each row to be inserted as part of an insert statement that is pushed through a pivot effectively consists of an address to a field in the pivoted table, along with its new value. In other words, the pre-pivoted row $< K, A, V >$ that is inserted consists of a key value, an attribute value, and a data value; the key value uniquely identifies a row in the pivoted table, and the attribute value specifies the column in the pivoted table. The only difficulty is that it is not known whether the row exists in the pivoted table yet or not; so, it is unclear whether we need to update an existing row with a new value or create a new row. So, we separate the new rows into two parts: those that correspond to existing rows and those that do not. We use a join against the existing data in the database to find the inserted rows that match existing rows in the pivoted output table, and an antisemijoin to find the inserted rows that correspond to pivoted rows that do not yet exist. Figure 3.6(c) shows an example of pushing inserts that correspond to both an existing row and a new row through a Pivot transformation.

For the inserted rows that correspond to pivoted rows that already exist, we turn the insert statement into a sequence of updates. Each inserted row $< K, A, V >$ becomes an update statement: update the attribute A in the existing row corresponding to key value K to the new value V. We use a *Loop* statement to perform the iteration through each newly inserted row. However, there is a subtle problem: Because the inserted rows become updates, we must check for primary key violations manually, as evidenced by the following example.

Consider adding a new row to the table $T(\underline{K, A}, V)$ with data $(1, C, x)$. If this table

were stored in a physical database, there would be a primary key enforced on columns K and A; so, if row $(1, C, y)$ already existed in the table, the DBMS would throw an error.

For all transformations other than Pivot, there is no problem translating this insert statement. Consider the case where table T is subject to an HSplit transformation using column A as the splitting column. If we add $(1, C, x)$ to the pre-channel image of T, the channel will process the tuple and try to add $(1, x)$ to the post-channel image of table C. If $(1, C, y)$ already existed in table T in the pre-image, $(1, y)$ already exists in the post-channel image of C, and the primary key violation will still be thrown. For all operators except Pivot, we can safely rely on the existing primary key enforcement on the post-channel image to enforce the primary key semantics of the natural schema.

Now consider the case where table $T(\underline{K, A}, V)$, rather than being pushed through an HSplit, is transformed by a Pivot, using column A as the attribute column and V as the value column. The existing row $(1, C, y)$ corresponds to a row in the pivoted table with key 1 and whose C value is y. If we try to insert $(1, C, x)$ into the pre-pivot image, $x \neq y$, the post-pivot equivalent is to update the row with key value 1 and set its C value to x. No error is thrown. However, the primary key semantics of the natural schema would not be enforced. Therefore, we add an error statement to the translation of inserts. The error check tests to see if there are any values in common between the new rows and the existing rows in the pivot table, and if so, returns an error.

For the inserted rows that do not yet exist, we simply take that set of rows, pivot them, and insert them into the table.

Example: Add Element pushed through the Unpivot transformation

The Unpivot transformation (Table 3.9) requires that all of the non-key columns be union-compatible, i.e., that they have the same domain (so that their values may be stored in the same column as a result of the Unpivot). If an Add Element statement is issued to extend the domain of one of the non-key columns, this would cause union compatibility to fail and throw an error. However, there is one situation in which there does not need to be an error thrown. If, within the same transaction $\mathcal{T}$ there are Add Element statements for all of the other non-key columns, then union compatibility is maintained; a single Add Element statement is placed in the output to change the domain of the single, combined value column.

3.2.2 Transactions and Transactional Semantics

Statements pass through the channel as part of a transaction.

Definition: A *transaction* in Guava is a sequence $\{s_0, s_1, \ldots, s_k\}$ where each s_i is one of the following:

- A query expressed in extended relational algebra

- One of the statements shown in Table 3.1 with its parameters filled in

- One of the foreign key statements (FK, DFK, Check, DCheck), introduced in Section 3.1.3

Table 3.3: Defining the action of VPartition. Statements that do not meet any condition pass through unaffected (remains in the transaction).

Statement	VPartition $(T_a, \vec{Cs}, T_n)$
Insert $I(T, \vec{C}, Q)$	$T = T_a \implies (\vec{C} \cap \vec{Cs} \neq \varnothing) \to I(T_a, (\mathbf{Keys}(T_a) \cup \vec{Cs}) \cap \vec{C}, \pi_{(\mathbf{Keys}(T_a) \cup \vec{Cs}) \cap \vec{C}} Q),$ $(\vec{C} \cap (\mathbf{Cols}(T_a) - (\mathbf{Keys}(T_a \cup \vec{Cs})))) \neq \varnothing) \to I(T_n, \vec{C} - \vec{Cs}, \pi_{\vec{C} - \vec{Cs}} Q)$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \implies U(T_a, \vec{F}, \vec{C} \cap \vec{Cs}, \pi_{\vec{C} \cap \vec{Cs}} Q),$ $I(T_n, \mathbf{Keys}(T_a), \pi_{\mathbf{Keys}(T_a)} \sigma_{\vec{F}} T_a - \pi_{\mathbf{Keys}(T_a)} T_n), U(T_n, \vec{F}, \vec{C} - \vec{Cs}, \pi_{\vec{C} - \vec{Cs}} Q)$
Delete $D(T, \vec{F})$	$T = T_a \implies D(T_n, \vec{F}), D(T_a, \vec{F})$
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_a \implies AT(T_a, \vec{C} \cap (\vec{Cs} \cup \vec{K}), \{d \in \vec{D}|col(d) \in \vec{Cs} \cup \vec{K}\}, \vec{K}),$ $AT(T_n, (\vec{C} - \vec{Cs}) \cup \vec{K}, \{d \in \vec{D}|col(d) \in \vec{C} - \vec{Cs}\}, \vec{K}), FK(true|T_n.\vec{K} \to true|T_a.\vec{K})$
Rename Table $RT(T_{old}, T_{new})$	Unaffected
Drop Table $DT(T)$	$T = T_a \implies DT(T_a), DT(T_n)$
Add Column $AC(T, C, D)$	$T = T_a \wedge C \notin \vec{Cs} \implies AC(T_n, C, D)$
Rename Column $RC(T, C_o, C_n)$	$T = T_a \wedge C_o \notin \vec{Cs} \implies RC(T_n, C_o, C_n)$
Drop Column $DC(T, C)$	$T = T_a \wedge C \notin \vec{Cs} \implies DC(T_n, C)$
Add Element $AE(T, C, E)$	$T = T_a \wedge C \in \mathbf{Keys}(T_a) \implies AE(T_a, C, E), AE(T_n, C, E)$ $T = T_a \wedge C \notin (\vec{Cs} \cup \mathbf{Keys}(T_a)) \implies AE(T_n, C, E)$
Rename Element $RE(T, C, E_o, E_n)$	$T = T_a \wedge C \in \mathbf{Keys}(T_a) \implies RE(T_a, C, E_o, E_n), RE(T_n, C, E_o, E_n)$ $T = T_a \wedge C \notin (\vec{Cs} \cup \mathbf{Keys}(T_a)) \implies RE(T_n, C, E_o, E_n)$
Drop Element $DE(T, C, E)$	$T = T_a \wedge C \in \mathbf{Keys}(T_a) \implies DE(T_a, C, E), DE(T_n, C, E)$ $T = T_a \wedge C \notin (\vec{Cs} \cup \mathbf{Keys}(T_a)) \implies DE(T_n, C, E)$
Foreign Key $FK(\vec{F}|T.\vec{X} \to \vec{G}|B.\vec{Y})$	$T = T_a \wedge \vec{X} \subseteq (\mathbf{Cols}(T_a) - (\vec{Cs} \cup \mathbf{Keys}(T_a))) \implies FK(\vec{F}|T_n.\vec{X} \to \vec{G}|B.\vec{Y})$ $T = T_a \wedge \vec{X} \cap \vec{Cs} \neq \varnothing \wedge \vec{X} \cap (\mathbf{Cols}(T_a) - (\vec{Cs} \cup \mathbf{Keys}(T_a))) \neq \varnothing \implies$ $Check(\pi_{\vec{X}} \sigma_{\vec{F}} (T_a \bowtie T_n), \pi_{\vec{Y}} \sigma_{\vec{G}} B)$

Table 3.4: Defining the action of VMerge. Statements that do not meet any condition pass through unaffected (remain in the transaction).

Statement	VMerge (T_a, T_n)
Insert $I(T, \vec{C}, Q)$	$T = T_n \implies Loop(t, Q, U(T_a, < \mathbf{Keys}(T_n), \pi_{\mathbf{Keys}(T_n)} t >,$ $\vec{C} - \mathbf{Keys}(T_n), \pi_{\vec{C} - \mathbf{Keys}(T_n)} t))$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \vee T = T_n \implies U(T_a, \vec{F}, \vec{C}, Q)$
Delete $D(T, \vec{F})$	$T = T_n \implies U(T_a, \vec{F}, \mathbf{Cols}(T_n) - \mathbf{Keys}(T_n), \forall_{c \in \mathbf{Cols}(T_n) - \mathbf{Keys}(T_n)} null)$
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_n \wedge \exists_{FK(true\|T_n.\vec{X} \to true\|T_a.\vec{X}) \in \mathcal{T}} \vec{X} = \mathbf{Keys}(T_n) \implies$ $\forall_{c \in \vec{C} - \vec{X}} AC(T_a, c, \mathbf{Dom}(c))$, Drop this FK
Rename Table $RT(T_{old}, T_{new})$	$T = T_n \implies$ Drop statement
Drop Table $DT(T)$	$T = T_n \implies \forall_{c \in (\mathbf{Cols}(T_n) - \mathbf{Keys}(T_n))} DC(T_a, c)$
Add Column $AC(T, C, D)$	$T = T_a \vee T = T_n \implies AC(T_a, C, D)$
Rename Column $RC(T, C_o, C_n)$	$T = T_a \vee T = T_n \implies RC(T_a, C_o, C_n)$
Drop Column $DC(T, C)$	$T = T_a \vee T = T_n \implies DC(T_a, C)$
Add Element $AE(T, C, E)$	$T = T_a \vee T = T_n \implies AE(T_a, C, E)$
Rename Element $RE(T, C, E_o, E_n)$	$T = T_a \vee T = T_n \implies RE(T_a, C, E_o, E_n)$
Drop Element $DE(T, C, E)$	$T = T_a \vee T = T_n \implies DE(T_a, C, E)$
Foreign Key $FK(\vec{F}\|T.\vec{X} \to \vec{G}\|B.\vec{Y})$	$T = T_n \wedge \vec{X} \cap \mathbf{Keys}(T_a) = \emptyset \implies FK(\vec{F}\|T_a.\vec{X} \to \vec{G}\|B.\vec{Y})$ $T = T_n \wedge \vec{X} \cap \mathbf{Keys}(T_a) \neq \emptyset \implies Check(\pi_{\vec{X}} \sigma_{\vec{F} \wedge N} T, \pi_{\vec{Y}} \sigma_{\vec{G}} B)$ $B = T_n \wedge \forall_{<c,v> \in \vec{G}} c \in \mathbf{Keys}(T_a) \implies FK(\vec{F}\|T.\vec{X} \to \vec{G}\|T_a.\vec{Y})$ $B = T_n \wedge \exists_{<c,v> \in \vec{G}} c \notin \mathbf{Keys}(T_a) \implies Check(\pi_{\vec{X}} \sigma_{\vec{F}} T, \pi_{\vec{Y}} \sigma_{\vec{G} \wedge N} B)$ (N is the condition that all non-key columns in T_n are not null)

Table 3.5: Defining the action of HPartition. Statements that do not meet any condition pass through unaffected.

Statement	HPartition (T_a, C_{in})
Insert $I(T, \vec{C}, Q)$	$T = T_a \implies \forall_{t \in \mathbf{Dom}(C_{in})} I(t, \vec{C} - \{C_{in}\}, \pi_{\vec{C} - \{C_{in}\}} \sigma_{C_{in} = t} Q)$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \wedge \exists_{<c,v> \in \vec{F}} c = C_{in} \implies U(v, \vec{F} - \{< c, v >\}, \vec{C}, Q)$ $T = T_a \wedge \nexists_{<c,v> \in \vec{F}} c = C_{in} \implies \forall_{t \in \mathbf{Dom}(C_{in})} U(t, \vec{F}, \vec{C}, Q)$
Delete $D(T, \vec{F})$	$T = T_a \wedge \exists_{<c,v> \in \vec{F}} c = C_{in} \implies D(v, \vec{F} - \{< c, v >\})$ $T = T_a \wedge \nexists_{<c,v> \in \vec{F}} c = C_{in} \implies \forall_{t \in \mathbf{Dom}(C_{in})} D(t, \vec{F})$
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_a \implies \forall_{c \in \mathbf{Dom}(C_{in})} AT(c, \vec{C} - \{C_{in}\}, \vec{D} - \{\mathbf{Dom}(C_{in})\}, \vec{K} - \{C_{in}\})$
Rename Table	Drop Statement
Drop Table $DT(T)$	$T = T_a \implies \forall_{t \in \mathbf{Dom}(C_{in})} DT(t)$
Add Column $AC(T, C, D)$	$T = T_a \implies \forall_{t \in \mathbf{Dom}(C_{in})} AC(t, C, D)$
Rename Column $RC(T, C_o, C_n)$	$T = T_a \implies \forall_{t \in \mathbf{Dom}(C_{in})} RC(t, C_o, C_n)$
Drop Column $DC(T, C)$	$T = T_a \implies \forall_{t \in \mathbf{Dom}(C_{in})} DC(t, C)$
Add Element $AE(T, C, E)$	$T = T_a \wedge C = C_{in} \implies AT(E, \mathbf{Cols}(T_a) - \{C_{in}\}, \mathbf{Domains}(T_a) - \{\mathbf{Dom}(C_{in})\},$ $\mathbf{Keys}(T_a) - \{C_{in}\}), AdjustForeignKeys(T, C, E)$ $T = T_a \wedge C \neq C_{in} \implies \forall_{t \in \mathbf{Dom}(C_{in})} AE(t, C, E)$
Rename Element $RE(T, C, E_o, E_n)$	$T = T_a \wedge C = C_{in} \implies RT(E_o, E_n)$ $T = T_a \wedge C \neq C_{in} \implies \forall_{t \in \mathbf{Dom}(C_{in})} RE(t, C, E_o, E_n)$
Drop Element $DE(T, C, E)$	$T = T_a \wedge C = C_{in} \implies DT(E)$ $T = T_a \wedge C \neq C_{in} \implies \forall_{t \in \mathbf{Dom}(C_{in})} DE(t, C, E)$
Foreign Key $FK(\vec{F}|T.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$	$T = T_a \wedge C_{in} \in \vec{X} \implies \forall_{t \in \mathbf{Dom}(C_{in})} Check(\pi_{\vec{X}} \sigma_{\vec{F}}(t \times \{name(t)\}), \pi_{\vec{Y}} \sigma_{\vec{G}} B)$ $T = T_a \wedge \exists_{<c,v> \in \vec{F}} c = C_{in} \wedge C_{in} \notin \vec{X} \implies FK(\vec{F} - \{< C_{in}, v >\}|v.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ $T = T_a \wedge \nexists_{<c,v> \in \vec{F}} c = C_{in} \wedge C_{in} \notin \vec{X} \implies \forall_{t \in \mathbf{Dom}(C_{in})} FK(\vec{F}|t.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ $B = T_a \wedge C_{in} \in \vec{Y} \implies \forall_{t \in \mathbf{Dom}(C_{in})} Check(\pi_{\vec{X}} \sigma_{\vec{F}} T, \pi_{\vec{Y}} \sigma_{\vec{G}}(t \times \{name(t)\}))$ $B = T_a \wedge \exists_{<c,v> \in \vec{G}} c = C_{in} \wedge C_{in} \notin \vec{Y} \implies FK(\vec{F}|T.\vec{X} \rightarrow \vec{G} - \{< C_{in}, v >\}|v.\vec{Y})$ $B = T_a \wedge \nexists_{<c,v> \in \vec{G}} c = C_{in} \wedge C_{in} \notin \vec{Y}) \implies \forall_{t \in \mathbf{Dom}(C_{in})} FK(\vec{F}|T.\vec{X} \rightarrow \vec{G}|t.\vec{Y})$

Table 3.6: Defining the action of HMerge. Statements that do not meet any condition pass through unaffected.

Statement	HMerge $(\vec{Ts}, T_a, C_{out})$
Insert $I(T, \vec{C}, Q)$	$T \in \vec{Ts} \implies I(T_a, \vec{C} \cup \{C_{out}\}, Q \times \{name(T)\})$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T \in \vec{Ts} \implies U(T_a, \vec{F} \cup \{< C_{out}, name(T) >\}, \vec{C}, Q)$
Delete $D(T, \vec{F})$	$T \in \vec{Ts} \implies D(T_a, \vec{F} \cup \{< C_{out}, name(T) >\})$
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T \in \vec{Ts} \wedge T_a$ has been created $\implies AE(T_a, C_{out}, T)$, $\forall_{c \in \mathbf{Cols}(T)}(\nexists_{t \in \vec{Ts}} c \in \mathbf{Cols}(t)) \rightarrow AC(T_a, c, \mathbf{Dom}(c))$ $T \in \vec{Ts} \wedge T_a$ not yet created $\implies AT(T_a, \vec{C} \cup \{C_{out}\}, \vec{D} \cup \{\{T\}\}, \vec{K} \cup \{C_{out}\})$
Rename Table $RT(T_o, T_n)$	$T \in \vec{Ts} \implies RE(T_a, C_{out}, T_o, T_n)$
Drop Table $DT(T)$	$T \in \vec{Ts} \implies DE(T_a, C_{out}, T), \forall_{c \in \mathbf{Cols}(T)}((\nexists_{t \in (\vec{Ts} - \{T\})} c \in \mathbf{Cols}(t)) \rightarrow DC(T_a, c))$
Add Column $AC(T, C, D)$	$T \in \vec{Ts} \wedge \forall_{t \in \vec{Ts} - \{T\}}(C \notin \mathbf{Cols}(t) \vee \mathbf{Dom}(t.C) = D) \implies AC(T_a, C, D)$ $T \in \vec{Ts} \wedge \exists_{t \in \vec{Ts} - \{T\}}(C \in \mathbf{Cols}(t) \wedge \mathbf{Dom}(t.C) \neq D) \implies$ Throw error $T \in \vec{Ts} \wedge \exists_{t \in \vec{Ts} - \{T\}}(C \in \mathbf{Cols}(t) \wedge \mathbf{Dom}(t.C) = D) \implies$ Drop statement
Rename Column $RC(T, C_o, C_n)$	$T \in \vec{Ts} \wedge \exists_{t \in \vec{Ts} - \{T\}} C_o \in \mathbf{Cols}(t) \wedge C_n \in \mathbf{Cols}(T_a) \implies$ $U(T_a, < C_{out}, name(T) >, \{C_n, C_o\}, \{C_o, null\})$, $T \in \vec{Ts} \wedge \exists_{t \in \vec{Ts} - \{T\}} C_o \in \mathbf{Cols}(t) \wedge C_n \notin \mathbf{Cols}(T_a) \implies$ $AC(T_a, C_n, \mathbf{Dom}(C_{old})), U(T_a, < C_{out}, name(T) >, \{C_n, C_o\}, \{C_o, null\})$, $T \in \vec{Ts} \wedge \forall_{t \in \vec{Ts} - \{T\}} C_o \notin \mathbf{Cols}(t) \wedge C_n \notin \mathbf{Cols}(T_a) \implies RC(T_a, C_o, C_n)$ $T \in \vec{Ts} \wedge \forall_{t \in \vec{Ts} - \{T\}} C_o \notin \mathbf{Cols}(t) \wedge C_n \in \mathbf{Cols}(T_a) \implies$ $U(T_a, < C_{out}, name(T) >, \{C_n\}, \{C_o\}), DC(T_a, C_o)$
Drop Column $DC(T, C)$	$T \in \vec{Ts} \wedge \forall_{t \in \vec{Ts} - \{T\}} C \notin \mathbf{Cols}(t) \implies DC(T_a, C)$ $T \in \vec{Ts} \wedge \exists_{t \in \vec{Ts} - \{T\}} C \in \mathbf{Cols}(t) \implies U(T_a, < C_{out}, name(T) >, C, null)$
Add Element $AE(T, C, E)$	$T \in \vec{Ts} \wedge \forall_{t \in \vec{Ts}}(C \in \mathbf{Cols}(t) \rightarrow \exists AE(t, C, E) \in \mathcal{T}) \implies AE(T_a, C, E)$ $T \in \vec{Ts} \wedge \exists_{t \in \vec{Ts}}(C \in \mathbf{Cols}(t) \rightarrow \nexists AE(t, C, E) \in \mathcal{T}) \implies$ Throw error
Rename Element $RE(T, C, E_o, E_n)$	$T \in \vec{Ts} \wedge \forall_{t \in \vec{Ts}}(C \in \mathbf{Cols}(t) \rightarrow \exists RE(t, C, E_o, E_n) \in \mathcal{T}) \implies RE(T_a, C, E_o, E_n)$ $T \in \vec{Ts} \wedge \exists_{t \in \vec{Ts}}(C \in \mathbf{Cols}(t) \rightarrow \nexists RE(t, C, E_o, E_n) \in \mathcal{T}) \implies$ Throw error
Drop Element $DE(T, C, E)$	$T \in \vec{Ts} \wedge \forall_{t \in \vec{Ts}}(C \in \mathbf{Cols}(t) \rightarrow \exists DE(t, C, E) \in \mathcal{T}) \implies DE(T_a, C, E)$ $T \in \vec{Ts} \wedge \exists_{t \in \vec{Ts}}(C \in \mathbf{Cols}(t) \rightarrow \nexists DE(t, C, E) \in \mathcal{T}) \implies$ Throw error
Foreign Key $FK(\vec{F}|T.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$	$T \in \vec{Ts} \wedge B \notin \vec{Ts} \implies FK(\vec{F} \cup \{< C_{out}, T >\}|T_a.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ $T \notin \vec{Ts} \wedge B \in \vec{Ts} \implies FK(\vec{F}|T.\vec{X} \rightarrow \vec{G} \cup \{< C_{out}, B >\}|T_a.\vec{Y})$ $T \in \vec{Ts} \wedge B \in \vec{Ts} \implies FK(\vec{F} \cup \{< C_{out}, T >\}|T_a.\vec{X} \rightarrow \vec{G} \cup \{< C_{out}, B >\}|T_a.\vec{Y})$

Table 3.7: Defining the action of Apply. If none of the conditions are met, the statement passes through the transformation unaffected. Some DDL statements are not listed because they are unaffected by this transformation.

Statement	Apply $(T_a, \vec{C_{in}}, \vec{C_{out}}, f)$
Insert $I(T, \vec{C}, Q)$	$T = T_a \implies I(T_a, \vec{C} - \vec{C_{in}} \cup \vec{C_{out}}, \alpha_{\vec{C_{in}}, \vec{C_{out}}, f} Q)$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \land \forall_{c \in \vec{C_{in}}} c \in \vec{C} \implies U(T_a, \vec{F}, (\vec{C} - \vec{C_{in}}) \cup \vec{C_{out}}, \alpha_{\vec{C_{in}}, \vec{C_{out}}, f} Q)$ $T = T_a \land \exists_{c \in \vec{C_{in}}} c \notin \vec{C} \land \exists_{c \in \vec{C_{in}}} c \in \vec{C} \implies$ $U(T_a, \vec{F}, (\vec{C} - \vec{C_{in}}) \cup \vec{C_{out}}, \alpha_{\vec{C_{in}}, \vec{C_{out}}, f}((\pi_{\vec{C} - \vec{C_{in}}} Q) \times (\pi_{C_{in}} \alpha_{\vec{C_{out}}, \vec{C_{in}}, f^{-1}} \sigma_{\vec{F}}(T_a))))$
Delete $D(T, \vec{F})$	Unaffected
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_a \land (\vec{C_{in}} - \vec{C} \neq \varnothing) \implies$ Throw error (need to have all columns used by the function) $T = T_a \land (\vec{C_{in}} - \vec{C} = \varnothing) \implies$ $AT(T, \vec{C} - \vec{C_{in}} \cup \vec{C_{out}}, \vec{D} - \textbf{Inputdomains}(f) \cup \textbf{Outputdomains}(f), \vec{K})$
Drop Column $DC(T, C)$	$T = T_a \land C \in \vec{C_{in}} \implies$ Throw error
Add Element $AE(T, C, E)$	$T = T_a \land C \in \vec{C_{in}} \implies$ Throw error (function not defined on new domain element)
Rename Element $RE(T, C, E_o, E_n)$	$T = T_a \land C \in \vec{C_{in}} \implies$ Throw error (function not defined on new domain element)
Drop Element $DE(T, C, E)$	$T = T_a \land C \in \vec{C_{in}} \implies$ Throw error (function not defined on new domain without element E)
Foreign Key $FK(\vec{F}|T.\vec{X}$ $\rightarrow \vec{G}|B.\vec{Y})$	$T = T_a \land \vec{X} \cap \vec{C_{in}} \neq \varnothing \implies Check(\pi_{\vec{X}} \sigma_{\vec{F}} \alpha_{\vec{C_{out}}, \vec{C_{in}}, f^{-1}} T_a, \pi_{\vec{Y}} \sigma_{\vec{G}} B)$

Table 3.8: Defining the action of Pivot. Statements that either are not listed or do not meet specified conditions pass through the transformation unaffected.

Statement	Pivot (T_a, A, V)
Insert $I(T, \vec{C}, Q)$	$T = T_a \implies Error((\pi_{\mathbf{Keys_{in}}(T_a)}Q \bowtie T_a) \cap \pi_{\mathbf{Keys_{in}}(T_a)}\boxtimes_{\mathbf{Dom}(A),A,V}(\pi_{\mathbf{Cols_{out}}(T_a)}(Q \bowtie T_a)))$, $\forall_{c\in\mathbf{Dom}(A)}Loop(t, \sigma_{A=c}Q \bowtie (\pi_{\mathbf{Keys}(T_a)-\{A\}}T_a), U(T_a,$ $< \mathbf{Keys}(T_a) - \{A\}, \pi_{\mathbf{Keys}(T_a)-\{A\}}t >, \{c\}, \pi_V t)),$ $I(T_a, \mathbf{Cols_{out}}(T_a), \boxtimes_{\mathbf{Dom}(A),A,V}(Q\overline{\bowtie}(\pi_{\mathbf{Keys}(T_a)-\{A\}}T_a)))$ (If $\vec{C}$ does not contain V, perform the above using $Q \times null$)
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \wedge \exists_{<c,v>\in\vec{F}}c = A \implies U(T_a, \vec{F} - \{< c, v >\}, \vec{C} - \{V\} \cup \{v\}, Q)$ $T = T_a \wedge \nexists_{<c,v>\in\vec{F}}c = A \implies \forall_{c\in\mathbf{Dom}(A)}U(T_a, \vec{F}, \vec{C} - \{V\} \cup \{c\}, Q)$
Delete $D(T, \vec{F})$	$T = T_a \wedge \exists_{<c,v>\in\vec{F}}c = A \implies U(T_a, \vec{F} - \{< c, v >\}, \{c\}, \{null\})$
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_a \implies AT(T_a,$ $(\vec{C} - \{A, V\}) \cup \mathbf{Dom}(A), \vec{D} - \{\mathbf{Dom}(A), \mathbf{Dom}(V)\} \cup \{\forall_{a\in\mathbf{Dom}(A)}\mathbf{Dom}(V)\}, \vec{K} - \{A\})$
Add Column $AC(T, C, D)$	Throw error (all columns are either key columns or the value column, so this should never happen)
Drop Column $DC(T, C)$	Throw error (all columns are either key columns or the value column, so this should never happen)
Add Element $AE(T, C, E)$	$T = T_a \wedge C = A \implies AC(T_a, E, \mathbf{Dom}(V)), AdjustForeignKeys(T,C,E)$ $T = T_a \wedge C = V \implies \forall_{c\in\mathbf{Dom}(A)}AE(T, c, E)$
Rename Element $RE(T, C, E_o, E_n)$	$T = T_a \wedge C = A \implies RC(T_a, E_o, E_n)$ $T = T_a \wedge C = V \implies \forall_{c\in\mathbf{Dom}(A)}RE(T, c, E_o, E_n)$
Drop Element $DE(T, C, E)$	$T = T_a \wedge C = A \implies DC(T_a, E)$ $T = T_a \wedge C = V \implies \forall_{c\in\mathbf{Dom}(A)}DE(T, c, E)$
Foreign Key $FK(\vec{F}\|T.\vec{X}$ $\to \vec{G}\|B.\vec{Y})$	$T = T_a \wedge (A \in \vec{X} \vee (\exists_{<c,v>\in\vec{F}}c = A \wedge V \notin \vec{X} \wedge A \notin \vec{X})) \implies$ $Check(\pi_{\vec{X}}\sigma_{\vec{F}}\boxtimes_{\mathbf{Cols}(T_a)-\mathbf{Keys}(T_a),A,V}T_a, \pi_{\vec{Y}}\sigma_{\vec{G}}B)$ $B = T_a \wedge A \in \vec{Y} \implies Check(\pi_{\vec{X}}\sigma_{\vec{F}}T, \pi_{\vec{Y}}\sigma_{\vec{G}}\boxtimes_{\mathbf{Cols}(T_a)-\mathbf{Keys}(T_a),A,V}T_a)$ $T = T_a \wedge \exists_{<c,v>\in\vec{F}}c = A \wedge V \in \vec{X} \wedge A \notin \vec{X} \implies$ $FK(\vec{F} - \{< c, v >\}\|T.(\vec{X} - \{V\} \cup \{v\} \to \vec{G}\|B.\vec{Y}))$ $T = T_a \wedge \nexists_{<c,v>\in\vec{F}}c = A \wedge V \in \vec{X} \wedge A \notin \vec{X} \implies$ $\forall_{v\in\mathbf{Dom}(A)}(FK(\vec{F}\|T.(\vec{X} - \{V\} \cup \{v\} \to \vec{G}\|B.\vec{Y})))$

Table 3.9: Defining the action of Unpivot. If none of the conditions are met, the statement passes through the transformation unaffected. Some DDL statements are not listed because they are unaffected by this transformation.

Statement	Unpivot (T_a, A, V)
Insert $I(T, \vec{C}, Q)$	$T = T_a \implies \forall_{c \in \vec{C} - \mathbf{Keys}(T_a)} I(T_a, \mathbf{Keys}(T_a) \cup \{V, A\},$ $(\pi_{\mathbf{Keys}(T_a) \cup \{c\}} \sigma_{c \neq null} Q) \times \{name(c)\})$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \implies \forall_{c \in \vec{C}} I(T_a, \mathbf{Keys}(T_a) \cup \{A\}, (\pi_{\mathbf{Keys}(T_a)} \sigma_{\vec{F}} T_a - \pi_{\mathbf{Keys}(T_a)} \sigma_{\vec{F} \wedge A = c} T_a) \times \{c\}),$ $\forall_{c \in \vec{C}} ((\pi_{\{c\}} Q \neq null \rightarrow U(T_a, \vec{F} \cup \{< A, c >\}, \{V\}, (\pi_{\{c\}} Q)))$ $\wedge ((\pi_{\{c\}} Q = null \rightarrow D(T_a, \vec{F} \cup \{< A, c >\}))$
Delete $D(T, \vec{F})$	Unaffected
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_a \implies AT(T_a, \vec{K} \cup \{A, V\}, \{d \in \vec{D} \mid col(d) \in \vec{K}\} \cup \{\vec{C} - \vec{K}\}$ $\cup \{d \mid col(d) \in \vec{C} - \vec{K}\}, \vec{K} \cup \{A\})$
Add Column $AC(T, C, D)$	$T = T_a \wedge C \notin \mathbf{Keys}(T_a) \implies AE(T_a, A, C)$
Rename Column $RC(T, C_o, C_n)$	$T = T_a \wedge C_o \notin \mathbf{Keys}(T_a) \implies RE(T_a, A, C_o, C_n)$
Drop Column $DC(T, C)$	$T = T_a \wedge C \notin \mathbf{Keys}(T_a) \implies DE(T_a, A, C)$
Add Element $AE(T, C, E)$	$T = T_a \wedge C \notin \mathbf{Keys}(T_a) \wedge \forall_{c \in (\mathbf{Cols}(T_a) - \mathbf{Keys}(T_a))} \exists AE(T_a, c, E) \in \mathcal{T} \implies AE(T_a, V, E)$ $T = T_a \wedge C \notin \mathbf{Keys}(T_a) \wedge \exists_{c \in (\mathbf{Cols}(T_a) - \mathbf{Keys}(T_a))} \nexists AE(T_a, c, E) \in \mathcal{T} \implies$ Throw error
Rename Element $RE(T, C, E_o, E_n)$	$T = T_a \wedge C \notin \mathbf{Keys}(T_a) \wedge \forall_{c \in (\mathbf{Cols}(T_a) - \mathbf{Keys}(T_a))}$ $\exists RE(T_a, c, E_o, E_n) \in \mathcal{T} \implies RE(T_a, V, E_o, E_n)$ $T = T_a \wedge C \notin \mathbf{Keys}(T_a) \wedge \exists_{c \in (\mathbf{Cols}(T_a) - \mathbf{Keys}(T_a))}$ $\nexists RE(T_a, c, E_o, E_n) \in \mathcal{T} \implies$ Throw error
Drop Element $DE(T, C, E)$	$T = T_a \wedge C \notin \mathbf{Keys}(T_a) \wedge \forall_{c \in (\mathbf{Cols}(T_a) - \mathbf{Keys}(T_a))}$ $\exists DE(T_a, c, E) \in \mathcal{T} \implies DE(T_a, V, E)$ $T = T_a \wedge C \notin \mathbf{Keys}(T_a) \wedge \exists_{c \in (\mathbf{Cols}(T_a) - \mathbf{Keys}(T_a))}$ $\nexists DE(T_a, c, E) \in \mathcal{T} \implies$ Throw error
Foreign Key $FK(\vec{F} \mid T.\vec{X}$ $\rightarrow \vec{G} \mid B.\vec{Y})$	$T = T_a \wedge \vec{X} - \mathbf{Keys}(T_a) \neq \varnothing \implies Check(\pi_{\vec{X}} \sigma_{\vec{F}} \boxtimes_{\mathbf{Cols}(T_a) - \mathbf{Keys}(T_a), A, V} T_a, \pi_{\vec{Y}} \sigma_{\vec{G}} B)$

- An error statement $Error(Q)$ that returns an error (aborts transaction), introduced in Section 3.1.4

- A loop construct $Loop(t, Q, T)$ for some variable t, query Q, and nested transaction T, introduced in Section 3.1.4

Channels process transactions that can contain one or more update statements. When a transaction is pushed through a channel transformation, the transformation works on each statement in the transaction in place; if a statement in a transaction is processed non-trivially by a transformation, the output of that processing replaces the original statement in the position of the transaction held by that statement. While processing a transaction, the situation may arise in which one or more or even all of the statements in the transaction are dropped, in which case no further processing is necessary. It may also be the case that a transformation throws an error, in which case the entire transaction needs to be aborted. An error also means that any changes made to the internal state of the transformations by the transaction must be rolled back.

There are some instances where a transformation will throw an error when it encounters a statement, which aborts the transaction immediately. This situation only occurs with DDL statements, namely when:

- A schema change has caused columns to have different domains, where domain equality is required for a transformation

- A change to the elements of a domain has caused a function's input to change

- A schema change has invalidated one of the pre-conditions of a transformation, e.g. adding a second non-key column to a table that will be pivoted

On system startup, the DDL statements required to represent the natural schema are generated automatically and enter the channel just as any other update. This process initializes the internal state of each transformation, so that it is aware of the schema and referential integrity constraints at its position in the channel. The process also generates the DDL for the schema of the physical database as a result, so if the physical database instance does not yet exist, system initialization creates it.

As additional DDL statements pass through the channel, the various transformations update their parameters dynamically. For instance, if an Unpivot transformation acts on the table A, and a Rename Column statement passes through the channel to rename A to B, the operator will change its operation to work on B. If the rename occurred during a transaction that is later aborted, that change is rolled back so that, during subsequent transactions, the Unpivot transformation will revert to operating on table A.

One final note about transactions: Some DDL statements can only pass through a transformation if other statements are present in the current transaction. For instance, consider the case introduced above where an Add Element statement is pushed through an Unpivot transformation. If a single Add Element statement passes through an Unpivot, it may cause a union-compatibility error because the non-key domains no longer have the same domain. However, if Add Element statements for all of the non-key columns pass through in the same transaction, they collapse into a single Add Element statement on the new value column, thereby providing transactional consistency: union-compatibility of the non-key columns both before and after the transaction executes.

3.2.3 Instance Transformations and Copy Inserts

We can copy one table, T, into another table with a new name but identical schema, say T-copy, using the DML statement `INSERT INTO T-copy (SELECT * FROM T)`. By pushing a statement of this form for each table in a schema through a channel, we can transform entire instances of a table in both directions: (1) to materialize a table in the conceptual schema, or (2) to push an instance of a table in the conceptual schema (assuming that an instance exists), in its entirety, into the physical database. Figure 3.7 gives more details on how a channel can be used to materialize an instance.

First, we recognize that the statement `S = INSERT INTO T2 SELECT * FROM T1` is exactly the statement that copies the table `T1` into table `T2` (we refer to this kind of statement as a *copy insert*). Therefore, if `T1 = db1.T` and `T2 = db2.T`, the statement is effectively the identity transformation of table T, just moving from database 1 to database 2. Before channel transformation, both `db1` and `db2` have the same schema.

Now, consider a given channel `C`, but constructed two different ways. First, construct the channel `C1`, which is the channel `C` with all of its transformations operating on tables in database `db1`. We can push `S` through `C1`, which would only transform the query part of the statement, producing `S1 = INSERT INTO db2.T C1(SELECT * FROM db1.T)`. The insert part of the statement remains unchanged because `db2.T` is in database `db2` and thus invariant under channel `C1`. The effect of statement `S1` is to populate table `db2.T` with the results of `SELECT * FROM db1.T`: in effect, materializing table *T* in the natural schema. Note that if we consider the natural schema to be a view over the physical schema, this situation is equivalent to materializing a view.

The other option is to construct the channel `C2`, which is the channel `C` with all of
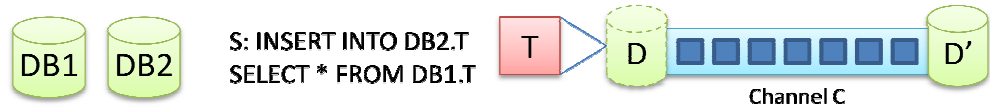
Starting point: two databases, a schema D which contains table T, a statement S that copies data from DB1.T to DB2.T, and a channel from schema D to schema D'
In both cases, DB1 stores the data. The question is: which database has schema D?

Option 1: Translate data instance from D' to D
In DB1, D serves as the natural schema for the channel C, DB1 stores data in schema D'
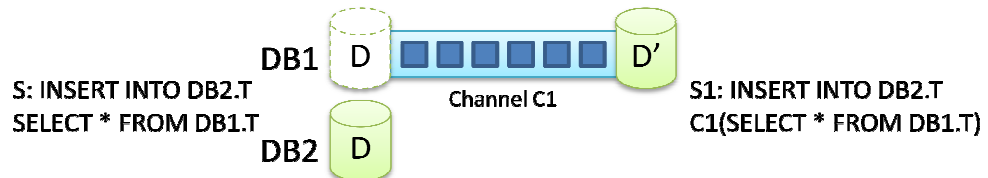
C1 is exactly channel C, acting on the tables in DB1
Push statement S through the channel into statement S1

Option 2: Translate data instance from D to D'
In DB2, D serves as the natural schema for the channel C, DB1 stores data in schema D
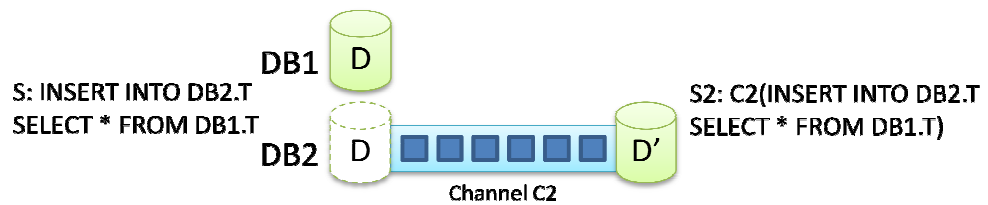
C2 is exactly channel C, acting on the tables in DB2
Push statement S through the channel into statement S2

Figure 3.7: Using a channel and an insert statement to move entire instance of a table through a channel

its transformations operating on tables in database db2. If we push statement S through C2, we get statement S2 = C2(INSERT INTO db2.T SELECT * FROM db1.T). The query portion of this statement will pass through the channel essentially unaffected, since the transformations in C2 only affect db2; however, the processing of the insert statement may affect the query by treating it as a constant result during statement transformation. For instance, pushing S through $VPartition(db2.T, \vec{C}s, db2.T')$ will yield two statements, INSERT INTO db2.T SELECT $\vec{A}$ FROM db1.T and INSERT INTO db2.T' SELECT $\vec{B}$ FROM db1.T WHERE C (where $\vec{A}$ and $\vec{B}$ are the columns in the output schema of db2.T and db2.T' and C is the not-all-nulls condition. The queries in the two output statements are exactly the single-table query SELECT * FROM db1.T with some additional operators placed above it in the tree. The effect of statement S2 is to take the table db2.T and effectively push it in its entirety through the channel — in the opposite direction as statement S1 was pushed through a channel.

This second application of pushing a copy insert through a channel does not have a clear analog in the view literature. It would be equivalent to taking a materialized view and pushing it back through the view definition to create its image in the physical storage, which can only be done for views that meet the restrictive definition of being updatable (unless a developer decides to manually develop rules or triggers to define the update rules). This application can, however, support the typical physical design workflow; a database developer starts with a database schema, constructs a channel using the schema as the channel's input, then "pushes" the instance through the newly-designed channel using copy inserts, leaving the original schema as the input schema to the channel and the virtual view of the data.

3.3 PHYSICAL DATABASE DESIGN AND OPTIMIZATION

The transformations supported in a channel (listed in Table 3.2) are frequently used by database developers for physical database design. In other words, by applying one of these seven transformations, a database designer may be able to save space on disk, or improve the speed of query execution. But there are a number of other decisions that a physical database designer makes as well. Tools called database tuning advisors exist [10, 18] that automatically or semi-automatically recommend physical structures (e.g., indexes and materialized views) to create for a database for a given workload of queries, DML updates, and (in some cases) DDL updates.

Two long-term goals with our channel research are fully automated channel design and channel optimization. The first of these goals — automated channel design — is already partially supported by some physical database tuning advisors; for instance, one database tuning advisor [1] can recommend either horizontal or vertical partitioning. In general, for any of our seven transformations, it is possible to develop heuristics to decide whether applying a transformation will provide some benefit.

Consider the case of the Unpivot transformation. One can roughly calculate how much space one would save by moving to a generic layout using an Unpivot transformation. More specifically, one can calculate how sparse the database must be for there to be a space savings. Making two fundamental assumptions — that rows are stored in a fixed-width representation and that storage overhead for each row and column is negligible — if I is the number of bytes in the primary key, A is the maximum number of bytes in an attribute name, S is the size of a data column, N is the number of data (pivot) columns, R is the number of rows, and P is the percentage of non-null data values, then

for an unpivot transformation to save space, the following inequality must hold:

$$(I + A + S)NPR \leq (I + NS)R$$

Solving for P, we have:

$$P \leq \frac{I/N + S}{I + A + S}$$

For some sample values, say $A = 4$, $I = 4$, $S = 4$, and $N = 50$, we see that for a 50-column natural schema with all columns being 32-bits wide, $P \leq 34\%$. Of course, in physical database design, space is not the only concern; the Unpivot transformation introduces a Pivot query operator into queries. So, even if a table with the above characteristics has a non-null data percentage of 5%, the added query overhead may make the space savings too expensive.

Incorporating design decisions (such as the analysis above of the Unpivot transformation) into tuning advisors is still largely future work. We describe here our work that allows the use of existing database tuning advisors to suggest physical storage structures when designing a channel, thereby combining the physical design capabilities of a channel with those of a traditional DBMS tuning advisor in a natural way.

3.3.1 Physical Characteristics

A typical physical tuning advisor requires two inputs: statistics that describe the data contained in the database and a workload of statements that is representative of the application workload that will run on the database. As described in Section 3.2.2, a channel can accept a transaction of statements against the natural schema (i.e., an application workload) and translate them into a transaction of statements against the physical

database (i.e., a physical workload). To support the use of tuning advisors, we define here how statistics can be pushed through channel transformations. Specifically, a transformation can take a row count of a table and histograms of value distributions of columns in the natural schema and transform them into statistics of the same form against the physical schema. Therefore, one can specify or gather statistics and a workload against the natural schema and push them both through the channel to be fed to database tuning advisors. It is generally the case that tuning advisors will accept a workload specified by the user, but will rely on database statistics generated within the DBMS rather than accept statistics from outside. If the tuning advisor cannot be adapted to work with outside statistics, as a workaround we can generate a sample database that meets the statistical characteristics, and then let the DBMS reconstruct the statistics internally.

Formally, we describe table statistics as a new statement $TStat(T, R, \vec{H})$, where T is the name of a table, R is the row count for that table, and $\vec{H}$ is a collection of histograms. It is this statement that we can push through channel transformations. The ith entry in the histogram collection $h_i \in \vec{H}$ describes the distribution of values in the ith column C_i of the table T, including null values, and is described as a function from $(domain(C_i) \cup \{null\}) \rightarrow \mathbb{Q}$ (where $\mathbb{Q}$ is the set of rational numbers). So, $h_{id}(a) = 5$ means that 5 rows in the table have a value of 'a' in the 'id' column. The output of the function may be a rational (non-integral) number to reflect an estimated value. We use the notation h_C to represent the histogram for column C and thus avoid the need to refer to position.

Since each histogram is associated with a particular column, the transformations must know the input schema to connect each histogram to the proper column, as well

as the domain of each column to know the input domains of each histogram. In other words, before a channel can process a statement in the form $TStat(T_a, R, \vec{H})$ for some table T_a, the channel must have at some point in its operation seen a statement of the form $AT(T_a, \vec{C}, \vec{D}, \vec{K})$ or other DDL statements describing the state of the input schema. The channel then knows that the set of histograms $\vec{H}$ correspond to the columns $\vec{C}$.

A channel transformation operates on statistical information in the same manner as queries or updates. Given a statement of the form $TStat(T, R, \vec{H})$ over a transformation's input schema, the transformation generates *TStat* statements representing the estimated physical characteristics of data over the transformation's output schema. Table 3.10 describes the action of each of the seven transformations on table statistics, along with a description of any estimates that are made along the way.

For our purposes, we assume that each histogram is complete, providing statistics on each member of the active domain for a column. In reality, histograms will store information on ranges of values, either height-balanced (equal number of data entries per bucket) or width-balanced (equal number of domain values per bucket). Table 3.10 can be easily extrapolated to work on summarized data, assuming uniform distribution within each bucket if necessary.

One immediate benefit of pushing statistics through a channel is the ability to estimate the size of the physical database based on the anticipated size of the input data. Given a statistics statement $TStat(T_a, R, \vec{H})$ and the list of columns $\vec{C}$ with associated domains $\vec{D}$, calculating the size of table T_a is trivial.

Table 3.10: Defining the action of each transformation on table statistics.

Transformation	Description of Statistics Transformation
VPartition $(T_a, \vec{Cs}, T_n)$	$TStat(T_a, R, \vec{H}) \implies TStat(T_a, R_a, \vec{H^a}), TStat(T_n, R_n, \vec{H^n})$, where: $R_a = R$, $R_n = R_a - K$, $H_C^a = H_C$ for all $C \in \vec{Cs} \cup \mathbf{Keys}(T_a)$, $H_C^n = (f_C \circ H_C)$ for all $C \in \mathbf{Cols_{in}}(T_a) - (\vec{Cs} \cup \mathbf{Keys}(T_a))$ $f_C(x) = x - K$ (C non-key, null), x (C non-key, non-null), $x \times ((R-K)/R)$ (else) $K = \lfloor R_a * \Pi_{c \in \mathbf{Cols_{in}}(T_a) - (\vec{Cs} \cup \mathbf{Keys}(T_a))}(H_c(null)/R_a) \rfloor$ (K is an estimate of all-null rows in T_n, assuming random distribution of nulls)
VMerge (T_a, T_n)	$TStat(T_a, R_a, \vec{H^a}), TStat(T_n, R_n, \vec{H^n}) \implies TStat(T_a, R_a, \vec{H})$, where: $H_C = H_C^a$ for all $C \in \mathbf{Cols_{in}}(T_a)$, and $(f_C \circ H_C^n)$ for all $C \in \mathbf{Cols_{in}}(T_n) - \mathbf{Keys}(T_n)$ $f_C(x) = x + R_a - R_n$ when value of C is null, x otherwise
HPartition (T_a, C_{in})	$TStat(T_a, R, \vec{H}) \implies \forall_{t \in \mathbf{Dom}(C_{in})} TStat(t, R_t, \vec{H^t})$, where: $R_t = H_{C_{in}}(t)$, H_c^t for any column c and table t is defined by $H_c^t(x) = H_c(x)/H_{C_{in}}(t)$ (Assume values in columns $\mathbf{Cols_{in}}(T_a) - \{C_{in}\}$ evenly distributed among new tables)
HMerge $(\vec{Ts}, T_a, C_{out})$	$(\forall_{t \in \vec{Ts}} TStat(t, R_t, \vec{H^t})) \implies TStat(T_a, R_a, \vec{H^a})$, where: $R_a = \Sigma_{t \in \vec{Ts}} R_t$, $H_{C_{out}}^a$ is defined by $H_{C_{out}}^a(x) = \begin{cases} 0 & \text{if } x = null \\ R_x & \text{else} \end{cases}$ $H_c^a, c \neq C_{out}$, is defined by $H_c^a(x) = \begin{cases} \Sigma_{\{t \mid c \text{ is not a column in } t\}}(R_t) & \text{if } x = null \\ \Sigma_{\{t \mid c \text{ is a column in } t\}}(H_c^t(x)) & \text{else} \end{cases}$
Apply $(T_a, \vec{C_{in}}, \vec{C_{out}}, f)$	$TStat(T_a, R, \vec{H}) \implies TStat(T_a, R, \vec{H^o})$ where: $H_c^o = H_c$ for any column $c \in \mathbf{Cols}(T_a) - \vec{C_{in}}$ $H_c^o(x) = R \times \Sigma_{v \in V_x} \Pi_{b \in \vec{C_{in}}}(H_b(v_b)/R)$ where c is the ith member of $\vec{C_{out}}$ and $V_x = \{v = (v_1, v_2, \ldots, v_{|\vec{C_{in}}|}) \mid f(v)_i = x\}$ (Assume that there are no correlations between non-key columns)
Pivot (T_a, A, V)	$TStat(T_a, R, \vec{H}) \implies TStat(T_a, R_p, \vec{H^p})$, where: $R_p = \min(R, \Pi_{c \in \mathbf{Keys_{in}}(T_a) - \{A\}}(|activedom(c)|))$, where $activedom(c)$ is the number of distinct values x with non-zero values for $H_c(x)$, $H_c^p = H_c$ for $c \in \mathbf{Keys_{in}}(T_a) - \{A\}$ $H_c^p, c \in \mathbf{Dom}(A)$, is defined by $H_c^p(x) = \begin{cases} R_p - H_A(c) & \text{if } x = null \\ H_V(x) \times H_A(c)/R & \text{else} \end{cases}$ (Assume that the values in column V are evenly distributed among pivot columns)
Unpivot (T_a, A, V)	$TStat(T_a, R, \vec{H}) \implies TStat(T_a, R_u, \vec{H^u})$, where: $R_u = \Sigma_{c \in \mathbf{Cols_{in}}(T_a) - \mathbf{Keys_{in}}(T_a)}(R - H_c(null))$, $H_c^u = H_c$ for any column $c \in \mathbf{Keys_{in}}(T_a)$ $H_A^u(x) = R - H_x(null)$, $H_V^u(x) = \Sigma_{c \in \mathbf{Cols_{in}}(T_a) - \mathbf{Keys_{in}}(T_a)} H_c(x)$

3.3.2 Transformation Equivalences

Our second long-term goal — optimization — refers to reducing the amount of over-head that a channel introduces. One simple operational optimization we can make is to avoid using the entire channel for each statement. If the statements in a transaction S only reference two tables, and 95% of the transformations in the channel do not have an effect on those tables, it would be useful if S were only processed by the remaining 5% of the transformations. The *trace* of a channel C on a table T (denoted $trace_C(T)$) is the list of transformations within C that will potentially operate on table T or its interme-diate products. The channel represents a trace as a bitmap, where each bit represents a transformation in the channel. One can generate a trace for T by pushing the Add Table statement for T through the channel, and have each transformation report if it operated on the transaction. All Add Table statements for the natural schema are pushed through the channel at system initialization, and any new tables must be also pushed through the channel, so creating traces requires no extra work. To generate the *footprint* of a transaction, simply take the logical 'or' of the traces for all of the tables referenced by the transaction.

To further optimize a channel, we consider equivalences:

Definition 3.3.1 *Two channels A and B are* equivalent *($\equiv$) if, for any transaction $\mathcal{T}$, $A(\mathcal{T}) = B(\mathcal{T}$), and for any query Q in extended relational algebra, A(Q) is equivalent to B(Q).*

In the definition above, the notation $C(S)$ for transformation C and statement or transaction S refers to the output of running S through the transformations defined for

C. Given a channel *C*, channel optimization is the process of finding another channel *C'* such that *C* ≡ *C'* and that *C'* is more desirable with respect to a metric, for instance, trace size. We provide a sample of transformation equivalences that one can use to rewrite channels into equivalent forms. These equivalences fall into two categories: commutativity and inverses. We use these equivalences to try to create a more efficient channel in two ways. First, we can use equivalences to reduce the size of traces for tables, meaning that fewer processing cycles are spent in the channel. Second, we can use inverse relationships to identify transformations that can be eliminated entirely.

One can use commutativity to move the transformations around within a channel. This theorem says that two channel transformations can commute if they do not act on any common tables in the input schema:

Theorem 3.3.1 *If channel C = [A, B],*

$$\nexists_T (A \in trace_C(T) \land B \in trace_C(T)) \rightarrow [A, B] \equiv [B, A]$$

This theorem is trivial to prove, and follows from the fact that no table will be affected by both of the transformations. There are also examples of commutativity that are not trivial. We provide two examples here:

Theorem 3.3.2 $[VMerge(T_a, T_n), Apply(T_a, \vec{C}_{in}, \vec{C}_{out}, f)] \equiv$

$[Apply(T_a, \vec{C}_{in}, \vec{C}_{out}, f), VMerge(T_a, T_n)]$ *if* $\vec{C}_{in} \subseteq \mathbf{Cols_{in}}(T_a)$

This theorem allows an Apply to be pushed through a VMerge when the input columns for the function are not spread over the two tables being merged. The trace of T_n for the left side of the equivalence has two transformations, while its trace on the

right side only has one. A query involving T_n and not T_a will spend fewer processing cycles in the channel and thus run faster.

There is a companion theorem if the Apply operates on data in the other table:

Theorem 3.3.3 $[VMerge(T_a, T_n), Apply(T_a, \vec{C_{in}}, \vec{C_{out}}, f)] \equiv$

$[Apply(T_n, \vec{C_{in}}, \vec{C_{out}}, f), VMerge(T_a, T_n)]$ *if* $\vec{C_{in}} \cap \mathbf{Cols}(T_a) = \varnothing$

Here is another commutativity equivalence:

Theorem 3.3.4 $[HMerge(\vec{T}s, T, C), Unpivot(T, A, V)] \equiv$

$[\forall_{t \in \vec{T}s} Unpivot(t, A, V), HMerge(\vec{T}s, T, C)]$

This equivalence pushes an HMerge through an Unpivot — the result is that each of the HMerged tables is run through its own Unpivot first. This equivalence has no effect on table traces; any table T in the natural schema will either be affected by both an Unpivot and an HMerge or by neither, regardless of whether the equivalence is applied. However, pushing the HMerge through the Unpivot may allow further equivalences to be applied later.

Finally, we identify inverses, where a combination of transformations produces the same results as the channel with no operators, ϵ. Each of the seven transformations has an inverse transformation. The inverse relationships can be divided into two categories. First, there are the inverses that can be determined statically, without knowing the natural schema:

Theorem 3.3.5 $[Pivot(T_a, A, V), Unpivot(T_a, A, V)] \equiv \epsilon$

$[Unpivot(T_a, A, V), Pivot(T_a, A, V)] \equiv \epsilon$

$$[Apply(T_a, \vec{C_{in}}, \vec{C_{out}}, f), Apply(T_a, \vec{C_{out}}, \vec{C_{in}}, f^{-1})] \equiv \epsilon$$

$$[VPartition(T_a, \vec{C}s, T_n), VMerge(T_a, T_n)] \equiv \epsilon$$

We provide a proof of the last equivalence shown above in Chapter 5. These inverses are always true, and should always be applied to reduce work. Second, there are inverses that can only be determined knowing at least part of the natural schema (if one is using a graphical channel designer, e.g., Figure 3.2(b), this information would be available, as it could read in the current natural schema at runtime):

Theorem 3.3.6 $[HPartition(T_a, C_{in}), HMerge(\vec{D}, T_a, C_{in})] \equiv \epsilon$, *where $\vec{D}$ is the set of all elements in the domain of C_{in} in the channel's input schema.*

In the theorem above, we need to know the domain of column C_{in} to check if $\vec{D}$ covers the correct set of values.

Theorem 3.3.7 $[HMerge(\vec{T}s, T_a, C_{out}), HPartition(T_a, C_{out})] \equiv \epsilon$, *only if all of the tables in $\vec{T}s$ are union-compatible.*

Theorem 3.3.8 $[VMerge(T_a, T_n), VPartition(T_a, \vec{C}s, T_n)] \equiv \epsilon$, *where $\vec{C}s$ is the list of non-key columns in table T_a in the natural schema.*

A channel optimizer, upon encountering the opportunity to apply one of these equivalences, should present it to the developer before applying it, or should only apply the equivalence at runtime; it may be the case that the developer really meant for both transformations to be in the channel, or that the equivalence will not hold after the schema evolves.

3.4 IMPLEMENTATION DETAILS AND INSIGHTS

We have implemented a prototype of the channel that includes the seven transformations introduced in this chapter in Visual Studio 2008 using the C# language. Figure 3.8 shows some basic statistics about our implementation. The chart on the top shows a line count of each of our seven transformations' implementation. None of the transformations took more than 800 lines of code to implement, including the relatively complex Pivot transformation. In Chapter 7, we discuss the possibility of allowing developers to create their own transformations; 800 lines of code per transformation implies that new transformations may be relatively simple to write in our framework. The bottom chart shows the distribution of lines of code in the entire Guava project, sub-divided by function. The total lines of code for the project was less than 25,000, and could be substantially smaller if we had access to the source code of the C# GUI widget library (so that we could avoid redundant code in some places). The compiled size of the entire framework is about 400k, which is small compared to the approximately 2MB required by the System.Windows.Forms.dll component that houses the GUI widget library.

Our implementation of the channel can be broken into two components: the visitor pattern and the provider model. We discuss each of these components individually. Our work with the visitor pattern and the provider model are inspired by the Microsoft Entity Framework [8], which uses a similar internal architecture.

3.4.1 Command Trees and the Visitor Pattern

Each supported statement that can be processed as input to a channel in the Guava model is represented in a tree structure. For example, for each supported operator in
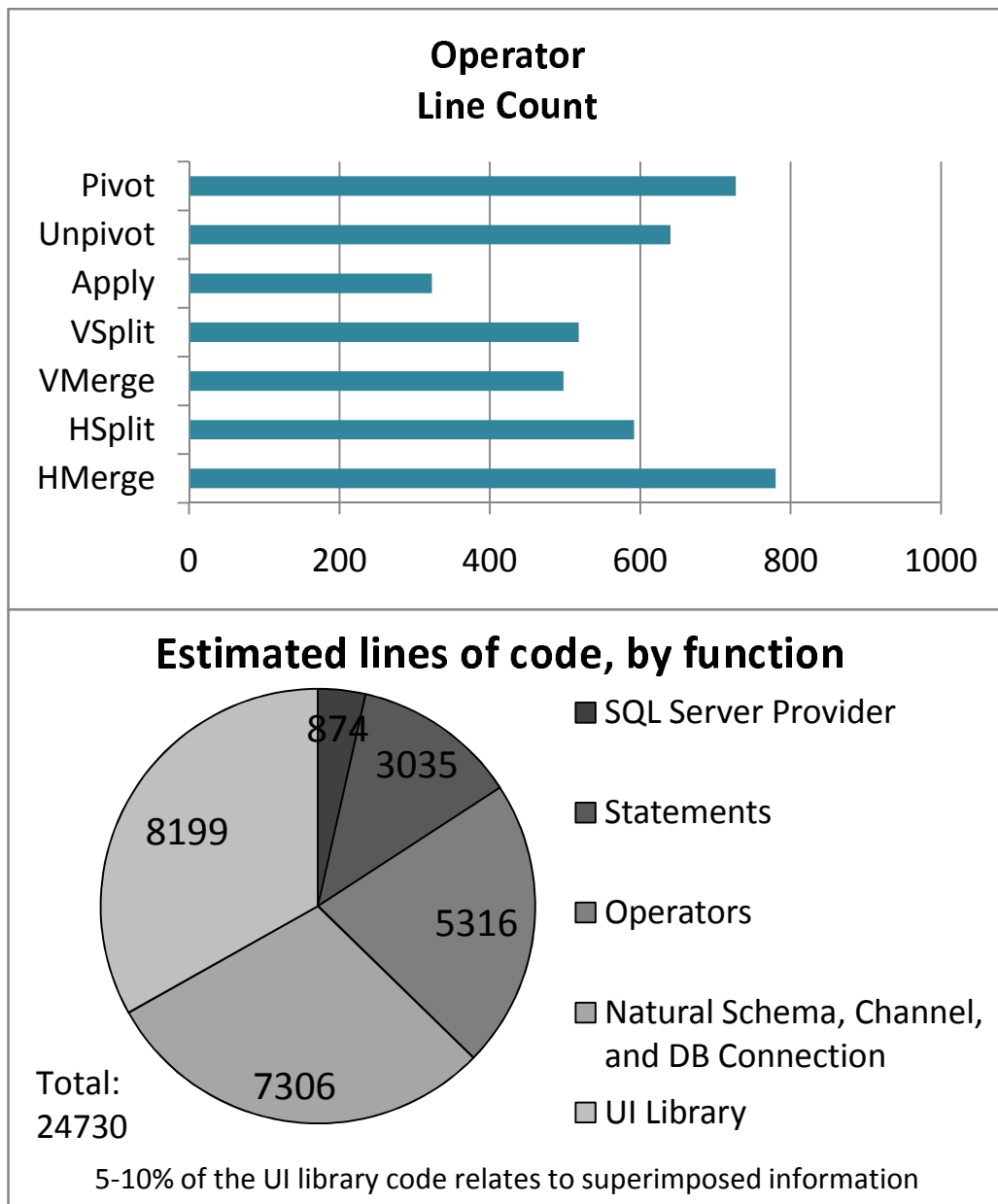
Figure 3.8: Statistics of the code in our prototype channel implementation. Statistics were gathered as of June 20th, 2008

our extended relational model (i.e., each operator that can appear in a query), there is an object class that defines the structure of that operator. As an example, there is an object class that represents a join operator; the `Join` class has two children, representing the two expressions that the query operator is joining. The class also has properties representing the join condition between the two expressions. A query is represented internally as a tree of such objects.

Command statements are also stored in a tree format. There is an object class, for instance, representing a Drop Element statement. The `DropElement` class contains the name of the table and column from which to drop the element, and the name of the dropped element. Another class, `Insert`, represents an Insert statement and contains the name of the table into which to insert the new rows, the columns into which to insert the data, and the root node of the query tree that represents the data to insert.

Since the commands and queries are stored as trees, we process them using a common programming paradigm: the *visitor pattern* [26]. The visitor pattern is, in brief, a way to traverse a tree structure, where all of the nodes in the tree are objects that implement a common subclass or interface. All of the object classes that represent the supported Guava statements derive from a common class called `Operator`, which has the following method:

```
public virtual void Visit(Transform t, Command c)
{
    t.Expand(this, c);
}
```

The first argument, `t`, is a channel transformation. The second argument, `c`, is a

pointer to the root node of the tree, which contains information such as a list of used table aliases in the current command (to avoid duplicate aliases within a tree). The class `Transform` is an abstract base class from which channel transformations derive. In the `Transform` class, there is a wide collection of methods implemented that appear like the following:

```
internal virtual void Expand(Insert i, Command c)
{
    i.Child.Visit(this, c);
}
internal virtual void Expand(Intersect i, Command c)
{
    foreach (Operator o in i.Children)
        o.Visit(this, c);
}
internal virtual void Expand(Join j, Command c)
{
    j.Child1.Visit(this, c);
    j.Child2.Visit(this, c);
}
```

There is one `Expand` method for each class that derives from `Operator`. These Visit and Expand methods, put together, effectively walk any command tree, because the method calls alternate between the two classes, visiting the children along the way. Because the tree is left unaltered by this algorithm, walking the tree in this fashion is

effectively a null transform; our implementation of each of the seven transformations extends the base class `Transform` and overrides only those methods that the transformation actually has an effect on.

For instance, consider the Pivot transformation. Looking at Table 3.8, we can see that Pivot has no effect on Rename Table statements. Therefore, the implementation of the `Pivot` class will not override the default transformation of Rename Table. However, the `Expand(Insert i, Command c)` method is overridden to translate the insert statement into an Error statement (Section 3.1.4), a Loop statement (Section 3.1.4), and an Insert statement (Table 3.1).

3.4.2 The Provider Model

A provider model is a way to generate implementation-specific syntax from an abstract representation of a language. In the case of channels, we use a provider model to translate a command tree that emerges from the channel into a concrete instance of a SQL command in a native SQL dialect. Most relational database systems share a common syntax for basic constructs like joins, projections, nested queries, and table aliases. However, each relational database management system may have a different syntax for constructs that are more complex, or constructs that are more recent in their addition to the SQL language. Examples of constructs that differ between flavors of the SQL language are pivots, unpivots, and the use of algebra and arithmetic. We have implemented a provider for the SQL Server 2005 database, which takes our command trees and translates them into the native SQL dialect, called Transact-SQL.

Our SQL Server provider is implemented as another visitor that traverses a command

tree. Here are some examples of the code in our SQL Server provider for translating
Rename Table, Drop Element, and Insert commands:

```
public SqlText Transform(AlterTableRename atr)

{

    SqlText sql = new SqlText();

    sql.Command = "EXEC sp_rename '" + atr.OldName + "', '"

                    + atr.NewName + "'";

    return sql;

}


public SqlText Transform(DropElement de)

{

    SqlText sql = new SqlText();


    // Determine if we need to set to null or to delete rows

    if (keyColumnsPerTable[de.Table].Contains(de.Column))

        sql.Command = String.Concat("DELETE FROM ", de.Table,

            " WHERE ", de.Column, "=", Common.Wrap(de.Element));

    else

        sql.Command = String.Concat("UPDATE ", de.Table, " SET ",

            de.Column, "=NULL WHERE ", de.Column, "=",

            Common.Wrap(de.Element));

    return sql;
```

```
}


public SqlText Transform(Insert i)

{

    SqlText sql = new SqlText();

    sql.Command = String.Concat("INSERT INTO ", i.Table, " (",

        String.Join(",", i.Columns.ToArray()), ") ",

                    i.Child.Visit(this).ToString());

    return sql;

}
```

Note that in the translation of an AlterTableRename object, the statement becomes the execution of a stored procedure "sp_rename", which is the SQL Server way to rename a table. In the code for translating an Insert, there is a call to "i.Child.Visit(this)", which is the visitor pattern at work; the result of the call will be the SQL corresponding to the query that creates the rows that are to be inserted.

3.5   CASE STUDY 1: CORI

In this case study, we compare the expressive power of our seven channel transformation against the middleware of a commercially available software application. Figure 3.9 shows part of the schema for the CORI application, version 4.0.23. This part of the schema corresponds to the part of the application that has been re-implemented using Guava components. In the CORI schema, there are no foreign keys; due to the introduction of artificial primary keys (e.g., proc_main_uid), one cannot introduce standard

relational foreign keys in their DBMS. So, the foreign keys must be enforced by the application code in CORI.

Figure 3.10, on the other hand, shows the natural schema of the Guava-ized CORI sample. The figure shows two types of foreign keys. The first are the type that have a graphic of a key on at least one end. These keys correspond to a standard foreign key; the foreign keys with two "key" graphics are one-to-one foreign keys between `id` columns, and the keys with only one key graphic are one-to-many foreign keys between a `fk` column and an `id` column. The second kind of foreign key is represented as a solid line with arrows at the end (there are two in the figure, one black and one grey). This second kind of foreign key is a Tier 2 foreign key with disjunction (from Section 3.1.3). For instance, the foreign key from `proc_find` states that the entries in `proc_find.id` must be found in either `COL.id` or `EGD.id`.

We construct a channel to conform the schema in Figure 3.10 to the schema in Figure 3.9 as best we can, maximizing the number of matching columns, tables, and domains (for brevity, we abbreviate some of the arguments and explain later):

```
Apply(patient, {White, Asian, HawaiianPacificIslander,
        Black, NativeAmerican, OtherEthnicity}, {race}, F)
HMerge({COL, EGD}, proc_data, procType)
VMerge(proc_data, SurgicalHistory)
VPartition(proc_data, proc_strn, C1)
VPartition(proc_data, proc_text, C2)
Apply(proc_data, C0, C0, F0)
Apply(proc_strn, C1, C1, F1)
```

proc_text
- 🔑 proc_text_uid
- procedure_uid
- finding_uid
- item_id
- required
- addid
- delid
- textval

proc_strn
- 🔑 proc_strn_uid
- procedure_uid
- finding_uid
- item_id
- required
- addid
- delid
- stringval

proc_data
- 🔑 proc_data_uid
- procedure_uid
- finding_uid
- item_id
- required
- addid
- delid
- numericval

proc_main
- 🔑 proc_main_uid
- procedure_uid
- patient_uid
- procedure_date
- procedure_type
- signed
- addendum_signed
- addid
- delid
- diff
- site_id
- proc_num

proc_find
- 🔑 proc_find_uid
- finding_uid
- procedure_uid
- caption
- points
- x
- y
- txtLeft
- txtTop
- kind
- regions
- addid
- delid
- diff
- procType

patient
- 🔑 patient_uid
- lastname
- firstname
- midname
- DOB
- MRN
- SSN
- gender
- race
- address1
- address2
- city
- state
- country
- zip
- phone1
- phone2
- notes
- source
- payor_id
- site_id
- deleted
- lastchanged

Figure 3.9: Part of the relational schema for the CORI application, version 4.0.23
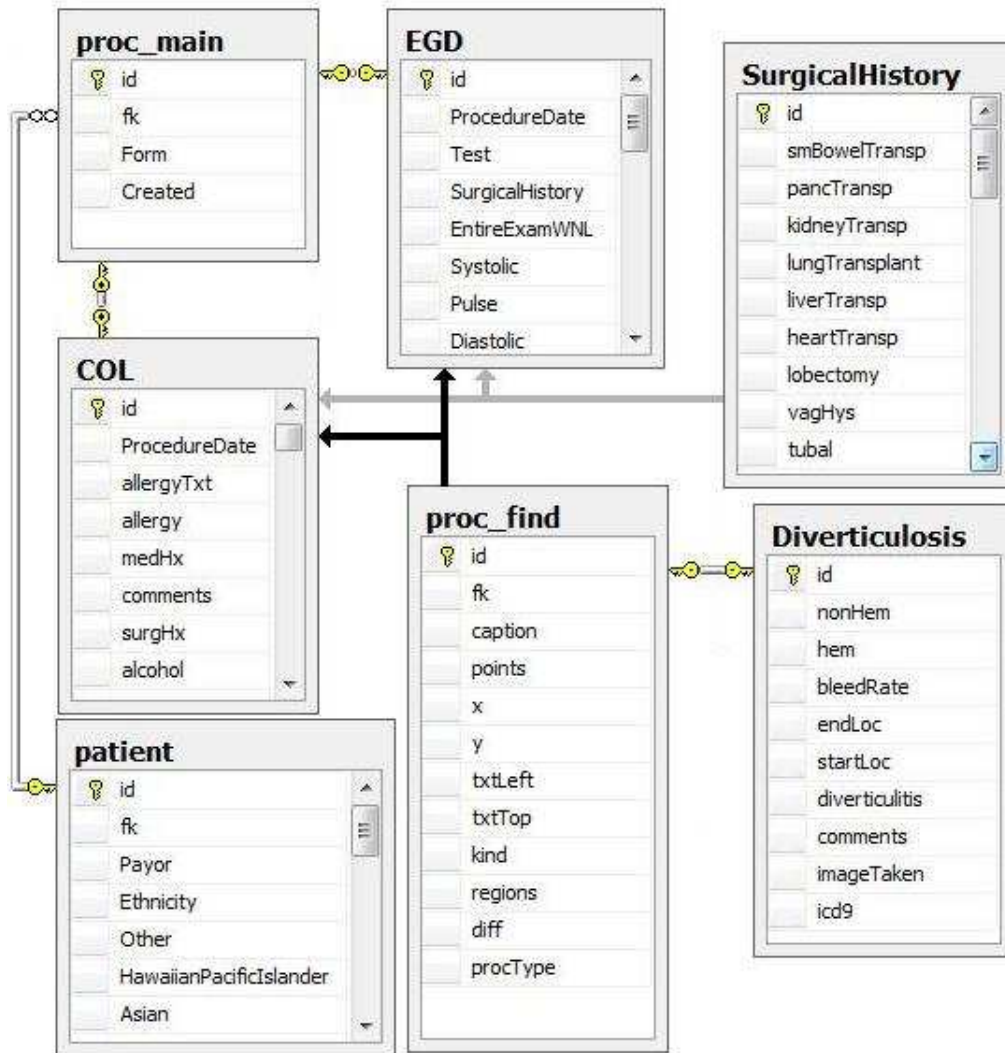
Figure 3.10: The natural schema for our Guava implementation of part of CORI

```
Apply(proc_text, C2, C2, F2)

Unpivot(proc_data, item, numericval)

Unpivot(proc_strn, item, stringval)

Unpivot(proc_text, item, textval)

HMerge({Diverticulosis}, find_data, findType)

VPartition(find_data, find_strn, C4)

VPartition(find_data, find_text, C5)

Apply(proc_data, C3, C3, F3)

Apply(proc_strn, C4, C4, F4)

Apply(proc_text, C5, C5, F5)

Unpivot(find_data, item, numericval)

Unpivot(find_strn, item, stringval)

Unpivot(find_text, item, textval)
```

Function F is a function that takes the value of each of the race and ethnicity values and encodes them as a string value. So, if a row has values of "true" for white, other, and ethnicity (meaning Hispanic), and values of "false" for all other columns, it will have a value of "WOH" for race in the output.

Column sets C0, C1, and C2 are the columns in proc_data that have numeric domains, short text fields (100 characters or fewer), and long text fields (longer than 100 characters) respectively. The functions F0, F1, and F2 are domain-alignment functions (since each column set is domain-compatible within itself, but not completely equivalent). For instance, the function F0 takes all of the values in columns C0 and translates them into 32-bit integers. These functions are only necessary because the formalism and

implementation of Unpivot require that the domains of all non-key columns be identical. Note that these functions are tedious to write. A simple solution to this problem is to extend the definition and implementation of Unpivot to automatically align the domains of the non-key columns to a domain that is castable from each input domain.

The column sets `C3`, `C4`, and `C5`, and the functions `F3`, `F4`, `F5`, are similar to the column sets `C0`, `C1`, and `C2` and functions `F0`, `F1`, `F2`, except they pertain to the finding tables instead of the procedure tables.

The result of applying the channel above to the schema in Figure 3.10 is the schema in Figure 3.11. The result is not identical to Figure 3.9. Comparing the two schemas, all of the information that is present in the Guava version of CORI's database is present in CORI's native database. However, there is some information in CORI's native database that is not present in the Guava analog. For the rest of this section, we enumerate and discuss the differences between the two figures.

The differences between the two schemas fall into six categories:

- Additional columns in the original CORI schema that hold new data not contained in the natural schema

- Additional columns in the original CORI schema that hold temporal data about each row

- The `item_id` columns in the procedure tables in the original CORI schema, which are present in the Guava version but in a different form (`item` and `procType`)

- Primary key columns serving as surrogate key columns in the original CORI schema (for instance, `proc_main.proc_main_uid`)
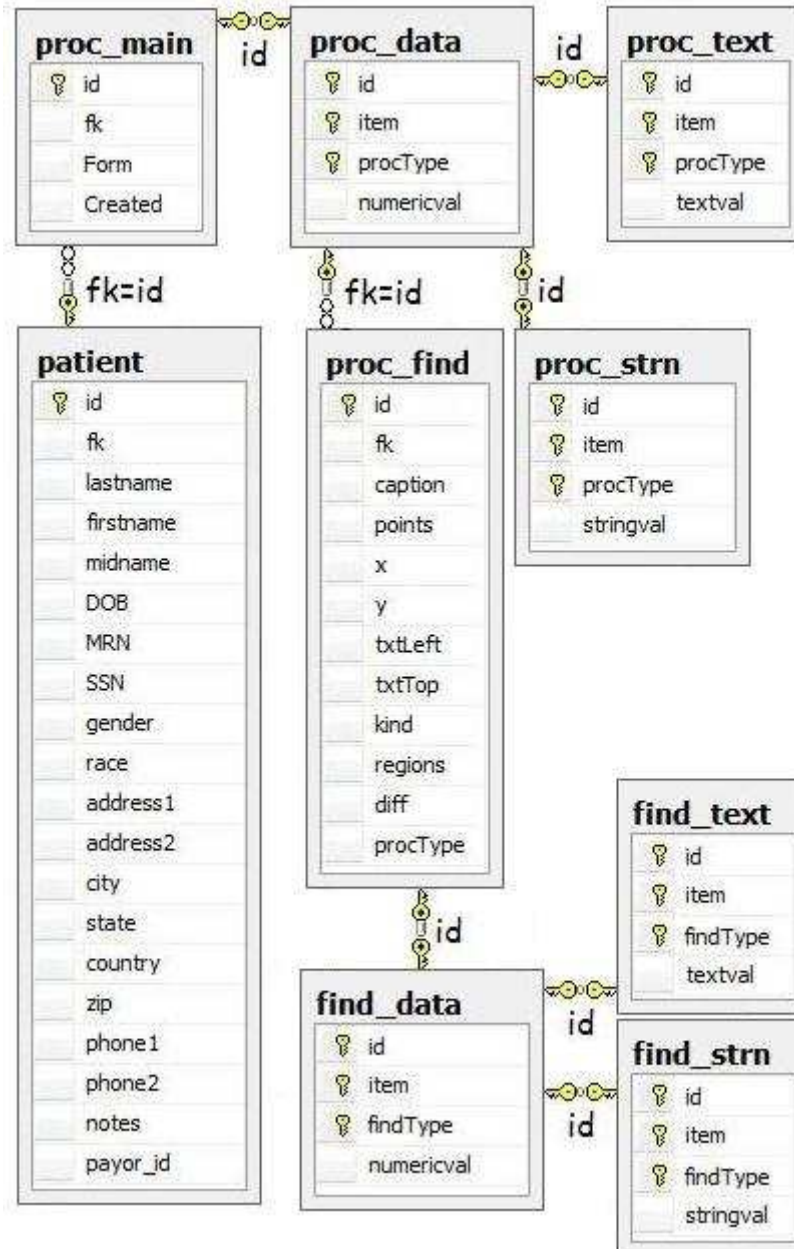
Figure 3.11: The schema from Figure 3.9, after applying a channel

- The separation between procedure ("proc") tables and finding ("find") tables, which are both stored in the procedure tables in the original CORI schema

- Columns that are present and identical in both versions but do not have the same name — this case is trivial, since it can be overcome simply by renaming the columns in the original schema (for instance, `patient.patient_uid` becomes `patient.id`)

For each of these categories (except the last one), we consider what it would take to overcome these differences in terms of additional transformations, ones that cannot be expressed in terms of existing transformations.

As an example of the first difference category, consider the field `proc_main.signed`. This field represents the ID of the user that signed the procedure. This information is present nowhere in the natural schema (and, in fact, nowhere in the procedure forms in the user interface). This information is drawn from the environment; as part of the business logic of the application, CORI records the ID of the user that is logged in at the time of signing for later accountability, and so that information can be included in printouts of the procedures. Other similar columns include the `required` columns in the procedure tables (recording whether a particular field was required at the time of persistence), the `addendum_signed` column of `proc_main` (storing the user ID of the last user to sign changes to a procedure after it was originally signed), and the `site_id` column of `proc_main` (storing the value of the `site` variable at the time the record is saved). We have abstracted this kind of addition of environmental information as the *Adorn* channel transformation, described in Chapter 4.

The second category of difference refers to the `addid` and `delid` columns present in

all of the procedure-related tables. These two additional columns do more than simply store environment data: They alter the way that updates and deletes are handled by the system. Together, the `addid` and `delid` columns represent the lifespan for each row. When a record is inserted into one of these tables, the new row receives a timestamp value representing the current time into the `addid` column, and a null value into the `delid` column (representing an unknown or non-existent end time). When the CORI application tries to delete a row, the CORI middleware transforms that instruction into an update — to take the deleted row and instead "deprecate" it by setting its `delid` value, effectively ending the row's lifespan. Likewise, an update from the application takes the original rows and sets the `delid` value on them, then inserts new rows with the new value (and null `delid` column value). We have abstracted this treatment of attributes by defining a channel transformation called *Audit*, described in Chapter 4.

The `item_id` columns in the procedure table store an integer value. This value is a key into a lookup table called `Control`, which is a single table that stores information about all of the controls from procedure and finding screens in the entire CORI application. The corresponding information in Figure 3.6 is stored in the `item` and `procType` or `findType` columns, information which is also located in the `Control` table. Given a value for `item_id`, there is a unique pair of values for `item` and `procType`/`findType`, and vice versa. We have captured this type of transformation, in either direction, in the *Lookup* transformation, described further in Chapter 4.

The surrogate key columns in the procedure tables in Figure 3.9 do not appear anywhere in Figure 3.11. For example, CORI developers added the column `proc_main_uid`

to table `proc_main` as a primary key column because the primary key would have otherwise spanned several columns, and the developer preferred to work with single-column primary keys. The surrogate keys do not participate in any queries or updates issued from the CORI application; all of the queries and updates issued by the application refer to the `procedure_uid` and `finding_uid` columns. Therefore, one possible solution for the current application is to drop the surrogate keys and to assert other combinations of columns as the new primary keys for each table. However, to accommodate the current layout of the CORI application without alteration, we would require a new transformation that would *establish surrogate keys* for procedure tables, where each inserted row is assigned a new, globally-unique value.

Finally, we consider the case of merging procedure data with finding data (the fifth difference category in the list). In the CORI database, the procedure data and the finding data is stored in the same set of tables. Each of the procedure data tables contain three different fields that hold globally-unique identifiers: procedure_uid, finding_uid, and an additional field named after the current table (a surrogate key). Data in each of these tables can be broken into two sets of rows: those rows that hold data pertaining to a procedure (in which case the finding_uid is null) and those rows that describe a finding attached to a procedure (in which case the finding_uid is not null). Since a primary key cannot be used on columns that can contain a null value, the additional GUID field is used as an artificial key, which is used for no other purpose. To emulate the existing database in Figure 3.9 from Figure 3.11:

- For each of the procedure data tables (data, strn, and text), change the key to be the columns `proc_uid` and `find_uid`. Place the old value of `id` in the `proc_uid`

column, and set the `find_uid` value to be the all-zeroes GUID.

- For each of the findings data tables (data, strn, and text), change the key to be the columns `proc_uid` and `find_uid`. Place the old value of `id` in the `find_uid` column, and set the `proc_uid` value to be the result of $\pi_{id}\sigma_{fk=G}proc\_find$, where G is the GUID of the current finding.

- Union each pair of like tables (data, strn, and text) together.

Collectively, we call this transformation *FindingMerge*.

None of these five additional transformations can be expressed using the transformations in Table 3.2. Hence, Figure 3.11 is as close as we can come to Figure 3.9 without extending our transformation language. In Chapter 4, we introduce new transformations that cover Adorn, Audit, and Lookup; in Chapter 7, we describe an extensibility mechanism by which the developer can write new transformations, including Establish Surrogate Keys and FindingMerge.

3.6 PERFORMANCE ANALYSIS

One hypothesis that we posit is that the overhead of the channel is dominated by the overhead of accessing the physical database. In other words, the time spent performing queries and updates in the database is significantly more than the amount of time that the channel spends processing the queries and update statements. This property of a channel is important because if the channel introduces as much overhead as the physical database (or worse, more overhead than the database) then it becomes the dominant source of latency in the application; developers would then opt to develop a custom,

hard-coded alternative to the channel to improve response time.

When it comes to interaction between user and interface, performance becomes a subjective quality, as many factors contribute to the user's experience, even including time spent typing. Different people will have different tolerance and expectation for what is an acceptable amount of time for a form to appear or for an information request to complete. As a baseline, we tested how long it takes to perform an insert of a procedure using the CORI application, version 4.0.23. The "insert" comprised data for a colonoscopy, some surgical history, and a finding of diverticulosis. We timed how long it took to perform four such inserts, and broke the timing results into two parts: time spent in the middleware (restructuring the insert statements), and time spent in the database (actually issuing the inserts). The results demonstrated that approximately four times as much time was spent doing operations in the middleware than was spent in the database. These timings were very informal and potentially misleading; the CORI middleware code includes functions that have nothing to do with data transformation, such as data validation, lock management, and PDF generation. However, it is informative to note that the overhead of the middleware and the overhead of the database were within an order of magnitude of one another with respect to time; the tests were also run over a very small database (thousands of rows per table), so over a larger database (millions of rows per table) the database overhead will be significantly larger with respect to the middleware.

To test the overhead of a channel, we took the sample application we built as a Guava analog to CORI 4.0.23 and tested it against a particular workflow. Specifically, we timed bursts of 100 procedure inserts, where each procedure insert comprised data for a

colonoscopy, a surgical history, and a finding of diverticulosis. We also timed bursts of 100 queries, where each query is `SELECT * FROM COL WHERE id=g`, for some GUID g that we know to represent a colonoscopy that exists in the database. This kind of query is indicative of the kind of query issued as part of an application, i.e., retrieving a single entity from a database for display on a form. We tested both of these workflows under two different conditions:

- The application has an empty channel (in other words, the physical schema is the natural schema)

- The application has a channel that includes an HMerge between the COL and EGD tables, a VMerge between the merged table and Surgical History, a VSplit to split off some columns, and an Unpivot on the split-off columns

The results of our analysis are in Table 3.11. The "empty channel" tests were run a total of 3333 times, and the "with channel" tests were run a total of 7582 times. Tests were initially run on an empty database. Table 3.11 shows the averages and standard deviations of the timing results of those tests, broken into three components:

- Transformation time: the time spent physically within the channel processing the insert or query statements (and their intermediate products)

- Provider time: the time spent by the provider translating the statements in a transaction into SQL Server-specific syntax

- Database time: the time spent within the database executing the statements

Table 3.11: Measuring the performance of inserts and queries against a natural schema, both with an empty channel and a non-empty channel

Scenario	Transformation Time		Provider Time		Database Time	
	Mean (ms)	Std. Dev.	Mean (ms)	Std. Dev.	Mean (ms)	Std. Dev.
Insert, empty channel	2.87	0.54	8.39	2.36	2456.53	916.24
Insert, with channel	26.76	1.89	15.76	1.74	3992.78	657.90
Query, empty channel	1.19	0.48	1.65	0.74	101.92	66.16
Query, with channel	48.22	3.74	35.40	2.85	123.10	67.54

The key observation to come away with from the results in Table 3.11 is that average transformation time and average provider time put together (in other words, the total overhead of the channel, or channel time) is less than the average database time for all cases. In the case of inserts, with or without transformations in the channel, that difference is several orders of magnitude. In the case of queries, the difference is less drastic in the presence of channel, but the channel time is still less than the database time. Recall that these tests began with an empty database; with a larger database, the channel time will remain constant, while the insert and query times will increase due to more disk or index accesses. This result confirms our hypothesis: channel overhead is dominated by database overhead.

3.7 CASE STUDY 2: INFOSONDE

As a second case study, we compare the expressive power of our seven transformations against another tool that has a similar atomic transformation paradigm. We thus also

Start with a pre-existing database instance for analysis

Write and execute a query that tests for conditions that indicate a transformation T might be useful

If query returns suitable results, create a channel transformation T and push the database instance through it; if not, try another query

Repeat until no more transformations are possible with the current data

Combine transformations into a channel; re-execute the channel if original database instance changes, or push statements from against the DB in real-time, as described in this chapter

Figure 3.12: Building a channel one transformation at a time, and fully instantiating the database along the way; this workflow is very similar to how the InfoSonde workflow currently operates. The final row of this figure demonstrates how a channel could respond to changes in the left database in two different ways

demonstrate how to use channels in a scenario independent of the GUI tools of Chapter 2, and unlike any of the usage patterns described in the previous section. In this case study, we examine using a channel in a scenario similar to Potter's Wheel [65], where a data expert applies a transformation at a time to a database instance. Note that this is a scenario where we are using a channel without any of the GUI components from Chapter 2.

Our target application in this study is the InfoSonde tool [34]. InfoSonde is part of a

larger effort to provide tools that allow a data expert to do empirical schema discovery over an unfamiliar, schemaless database. InfoSonde in particular works in a way similar to that shown in Figure 3.12. In this figure, we also show how we support the InfoSonde workflow by building a channel. A database user begins by being handed a database. The data may be stored in a generic format and there may be little or no information about functional dependencies, keys, and foreign keys exists. The user then proceeds to apply tests to the database; if a test returns a positive result, one of the following situations has occurred:

- The test demonstrates that there is a transformation that can be applied to the database that would make the data instance "better" in some sense, e.g., by pivoting the data out of a generic format.

- The test demonstrates that a foreign key, primary key, or functional dependency exists that is not currently expressed in the database metadata, or would exist were it not for some small amount of dirty data.

An example query for the first situation is `SELECT (SELECT count(DISTINCT Y)` `FROM T)/(SELECT count(*) FROM T)`. If the result of this second query is small (say, less than 0.05), and the column `Y` contains text (string) values, and also the column `Y` can be considered as part of the primary key for `T`, the column `Y` may be a prime candidate for either a horizontal split or a pivot. For example, consider the table in Figure 3.13(a). There are two columns in the table that are marked with an arrow; each of these columns stores data that represents the source of the data, and likely has a small, finite active domain and thus may be an enumerated domain. These two columns would

then be prime candidates for either horizontal partitioning or pivoting (serving as the attribute column).

An example query for the second situation is the query `SELECT count(count(*))` `FROM T GROUP BY X HAVING count(*) > 1`. This query returns the number of values in the `X` column that can be found in more than one row; if the result of this query is zero, then `X` appears to be a key for the table. This query can be extended easily to any number of columns in place of `X`. For example, consider the tables in Figure 3.13(b). The two different arrows represent two possible foreign keys between the two tables. Using some probing queries, we can determine if a foreign key already exists naturally between the two tables. We can also determine if a functional dependency exists between the two columns `lblcode` and `firm_seq_no` using a different query. It may also be the case that a foreign key or functional dependency may exist, except for a relatively small set of (potentially erroneous) rows; in this case, InfoSonde can partition the `listings` table in two: those rows that satisfy the dependency or foreign key, and the small set of those that do not.

Using InfoSonde tools, a user repeatedly applies a specific workflow:

- Apply a test (or series of tests) to the database

- When a test passes, consider applying a schema change or transformation to the database instance as a whole

- Repeat until no more new tests pass

We can use Guava to support InfoSonde's workflow in two ways. First, both Guava

(a)

(b)

Figure 3.13: Examples of tables that may exhibit schema-like characteristics without those characteristics being explicitly present in metadata; one may use InfoSonde to determine that two different columns in a table are finite-domain (a), or that a foreign key can be enforced between two tables (b) where a foreign key does not yet exist

and InfoSonde require a finite set of well-defined, encapsulated database transformations. The standard way to use Guava channels is to connect a virtual database instance to a concrete one, acting as a view definition. To support InfoSonde, we allow a developer to apply transformations one-at-a-time to an instantiated database instance, as shown in Figure 3.12. One can use the copy insert transformations described in Figure 3.7 to translate an entire database instance through a channel in either direction. We start with a materialized database instance that is effectively in the natural schema (Option 2 in Figure 3.7). Next, we create a separate database that has the schema of the desired physical schema. Finally, we create copy insert statements for each of the tables in the natural schema, push them through the channel, and execute them.

In addition, the list of channel transformations and the list of transformations required for the InfoSonde project overlap; however, neither set of transformations subsumes the other. In common between the two lists are the Pivot, HPartition, and HMerge transformations. Second, Guava's data model supports a more expressive form of referential integrity; using InfoSonde, one may be able to construct a test query that can expose the existence of Tier 2 foreign keys, which may in turn provide evidence for further schema modifications or transformations.

The collaboration between InfoSonde and Guava is ongoing. The rest of this section outlines the lessons that we have learned from our collaboration thus far.

The queries that the user issues as part of the iterative schema discovery process invariably include aggregation and grouping features. These features are not a part of Guava's supported query language in the current implementation. There are two workarounds for this situation; the simple option is to have the user issue a query using

the existing query language, then import the results of that query into a relational engine and evaluate the aggregation separately. This workflow is simple to implement from a user interface perspective, and requires no extension of the underlying architecture.

The second option is to extend the Guava query language to include aggregation and grouping. Doing so would require adding code for new operators, and updating the SQL Server provider to be able to translate the new operators into native SQL code. However, due to our use of the visitor pattern in our code, one would not need to update the implementation of any of the channel transformations. We would also not need to update the formalism behind Guava at all. The channel transformations process queries by view unfolding, which does not affect any other operators in the tree. Said differently, adding new operators to the query language does not affect the closure property of the supported Guava statements.

There are two transformations that InfoSonde requires that we do not yet provide:

- Predicate Partition (PPartition): Break up a table into multiple tables based on whether each row satisfies a predicate, similar to a generalized version of HPartition.

- Normalize: The standard normalization transformation along a transitive functional dependency. If a table T has a functional dependency $X \rightarrow Y$, where X and Y are collections of non-key columns, create a new table $T'(\underline{X}, Y)$, let $T' = \pi_{X \cup Y}T$, and let $T = \pi_{\mathbf{Cols}(T) - Y}T$.

In Chapter 4, we describe how to add PPartition to our set of supported transformations. In Chapter 7, we describe how a developer can add additional transformations, including Normalize. Functional dependencies are not explicitly part of the Guava model,

or the model of most relational database engines. Therefore, the Normalize transformation is unique among transformations in both Guava and Infosonde in that it is only information preserving if it assumes a condition on the data to be true in the conceptual schema that it cannot statically check (compared to, for instance, Vertical Merge, which requires that the merged tables have a foreign key defined between them that can be verified as it is a first-class entity in the relational model).

3.8 RELATED WORK

Relational Lenses [9] is an approach that uses an operational list of components to address the view update problem. A lens is bi-directional in the same way as a channel operator: each lens describes how to translate data in one direction and how to unambiguously update data in the reverse direction. The current set of operators that we consider and the set of lenses have different expressive power, though neither subsumes the other. Lenses, as well as other approaches to data independence [79], use relational algebra or something similar to define their views, and focus entirely on views expressed using conjunctive queries. Relational lenses support arbitrary joins. On the other hand, lenses do not handle schema-level changes in the current literature. Channel transformations do not mirror relational algebra operators; rather, each transformation encapsulates a higher-level restructuring of schema and data that has known properties, including updatability and accommodation of schema-level changes.

PRISM [17] and PRIMA [58] are two frameworks by the same research group that provide mechanisms for incremental schema changes. Both PRISM and PRIMA support low-level schema changes such as adding or dropping columns and renaming tables.

PRISM goes a step further and provides five high-level transformations that have direct analogs to the Apply, HPartition, HMerge, VPartition, and VMerge transformations in Guava. One uses a sequence of these Schema Modification Operators (SMOs) to describe the relationship between two versions of a physical database; one can then pose a query against one version of the database, and have that query automatically translated into equivalent forms that are valid and correct on other versions of the database. In this way, SMOs between two versions of a database acts like channel transformations between a natural schema and the physical database in Guava. SMOs cannot propagate DML updates or DDL updates, since once a version of a database has been made obsolete, the contents and schema of that version become static. There is also no SMO corresponding to Pivot or Unpivot.

There are several more areas of research that aim to describe data transformation using a collection of discrete, well-understood components [30]. Many of these projects treat such transformations as uni-directional and to be executed periodically as a batch process. The most prominent such technology is arguably Extract-Transform-Load (ETL). Both academic research [80] and commercial products [52] model ETL processes as a workflow of operations, where each operation comes from a finite menu of components. Another such paradigm is database refactoring, which refers to the process undertaken by a software engineer or database administrator to change the design of a database without changing the underlying semantics of the data. Ambler and Sadalage [3] present a list of atomic operations, each of which is a refactoring. Finally, Potter's Wheel [65] is a data cleaning and visualization tool that allows the user to apply operators to "sculpt" data dynamically to fix data errors and to reshape schemas. One can

then save Potter's Wheel operators and compile them into a script that resembles an ETL process.

There is much functional overlap between our channel transformations and the operators in these projects; for instance, Potter's Wheel includes pivot and unpivot operations, and Ambler's refactorings list includes partitioning. However, none of these projects can handle active communication between schemas. One can think of a channel as an ETL process or a refactoring that is bi-directional and dynamic. Also, none of these projects can handle propagation of schema changes through their constructs.

Some projects use declarative mappings rather than discrete operators to establish a relationship between two schemas. One approach is to compile mappings into a one-way transformation (e.g., Clio [56, 57]). The other approach is to compile mappings into a bi-directional transformation or two opposite-facing one-way transforms [49], which provides updatability at the cost of the expressive power of the mappings. While the declarative approach is higher-level than specifying operators or workflows, the expressive power of such mappings is generally limited to various flavors of joins and unions. Most mapping languages cannot express functions or express a pivot or unpivot. Clio is of particular note because one can associate a function with a mapping, whose effect is similar to our Apply transformation, and recently introduced constructs that can do pivot- and unpivot-like transformations [32]. However, Clio's flexibility comes at the cost that its mappings are not updatable, and it cannot handle schema evolution.

We introduced the Apply operator to handle attribute transformations; our Apply operator necessarily needs to use an invertible function so as to meet the invertibility requirement of Guava transformations. Larson et al. considered the problem of attribute

and values equivalences extensively, and provides a formal treatment of the kinds of value transformations that are invertible [41].

The COIN project [70] focuses on describing attributes with context elements, such as units. Guava captures context information about each user interface control, such as the label that appears on the screen. We do not focus specifically on the kind of context elements considered in the COIN project, such as units for a value, but such context elements could be easily incorporated in Guava.

There have been several approaches to try to extend the SQL query language to include enough expressive power to cover pivoting data. SQL Server 2005 [53] added a PIVOT ON clause to identify a pivot column. SchemaSQL [40] introduces relation and column variables that can produce the effect of pivoting. FISQL [83] is a variety of SQL that provides language features similar to SchemaSQL's relation and column variables, but with cleaner underlying semantics. Using these approaches, one can construct views that have similar capabilities as our pivot and unpivot operators, and relation variables provide a similar functionality as horizontal merging and partitioning. None of these approaches currently address the update problem, either for data or for schema.

FISQL uses as its formal foundation work from an earlier paper [82] that introduces five atomic operators as extensions to relational algebra. One can express pivot and un-pivot as compositions of these five operators. Similar to SchemaSQL, Wyss's version of pivot and unpivot does not assume a static schema; a pivot operator may produce a variable number of output columns depending on the active domain of the pivot column in the input instance. Also similar to SchemaSQL, FISQL cannot be implemented

in a database system without re-implementing the underlying model and all other operators in a DBMS because their version of pivot operates on dynamic schema. Both SchemaSQL and FISQL push as much of a query down to a static-schema relational database to leverage its relational engine, then implement the dynamic-schema portions in a separate module. Guava provides a solution that is a compromise between static and dynamic schema; we assume a static schema for the relational database and for the input and output of each operator, but we accommodate schema changes at the element, column, and relation levels.

Object-relational mapping tools (ORM's) [33, 49, 73] are an increasingly popular way to connect client data structures with persistent data. These tools typically provide a default mapping between a set of classes and a set of tables, and automatically moving data between the two. ORM's also come with a way to query objects, and translate those object queries into relational queries. The level of mapping customizability and expressive power varies between ORM's. Typically, the mappings used in ORM's can support selection, projection, and limited join and union, and cannot handle more complicated transformations such as pivoting.

In ORM's, there are three popular paradigms for storing data from objects in tables. First is *Table-per-type*, where there is one table in the database for every object class; if class A is a superclass for B, the table for B only stores the data for attributes unique to B. Second is *Table-per-concrete-type*, where there is still one table per class, but data is stored differently; if class A is a superclass for B, both A and B's tables have columns for the attributes of A. The only objects that are stored in the table for class A

are those objects that are in class A and no other subclass. Finally, there is *Table-per-hierarchy*, which stores all of the data for an entire class hierarchy in the same table; if an object is type A, in the table the row for that object will have a value of null for any attributes that do not belong to class A. There is also a provenance column added to indicate the type of the object in each row. In addition to these three paradigms, Mork et al. present a flexible method for translating an object hierarchy into persistent storage structures [59]. Though Guava does not explicitly support object hierarchies, if one treats each object class as a relation in the input schema of the channel, Guava can support all of the above paradigms as a combination of GVPartition, HMerge, and PPartition transformations. We introduce GVPartition, and fully define GVPartition and PPartition, in the next chapter.

Federated database systems provide another avenue for research into database mappings. In a federated database, there is a single schema (the *global schema*) that serves as the public view of the data in the federation, and a collection of schemas (the *local schemas*) corresponding to the physical data stored in various databases. The local databases are likely heterogeneous in the sense that like objects are probably not stored in like schemas, exhibiting conflicts of various kinds [72]. There are several approaches to establishing relationships between the global schema and the various local schemas; each approach provides a mechanism to rewrite queries expressed against the global schema into queries against the local schemas [42].

Global-as-View (GAV) expresses the global schema as a set of queries over the local schemas. Local-as-View (LAV) expresses each local schema as a set of queries over the global schema. Global-and-Local-as-View (GLAV) is a combination of the two;

the relationships between global and local schemas are expressed by statements in the form $Q_l \subseteq Q_g$, stating that the result of query Q_l over a local schema is contained in the result of query Q_g over the global schema. One final approach, called Both-as-View (BAV), describes the relationship between the global schema and a local schema as a sequence of discrete transforms, each adding, modifying, or dropping tables according to transformation rules. BAV has been demonstrated on both a high-level hypergraph data model [62] and the relational model [47]. All four of these approaches employ a query language as their mapping language; therefore, the expressive power of the mappings provided by any of these approaches depends on the chosen query language. For instance, Miller [55] uses a restricted form of SchemaSQL as the query language in a GAV setting, allowing unpivot-like operations from a local schema to the global schema.

Federated databases offer no guarantee of information preservation; query translation in federated databases follows *maximally contained* semantics, in that the resulting query post-translation offers the most complete set of results possible. Also, because all relationships in these approaches are expressed using views, processing of updates is handled in a similar fashion as in the materialized view literature [28] and view-updatability literature [19]. The ability to update through views, materialized or otherwise, depends heavily on the query language. Constructs such as unions are considered difficult, and pivots and unpivots are not considered; these operators are ambiguous to update in the general case. Guava allows specific instances of these operators in a channel and can still translate updates because Guava transformations are chosen to be updatable unambiguously.

3.9 SUMMARY

A channel is a tool for establishing a mapping between two schemas, effectively con-
necting a virtual schema to a physical schema, similar to a collection of views. A chan-
nel comprises a sequence of developer-selected transformations that are information
preserving. Once a channel has been created, the channel receives as input statements
issued against the virtual schema and translates them into equivalent statements issued
against the physical schema. Performance results illustrate that our implementation of
channel transformations are efficient enough for run-time applications, using the exist-
ing CORI 4.0.23 application as an objective benchmark. We performed case studies
that compared the expressive power of the seven transformations defined in this chapter
against an existing application middleware as well as a tool that performs data transfor-
mation.

Chapter 4

EXTENDING THE EXPRESSIVE POWER OF CHANNELS

The channel is a tool that introduces data independence between a relational schema and an underlying physical schema as a sequence of encapsulated transformations from one schema to another. As demonstrated in the case studies, the seven transformations in Table 3.2 are useful for modeling the kinds of transformations that exist in applications, but are not expressive enough to fully describe those transformations. In this chapter, we extend the expressive power of channels by introducing new transformations.

One assumption that the we made in the previous chapter regarding channels is that they should be transparent to any application accessing the natural schema. This assumption stems from the fact that all of the transformations from Chapter 3 represent physical design decisions, which are traditionally transparent to the user. Said another way, one of the benefits of data independence is precisely that the application can treat the virtual schema (in our case, the natural schema) as if it were the physical schema without having to worry about the data access mechanics. Some of the new transformations that we introduce in this chapter relax this assumption because they perform transformations that an application may want to know about. For instance, a transformation might add new data fields as it travels through the channel, and the application developer may choose to have the query interface present those fields to the user.

Another assumption that we make in Chapter 3 is that the information-preservation

property implies locality of updates. In other words, if one issues a DML statement against table T, one should not see any effects to any table other than T in the natural schema, unless there are cascading deletes involved. A developer may have additional knowledge about the extent of the natural schema. For instance, a developer may know that tables T and T' actually refer to the same data, as if they were both views over the same table. In that case, updates against T will have effects on T'. Some of the transformations in this chapter act as a developer directive stating that there exists a relationship between schema elements in the natural schema that will intentionally cause side effects.

In this chapter, we extend the expressive power of our channel transformation language in three ways. We discuss each of these research contributions separately:

- We demonstrate how to generalize three of the existing transformations: Pivot, VPartition, and HPartition.

- We introduce a new class of transformations — called *application-specific transformations* — that correspond to business-logic decisions rather than physical design decisions. Application-specific transformations provide information, on request, about the changes that they make to data.

- We introduce a third class of transformations called *correspondence assertions* that have applications in information integration. Correspondence assertions also modify the definition of *information preservation* to be relative to developer directives; for instance, if a developer asserts that tables T and T' refer to the same data, then a new row in T will also appear in table T'. This scenario violates our

FromA (Output Schema):

ID	Source	Target	Data
1	A	X	11
3	A	Y	13

Items (Input Schema):

ID	Source	Target	Data
1	A	X	11
2	B	X	12
3	A	Y	13
4	B	Y	14

FromB (Output Schema):

ID	Source	Target	Data
2	B	X	12
4	B	Y	14

ToX (Output Schema):

ID	Source	Target	Data
1	A	X	11
2	B	X	12

PPartition
(Items, {FromA, FromB, ToX, ToY},
{Source, Target}, {Source=A,
Source=B, Target=X, Target=Y})

ToY (Output Schema):

ID	Source	Target	Data
3	A	Y	13
4	B	Y	14

Figure 4.1: An example of the PPartition transformation acting on instances

original definition of information preservation (no unexpected side effects) but is valid relative to the assertion.

4.1 GENERALIZED TRANSFORMATIONS

The definitions of the transformations in Tables 3.3 through 3.9 are based on how we have seen these transformations in use in applications. Case study 1 in the previous chapter demonstrated that the transformations, as defined, are sufficient for the physical-design part of CORI's database. Anecdotally, the transformations were also sufficient to support several other applications we looked at. However, we make the following observations:

Data (Input Schema):

ID	A	B	X	Y
1	c	e	11	22
1	d	f	12	23
2	c	e	13	24
2	d	e	14	25

Data (Output Schema):

ID	X_c_e	X_c_f	X_d_e	X_d_f	Y_c_e	...
1	11			12	22	...
2	13		14		24	...

MultiPivot (Data, {A,B}, {X,Y})
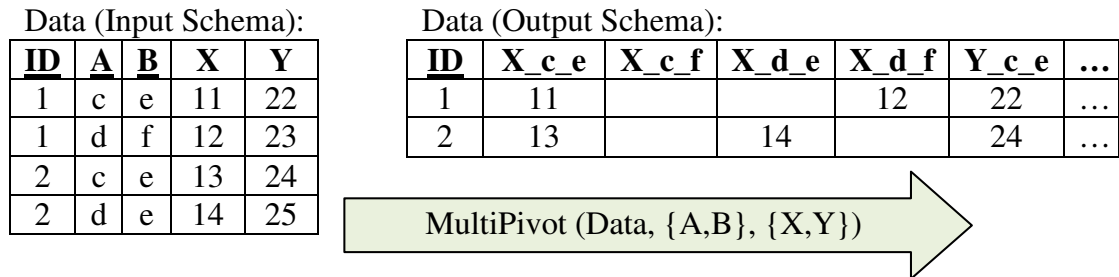
Figure 4.2: An example of the MultiPivot transformation acting on instances

Patient (Input Schema):

ID	Name	City	State	Insured	Prov
1	Bob	Boston	MA	T	102
2	Ted	Miami	FL	F	188

Patient (Output Schema):

ID	Name	Prov
1	Bob	102
2	Ted	188

GVPartition (Patient, {PatientA, PatientB},
f: f(Patient) = {Name, Prov},
        f(PatientA) = {City, State},
        f(PatientB) = {Name, Insured})

PatientA (Output Schema):

ID	City	State
1	Boston	MA
2	Miami	FL

PatientB (Output Schema):

ID	Name	Insured
1	Bob	T
2	Ted	F
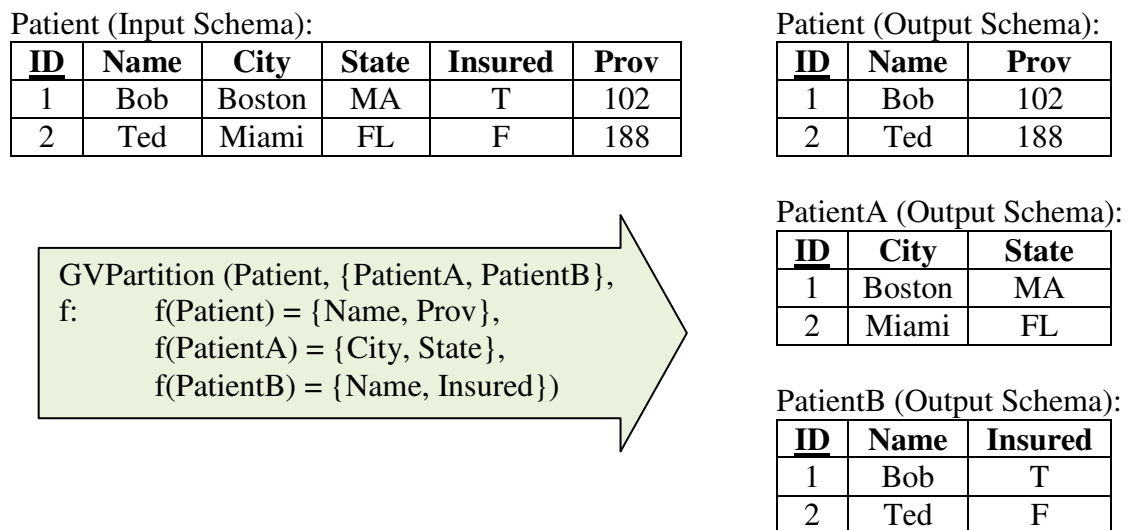
Figure 4.3: An example of the GVPartition transformation acting on instances

- The Pivot and Unpivot transformations both operate on a single dimension of analysis. In other words, they operate with a single attribute column and a single value column. In data warehousing applications, one may want to pivot or unpivot on multiple attribute or value columns simultaneously.

- The HPartition and VPartition transformations both produce non-redundant output in the sense that no non-key data is stored in more than one place. However, one may intentionally want to store non-key data redundantly. For instance, one may want to vertically partition a table into three tables, where all three tables have some non-key columns in common (similar to having multiple, overlapping covering indexes) to speed up certain queries.

In Table 4.1, we define generalized versions of three transformations that were originally introduced in Chapter 3. PPartition (Predicate Partition, Table 4.2, and an example in Figure 4.1) is a version of HPartition, where rows are distributed amongst a collection of tables based on whether each row satisfies the predicate associated with the output table. With PPartition, each row can end up in multiple tables if the row satisfies more than one predicate. The MultiPivot transformation (Table 4.3, and an example in Figure 4.2) is a generalization of Pivot, where now multiple attribute and value columns are allowed. GVPartition (Table 4.4, and an example in Figure 4.3) is a version of VPartition that can produce any number of tables (rather that just two) and that allows columns to occur in multiple tables.

Table 4.1: Additional channel transformations, their descriptions, and their effect on relational queries.

Transformation	Description of Transformation	Effect on Query
PPartition $(T_a, \vec{Ts}, \vec{Cs}, \vec{Ps})$	Horizontally partition a table T_a into a collection of tables according to a collection of predicates $\vec{P}$s into tables $\vec{T}$s. $\vec{P}$s must satisfy $\lvert\vec{P}\rvert = \lvert\vec{T}s\rvert$ and $\bigvee_{p\in\vec{P}s} p = true$. If a tuple t satisfies the i'th predicate in $\vec{P}$s, it will appear in the i'th table in $\vec{T}$s. Thus, tuples may appear in more than one output table, but every tuple must appear in at least one output table. $\vec{C}s$: the columns that are examined by the predicates in $\vec{P}s$ p_i: the predicate in $\vec{P}s$ associated with $t \in \vec{T}s$ t_p: the table associated with predicate p	$T_a \Longrightarrow \cup_{t\in\vec{T}s} t$ $\sigma_c T_a(\exists_{p\in\vec{P}s} c \wedge p = c) \Longrightarrow \sigma_c t_p$
MultiPivot $(T_a, \vec{As}, \vec{Vs})$	Similar to Pivot, except the "Attribute" and the "Value" parameters may span multiple attributes. The pivoted column names are defined by the function f_{name} : $(\vec{V}s \times \Pi_{a\in\vec{As}} \textbf{Dom}(a)) \rightarrow String$ that concatenates the arguments into a string joined by underscores. For instance, if the table $T_a(\underline{K, A_1, A_2, V_1, V_2})$ contains a tuple $(1, X, Y, H, T)$, the value H will be stored in the pivoted table in a column called $V_1\_X\_Y$. Function f_{name}^{-1} takes a pivoted table column and produces values for $\vec{A}s$ and one of the values from $\vec{V}$. For instance, $f_{name}^{-1}(V_1\_X\_Y) = (V_1, X, Y)$. V_C refers to the value in $\vec{V}$ that contributes to the name of column C. For instance, for column $C = V_1\_X\_Y$, $V_C = V_1$. One can evaluate V_C by evaluating $\pi_1 f_{name}^{-1}(C)$.	$T_a \Longrightarrow$ $\vec{\bowtie}_{\vec{V}s, A, V}\, \alpha_{\{A\}, \{A\}\cup\vec{As}, f_{name}^{-1}} \vec{\bowtie}_{\textbf{Cols}_{out}(T_a)-\textbf{Keys}(T_a), A, V} T_a$
GVPartition $(T_a, \vec{T}_n, f)$	Vertically partition a table T_a into any number of tables $\vec{T}_n$ and T_a. Keys are placed in all of the tables. Place each non-key column C in each table $t \in \vec{T}_n$ if $C \in f(t)$ where f is defined as follows: $f : \vec{T}_n \cup \{T_a\} \rightarrow 2^{\textbf{Cols}(T_a)}$ where $\forall_{t\in\vec{T}_n} f(t) \neq \emptyset$. All columns that are not accounted for in the output of f are placed into T_a. Define $coverage(f) = \cup_{t\in\vec{T}_n\cup\{T_a\}} f(t)$.	$T_a \Longrightarrow T_a(\vec{\bowtie}_{t\in\vec{T}_n} \bowtie t)$ where all joins are natural joins. Optimized: $\pi_{\vec{C}} T_a \Longrightarrow \pi_{\vec{C}} t$ if $\exists_{t\in\vec{T}_n}(\vec{C} - \textbf{Keys}(T_a)) \subseteq f(t)$

Just like the original seven transformations, these three generalized versions have a predictable effect on physical database characteristics. Table 4.5 presents the effect of the generalized transformations on table statistics.

The tradeoff between a transformation from Chapter 3 and its generalized counterpart is between simplicity and speed on the one hand and expressiveness on the other. The generalized transformations in Table 4.1 can provide a mapping between schemas that cannot be expressed using only the transformations in Table 3.2. However, the generalized transformations require significantly more input to be specified by the database developer; for instance, HPartition only required the developer to specify a table name and a column name, while its generalized version PPartition requires the developer to specify (potentially complicated) predicates along with the desired output table names. In addition, the generalized transformations have necessarily more complicated algorithms, requiring more in-memory processing, and may yield more complicated output.

For instance, compare the processing of DML delete statements between Pivot and MultiPivot in their respective tables. Delete processing by a Pivot results in either a single update statement or a single delete statement, depending on whether a condition on the attribute column exists. Delete processing by a MultiPivot may result in many update statements. Also, consider the query processing of the two transformations; Pivot introduces a single, $O(n \log n)$, operator, while MultiPivot introduces two $O(n \log n)$ operators and a function application.

Table 4.2: Defining the action of PPartition.

Statement	PPartition $(T_a, \vec{T's}, \vec{Cs}, f)$
Insert $I(T, \vec{C}, Q)$	$T = T_a \implies \forall_{t \in \vec{T's}} I(t, \vec{C}, \sigma_{p_t} Q)$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \wedge (\vec{C} \cap \vec{Cs} = \varnothing) \implies \forall_{t \in \vec{T's}} (\vec{F} \wedge p_t \neq false) \rightarrow U(t, \vec{F}, \vec{C}, Q)$ $T = T_a \wedge (\vec{C} \cap \vec{Cs} \neq \varnothing) \implies \forall_{t \in \vec{T's}} ((\vec{F} \wedge p_t \neq false) \rightarrow U(t, \vec{F}, \vec{C}, Q),$ $Loop(c, (\cup_{t' \in \vec{T's}-\{t\}} \sigma_{\vec{F} \wedge p_t} t') - t, I(t, \textbf{Cols}(t), c)),$ $Loop(c, \pi_{\textbf{Keys}(t)} \sigma_{\neg p_t} t, D(t, < \textbf{Keys}(t), c >)))$
Delete $D(T, \vec{F})$	$T = T_a \implies \forall_{t \in \vec{T's}} ((\vec{F} \wedge p_t) \neq false) \rightarrow D(t, \vec{F})$
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_a \implies \forall_{t \in \vec{T's}} AT(t, \vec{C}, \vec{D}, \vec{K})$
Rename Table $RT(T_o, T_n)$	Drop statement
Drop Table $DT(T)$	$T = T_a \implies \forall_{t \in \vec{T's}} DT(t)$
Add Column $AC(T, C, D)$	$T = T_a \implies \forall_{t \in \vec{T's}} AC(t, C, D)$
Rename Column $RC(T, C_o, C_n)$	$T = T_a \implies \forall_{t \in \vec{T's}} RC(t, C_o, C_n)$
Drop Column $DC(T, C)$	$T = T_a \wedge C \in \vec{Cs} \implies$ Throw error (cannot drop a column involved in predicates without changing function) $T = T_a \wedge C \notin \vec{Cs} \implies \forall_{t \in \vec{T's}} DC(t, C)$
Add Element $AE(T, C, E)$	$T = T_a \wedge C \in \vec{Cs} \implies$ Throw error (cannot alter a column involved in predicates without changing function) $T = T_a \implies \forall_{t \in \vec{T's}} AE(t, C, E)$
Rename Element $RE(T, C, E_o, E_n)$	$T = T_a \wedge C \in \vec{Cs} \implies$ Throw error (cannot alter a column involved in predicates without changing function) $T = T_a \implies \forall_{t \in \vec{T's}} RE(t, C, E_o, E_n)$
Drop Element $DE(T, C, E)$	$T = T_a \wedge C \in \vec{Cs} \implies$ Throw error (cannot alter a column involved in predicates without changing function) $T = T_a \implies \forall_{t \in \vec{T's}} DE(t, C, E)$
Foreign Key $FK(\vec{F}|T.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$	$T = T_a \implies \forall_{t \in \vec{T's}} ((\vec{F} \wedge p_t) \neq false) \rightarrow FK(\vec{F}|t.\vec{X} \rightarrow \vec{G}|B.\vec{Y})$ $B = T_a \implies \forall_{t \in \vec{T's}} ((\vec{F} \wedge p_t) \neq false) \rightarrow FK(\vec{F}|T.\vec{X} \rightarrow \vec{G}|t.\vec{Y})$

Table 4.3: Defining the action of MultiPivot. Some DDL statements are unaffected.

Statement	MultiPivot $(T_a, \vec{As}, \vec{Vs})$
Insert $I(T, \vec{C}, Q)$	$T = T_a \implies Error((\pi_{\textbf{Keys}_{\textbf{in}}(T_a)}Q \bowtie T_a) \cap \pi_{\textbf{Keys}_{\textbf{in}}(T_a)}\alpha_{\{A\},\{A\}\cup\vec{As},f_{name}^{-1}}$ $\boxtimes_{\textbf{Cols}_{\textbf{out}}(T_a)-\textbf{Keys}_{\textbf{out}}(T_a),A,V}(\pi_{\textbf{Cols}_{\textbf{out}}(T_a)}(Q \bowtie T_a))),$ $\forall_{t\in\Pi_{a\in\vec{As}}\textbf{Dom}(a)}Loop(t, \sigma_{\vec{As}=t}Q \bowtie (\pi_{\textbf{Keys}_{\textbf{out}}(T_a)}T_a),$ $U(T_a, < \textbf{Keys}_{\textbf{out}}(T_a), \pi_{\textbf{Keys}_{\textbf{out}}(T_a)}t >, \forall_{c\in\vec{C}}(f_{name}(c \times t)), \pi_{\vec{C}}t)), I(T_a, \textbf{Cols}_{\textbf{out}}(T_a),$ $\boxtimes_{\textbf{Cols}_{\textbf{out}}(T_a)-\textbf{Keys}_{\textbf{out}}(T_a),A,V}\alpha_{\{A\}\cup\vec{As},\{A\},f_{name}}\boxtimes_{\vec{C}-\textbf{Keys}_{\textbf{out}}(T_a),A,V}(Q\overline{\bowtie}(\pi_{\textbf{Keys}(T_a)-\{A\}}T_a)))$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \implies \forall_{t\in A_{space}}U(T_a, \vec{F} - \vec{F_0}, \forall_{c\in\vec{C}}(f_{name}(c \times t)), Q)$ where $A_{space} = \Pi_{a\in\vec{As}}\begin{pmatrix} \{v\} & \text{if } \exists_v < a, v >\in \vec{F} \\ \textbf{Dom}(a) & \text{else} \end{pmatrix}$ and $\vec{F_0}$ is the set of all conditions that reference columns in $\vec{As}$
Delete $D(T, \vec{F})$	$T = T_a \land \exists_{<c,v>\in\vec{F}}c \in \vec{As} \implies \forall_{t\in A_{space}}U(T_a, \vec{F} - \vec{F_0}, \forall_{c\in\vec{C}}(f_{name}(c \times t)), (null)_{pad})$ $T = T_a \land \nexists_{<c,v>\in\vec{F}}c \in \vec{As} \implies D(T_a, \vec{F})$
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_a \implies AT(T_a, (\vec{C} - (\vec{As} \cup \vec{Vs})) \cup \{\forall_{t\in(\vec{Vs}\times\Pi_{a\in\vec{As}}\textbf{Dom}(a))}f_{name}(t)\},$ $\vec{D} - (\forall_{a\in\vec{As}}\textbf{Dom}(a) \cup \forall_{v\in\vec{Vs}}\textbf{Dom}(v)) \cup \{\forall_{t\in(\vec{Vs}\times\Pi_{a\in\vec{As}}\textbf{Dom}(a))}\textbf{Dom}(\pi_1 t)\}, \vec{K} - \vec{As})$
Add Column $AC(T, C, D)$	$T = T_a \implies \forall_{t\in\Pi_{a\in\vec{As}}\textbf{Dom}(a)}AC(T_a, f_{name}(C \times t), D)$
Rename Column $RC(T, C_o, C_n)$	$T = T_a \implies \forall_{t\in\Pi_{a\in\vec{As}}\textbf{Dom}(a)}RC(T_a, f_{name}(C_o \times t), f_{name}(C_n \times t))$
Drop Column $DC(T, C)$	$T = T_a \implies \forall_{t\in\Pi_{a\in\vec{As}}\textbf{Dom}(a)}DC(T_a, f_{name}(C \times t))$
Add Element $AE(T, C, E)$	$T = T_a \land C \in \vec{As} \implies \forall_{v\in\vec{Vs}}\forall_{t\in\Pi_{a\in\vec{As}-\{C\}}\textbf{Dom}(a))}AC(T_a, f_{name}(v \times g_C(E,t)), \textbf{Dom}(v))$ where $g_C(E, t)$ is the function that places E in the C position in tuple t $T = T_a \land C \in \vec{Vs} \implies \forall_{c\in\textbf{Cols}_{\textbf{in}}}(c_v = C) \rightarrow AE(T, c, E)$
Rename Element $RE(T, C, E_o, E_n)$	$T = T_a \land C \in \vec{As} \implies$ $\forall_{v\in\vec{Vs}}\forall_{t\in\Pi_{a\in\vec{As}-\{C\}}\textbf{Dom}(a))}RC(T_a, f_{name}(v \times g_C(E_o,t)), f(v \times g(E_n,t)), E_n)$ $T = T_a \land C \in \vec{Vs} \implies \forall_{c\in\textbf{Cols}_{\textbf{in}}}(c_v = C) \rightarrow RE(T, c, E_o, E_n)$
Drop Element $DE(T, C, E)$	$T = T_a \land C \in \vec{As} \implies \forall_{v\in\vec{Vs}}\forall_{t\in\Pi_{a\in\vec{As}-\{C\}}\textbf{Dom}(a))}DC(T_a, f_{name}(v \times g_C(E,t)))$ $T = T_a \land C \in \vec{Vs} \implies \forall_{c\in\textbf{Cols}_{\textbf{in}}}(c_v = C) \rightarrow DE(T, c, E)$
Foreign Key $FK(\vec{F}\|T.\vec{X}$ $\rightarrow \vec{G}\|B.\vec{Y})$	$(T = T_a \land \vec{As} \cap \vec{X} \neq \varnothing) \lor (B = T_a \land \vec{As} \cap \vec{Y} \neq \varnothing) \implies \text{Treat as Tier 3}$ $T = T_a \land \exists_{<c,v>\in\vec{F}}c \in \vec{As} \land \vec{Vs} \cap \vec{X} = \varnothing \land \vec{As} \cap \vec{X} = \varnothing \implies \text{Treat as Tier 3}$ $T = T_a \land \vec{Vs} \cap \vec{X} \neq \varnothing \land \vec{As} \cap \vec{X} = \varnothing \implies$ $\forall_{t\in A_{space}}(FK(\vec{F}\|T.(\vec{X} - \vec{Vs} \cup \forall_{c\in\vec{X}\cap\vec{Vs}}f_{name}(c \times t) \rightarrow \vec{G}\|B.\vec{Y})))$

Table 4.4: Defining the action of GVPartition.

Statement	GVPartition $(T_a, \vec{T_n}, f)$
Insert $I(T, \vec{C}, Q)$	$T = T_a \implies \forall_{t \in \vec{T_n}} (\vec{C} \cap f(t) \neq \varnothing) \to I(t, (\mathbf{Keys}(T_a) \cup (\vec{C} \cap f(t))), \pi_{(\mathbf{Keys}(T_a) \cup (\vec{C} \cap f(t)))} Q),$ $I(T_a, (\mathbf{Keys}(T_a) \cup (\vec{C} - f(\vec{T_n})) \cup f(T_a)), \pi_{(\mathbf{Keys}(T_a) \cup (\vec{C} - f(\vec{T_n})) \cup f(T_a))} Q)$
Update $U(T, \vec{F}, \vec{C}, Q)$	$T = T_a \implies U(T_a, \vec{F}, (\vec{C} - f(\vec{T_n}) \cup f(T_a), \pi_{(\vec{C} - f(\vec{T_n})) \cup f(T_a)}))Q,$ $\forall_{t \in T_n \wedge} (\vec{C} \cap f(t) \neq \varnothing) \to I(t, \mathbf{Keys}(T_a), \pi_{\mathbf{Keys}(T_a)} \sigma_{\vec{F}} T_a - \pi_{\mathbf{Keys}(T_a)} t),$ $\forall_{t \in T_n \wedge} (\vec{C} \cap f(t) \neq \varnothing) \to U(T_n, \vec{F}, \vec{C} - \vec{C's}, \pi_{\vec{C} - \vec{C's}} Q)$
Delete	$T = T_a \implies D(T_a, \vec{F}), \forall_{t \in \vec{T_n}} D(t, \vec{F})$
Add Table $AT(T, \vec{C}, \vec{D}, \vec{K})$	$T = T_a \implies \forall_{t \in \vec{T_n}} AT(t, \vec{C} \cap (f(t) \cup \vec{K}), \{d \in \vec{D} | col(d) \in f(t) \cup \vec{K}\}, \vec{K}),$ $AT(T_a, (\vec{C} - coverage(f)) \cup \vec{K} \cup f(T_a), \{d \in \vec{D} | col(d) \in (\vec{C} - coverage(f))$ $\cup \vec{K} \cup f(T_a)\}, \vec{K}), \forall_{t \in \vec{T_n}} FK(true | t.\vec{K} \to true | T_a.\vec{K})$
Rename Table $RT(T_o, T_n)$	$T_n \in \vec{T_n} \implies$ Throw error (naming conflict)
Drop Table $DT(T)$	$T = T_a \implies \forall_{t \in \vec{T_n} \cup \{T_a\}} DT(t)$
Add Column $AC(T, C, D)$	$T = T_a \wedge C \notin coverage(f)) \implies AC(T_a, C, D)$ $T = T_a \wedge C \in coverage(f)) \implies \forall_{t \in \vec{T_n}} (\vec{C} \cap f(t) \neq \varnothing) \to AC(t, C, D)$
Rename Col. $RC(T, C_o, C_n)$	$T = T_a \wedge C_o \in \mathbf{Keys}(T_a) \implies \forall_{t \in \vec{T_n} \cup \{T_a\}} RC(t, C_o, C_n)$ $T = T_a \wedge C_o \notin coverage(f) \wedge C_o \notin \mathbf{Keys}(T_a) \implies RC(T_a, C_o, C_n)$ $T = T_a \wedge C_o \in coverage(f) \implies \forall_{t \in \vec{T_n}} C_o \in f(t) \to RC(t, C_o, C_n)$
Drop Column $DC(T, C, D)$	$T = T_a \wedge C \notin coverage(f) \implies DC(T_a, C)$ $T = T_a \wedge C \in coverage(f) \implies \forall_{t \in \vec{T_n}} C \in f(t) \to DC(t, C)$
Add Element $AE(T, C, E)$	$T = T_a \wedge C \in \mathbf{Keys}(T_a) \implies \forall_{t \in \vec{T_n} \cup \{T_a\}} AE(t, C, E)$ $T = T_a \wedge C \notin coverage(f) \wedge C \notin \mathbf{Keys}(T_a) \implies AE(T_a, C, E)$ $T = T_a \wedge C \in coverage(f) \wedge C \notin \mathbf{Keys}(T_a) \implies \forall_{t \in \vec{T_n} \cup \{T_a\}} C \in f(t) \to AE(t, C, E)$
Rename Elt. RE (T, C, E_o, E_n)	$T = T_a \wedge C \in \mathbf{Keys}(T_a) \implies \forall_{t \in \vec{T_n} \cup \{T_a\}} RE(t, C, E_o, E_n)$ $T = T_a \wedge C \notin coverage(f) \wedge C \notin \mathbf{Keys}(T_a) \implies RE(T_a, C, E_o, E_n)$ $T = T_a \wedge C \in coverage(f) \wedge C \notin \mathbf{Keys}(T_a) \implies \forall_{t \in \vec{T_n} \cup \{T_a\}} C \in f(t) \to RE(t, C, E_o, E_n)$
Drop Element $DE(T, C, E)$	$T = T_a \wedge C \in \mathbf{Keys}(T_a) \implies \forall_{t \in \vec{T_n} \cup \{T_a\}} DE(t, C, E)$ $T = T_a \wedge C \notin coverage(f) \wedge C \notin \mathbf{Keys}(T_a) \implies DE(T_a, C, E)$ $T = T_a \wedge C \in coverage(f) \wedge C \notin \mathbf{Keys}(T_a) \implies \forall_{t \in \vec{T_n} \cup \{T_a\}} C \in f(t) \to DE(t, C, E)$
Foreign Key $FK(\vec{F} | T.\vec{X} \to \vec{G} | B.\vec{Y})$	$T = T_a \wedge \exists_{t \in \vec{T_n}} ((\vec{X} - \mathbf{Keys}(T_a)) \subseteq f(t)) \implies$ $\forall_{t \in \vec{T_n}} ((\vec{X} - \mathbf{Keys}(T_a)) \subseteq f(t)) \to FK(\vec{F} | t.\vec{X} \to \vec{G} | B.\vec{Y})$ $T = T_a \wedge \nexists_{t \in \vec{T_n}} ((\vec{X} - \mathbf{Keys}(T_a)) \subseteq f(t)) \implies Check(\pi_{\vec{X}} \sigma_{\vec{F}} T_a (\forall_{t \in \vec{T_n}} \bowtie t) \subseteq \pi_{\vec{Y}} \sigma_{\vec{G}} B)$

Table 4.5: Defining the action of augmented transformation operator on table statistics.

Transformation	Effect of Transformation
PPartition $(T_a, \vec{T}s, \vec{C}s, \vec{P}s)$	$TStat(T_a, R, \vec{H}) \Longrightarrow \forall_{t \in \vec{T}s} TStat(t, R_t, \vec{H}^t)$ where: $R_t = R \times \Sigma_{v \in V_{p_t}} \Pi_{b \in \vec{C}s}(H_b(v_b)/R)$ H_c^t is the function defined by $H_c^t(x) = H_c(x) \times \Sigma_{v \in V_{p_t}} \Pi_{b \in \vec{C}s}(H_b(v_b)/R)$ V_p is the set of all values $v = (v_1, v_2, \ldots, v_{|\vec{C}s|})$ in $\Pi_{c \in \vec{C}s} domain(c)$ such that $p(v) = true$. (Assume that there is no correlation between columns)
MultiPivot $(T_a, \vec{A}s, \vec{V}s)$	$TStat(T_a, R, \vec{H}) \Longrightarrow TStat(T_a, R_p, \vec{H}^p)$, where: $R_p = \Pi_{c \in \mathbf{Keys_{in}}(T_a) - \vec{A}s}(|activedom(c)|)$, where $activedom(c)$ is the active domain of column c, or the number of distinct values x that have non-zero values for $H_c(x)$, $H_c^p = H_c$ for any column $c \in \mathbf{Keys_{out}}(T_a)$ H_c^p for any column $c \in \mathbf{Cols_{out}}(T_a) - \mathbf{Keys_{out}}(T_a)$ is the function defined by $H_c^p(x) = \begin{cases} R_p \times (1 - P_{att}(c)) & \text{if } x = null \\ H_{V_c}(x) \times P_{att}(c)/R & \text{else} \end{cases}$ $P_{att}(c) = \Pi_{a \in \vec{A}s}(H_a(\pi_a(f_{name}^{-1}(c)))/|\mathbf{Dom}(a)|)$ (Assume that the values in columns $\vec{V}s$ are evenly distributed among pivot columns, and that columns $\mathbf{Keys_{in}}(T_a)$ are uncorrelated)
GVPartition $(T_a, \vec{C}s, T_n)$	$TStat(T_a, R, \vec{H}) \Longrightarrow TStat(T_a, R_a, \vec{H}^a), \forall_{t \in \vec{T}_n} TStat(T_t, R_t, \vec{H}^t)$, where: $R_a = R, R_t = R_a - K_t, H_C^a = H_C$ for all $C \in \vec{C}s \cup \mathbf{Keys}(T_a)$, $H_C^t = (f_C^t \circ H_C)$ for all $C \in \mathbf{Cols}(T_a) - (\vec{C}s \cup \mathbf{Keys}(T_a))$ f_C^t is defined by $f_C^t(x) = \begin{cases} x - K_t & \text{if } x = null \text{ and C is non-key} \\ x & \text{if } x \neq null \text{ and C is non-key} \\ x \times ((R - K_t)/R) & \text{else} \end{cases}$ $K_t = \lfloor R_a * \Pi_{c \in f(t)}(H_c(null)/R_a) \rfloor$ (K_t is an estimate of number of all-null rows in T_t, assuming random distribution of nulls) $K_a = \lfloor R_a * \Pi_{c \in f(T_a) \cup (\mathbf{Cols}(T_a) - coverage(f))}(H_c(null)/R_a) \rfloor$

4.2 APPLICATION-SPECIFIC TRANSFORMATIONS

So far, all of the transformations we have described have had a particular motivation in mind: physical database design. Either there is an existing physical database schema to which the database developer maps, or there is no physical storage layer yet, in which case the developer uses channel transformations to craft their desired physical database. Thus, all of the transformations that we have considered so far have been information preserving. In this section, we consider transformations that have a different purpose. Namely, we define transformations that correspond to business logic decisions. We are interested in particular in transformations that perform interesting non-invertible transformations on statements generated by the user interface, and transformations that introduce new data in the business logic. As mentioned in Chapter 1, the channel artifact is part of application middleware, and there is often business logic included in middleware.

Consider the CORI application. The case study in Chapter 3 revealed that, in addition to the physical transformations that occur in the CORI middleware, a few additional changes occur. We can categorize some of these changes as instances of the following high-level transformations:

- Adorn: Add environment information to tuples, such as current time, user, or machine (an example is shown in Figure 4.4(a))

- Lookup: Take the value located in a column or set of columns, look it up in a table, and replace the value with its corresponding key value in the lookup table (an example is shown in Figure 4.4(b))

- Audit: Change tuple processing so that tuples are never changed or deleted; rather,

tuples are assigned a lifespan interval, indicating the time during which the associated data is valid (an example is shown in Figure 4.4(c))

We refer to these transformations — and any other transformations that are motivated by business logic decisions — as *application-specific transformations*. Each of the application-specific transformation can be defined in the same way that we define a physical design transformation; that is, as a set of algorithms that transform Guava-supported statements (e.g., Insert, Update, Rename Table, or Error) into other Guava-supported statements. For instance, the Lookup transformation will take an Insert statement and produce as output another Insert statement, identical to the input except with the values in the lookup columns replaced by their corresponding key in the lookup table. In addition, application-specific transformations must satisfy the same information-preservation property; in Chapter 5, we provide proof that Audit information-preserving, for example. Lookup, on the other hand, replaces existing data with other data.

One difference between a physical-design transformation and an application-specific transformation is that physical transformations simply restructure data into a new form. Application-specific transformations, however, may add new data or modify data according to business rules. For instance, the Adorn and Audit transformations add new columns to a table; the values in these columns cannot be reconstructed from other data in the table, since they come from the environment at the time the transformation occurs.

Applications that access the input schema of a channel (i.e., the natural schema) may want to access the additional data introduced by the channel. For instance, if an Adorn transformation adds a new column "X" to a table, the application developer may choose to allow the query interface that accesses the natural schema to include a new field for

Patient (Input Schema):

ID	Name	City
1	Bob	Boston
2	Ted	Toronto

Adorn ⟶

Patient (Output Schema):

ID	Name	City	LastUpdated
1	Bob	Boston	20070121
2	Ted	Toronto	20070120

(a)

Data (Input Schema):

ID	Required	Data
1	T	N
2	F	93

Controls (Lookup Table):

ID	Form	Item
1	92	36
2	92	62

Lookup ⟶

Data (Output Schema):

Form	Item	Required	Data
92	36	T	N
92	62	F	93

(b)

Patient (Input Schema):

ID	Name	City
1	Bob	Boston
2	Ted	Toronto

Audit ⟶

Patient (Output Schema):

ID	Name	City	Insert	Delete
1	Bob	Boston	2006	null
2	Ted	Toronto	2006	null

Assume that data was created in 2006. Then, in 2007, delete the second row.

Patient (Input Schema):

ID	Name	City
1	Bob	Boston

Patient (Output Schema):

ID	Name	City	Insert	Delete
1	Bob	Boston	2006	null
2	Ted	Toronto	2006	2007

Now, update the existing row and add a new one.

Patient (Input Schema):

ID	Name	City
1	Bob	NYC
3	Sally	Toronto

Patient (Output Schema):

ID	Name	City	Insert	Delete
1	Bob	Boston	2006	2007
2	Ted	Toronto	2006	2007
1	Bob	NYC	2007	null
3	Sally	Toronto	2007	null

(c)

Figure 4.4: Examples of the Adorn, Lookup, and Audit transformations acting on concrete instances

"X" to allow the user to have access to that data when the user formulates queries. To allow application developers to be informed about the changes made by application-specific transformations in the channel, applications may issue an additional statement against the channel called a *change spike*:

Definition: A *change spike* $\wedge(T)$ for a table T is a request made from the application level and sent through the channel, to accumulate all changes that the channel makes to table T.

The application may generate change spikes whenever it deems necessary. A likely scenario for a user interface (or Guava query interface) is to generate a change spike when a form loads for the first time during each run of an application. The application or query interface can then choose to display the most complete and up-to-date information. For the transformations introduced in this section, the response to a spike $\wedge(T)$ will consist of a (possibly empty) list of *change items* of the form *NewC(C, D)*, which represents that a new column was added to T called C with domain D.

Figure 4.5 presents an overview of how the channel responds to a change spike. For simplicity, we assume that the channel is broken into two segments, called *visible* (which contains transformations that may add or change data) and *invisible* (which contains physical design operators). (The language of possible change items would be significantly more complex if, say, change items need to be pushed through complex transformations such as Unpivot.) Since the application-specific transformations are all in the visible segment, the spike proceeds forward to the division between the two segments, where the channel generates an empty list of change items and sends it back toward the application.

Application Layer   Channel/Middleware   Physical Layer

Visible   Invisible

| Application | Natural Schema | | S1 | S2 | S3 | S4 | | R1 | R2 | R3 | R4 | | Physical DB |

$\wedge(T)$

Change List

Spike is sent into visible segment of channel
Response begins empty, but accumulates as each
transformation adds its change items, if any
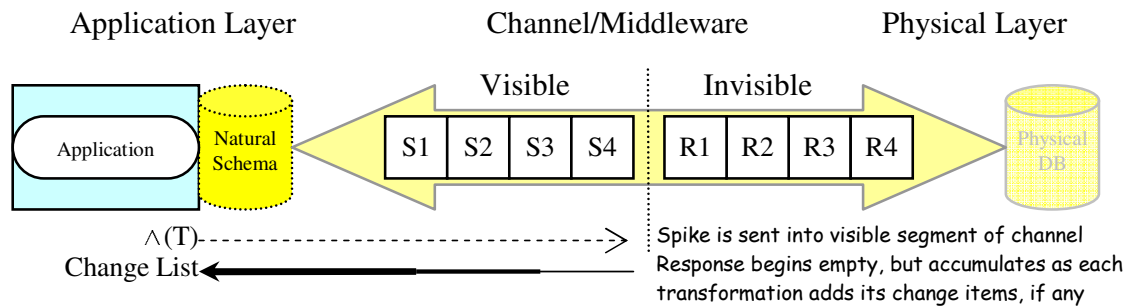
Figure 4.5: Processing a change spike in the channel

Each transformation in the visible segment then, in reverse order, examines whatever change items exist in the list and alters them to adjust the table and column references if necessary. We allow application-specific transformations to alter items that already exist in the change list because the change items are expressed in the transformation's output schema. Some alteration may be necessary to make sure that a change item is valid in the transformation's input schema (which is also the reason why change items are added and processed in the reverse direction of the channel). Then, the transformation produces its own list of changes $\Delta(T)$ that the transformation makes to the schema belonging to table T and adds them to the change list. Once each transformation has produced its list of changes in turn, the channel returns the complete change list to the application level. There, the query interface (or other application-level service) consumes the list and modifies its own appearance or behavior to reflect the changes.

Note that the changes contained in a change list do not affect the input schema of the channel, but only provides information to the querying application. In other words, if a change list comes back from the spike $\wedge(T_a)$ that includes a new column item $NewC(Time, DateTime)$, the table T_a in the natural schema is not changed to add the

new column *Time*. However, an application can issue queries that reference the new column by explicitly referencing the column, e.g., as part of a projection or selection. So, for instance, an application can issue the query $\pi_{Name,DOB,Time} T_a$ even though the column *Time* is not a valid column in T_a. Such column references are called *late-bound* because the column reference may bind to a column introduced at a point midway through the channel. Late-bound references that never bind to a column on their way through the channel will (correctly) throw an error when the query is eventually executed against the database; thus, applications are effectively restricted to posing queries that are valid against the channel's input schema with the possibility of late-bound references that correspond only to *NewC* entries in the change list.

For the rest of this section, we describe these three application-specific transformations (Adorn, Lookup, and Audit) one at a time. We then describe how a developer can write additional application-specific transformations when needed. The section concludes with a discussion of how application-specific transformations relate to the other properties of channels: namely, inverse relationships, commutativity with other transformations, and the effect of a transformation on physical properties. In Chapter 7 (future work), we describe in detail the process by which we envision developers creating their own transformations, either new versions of the three transformations presented here or entirely new ones.

4.2.1 Adorn

The Adorn transformation is the simplest of the transformations presented in this chapter. Its purpose is to add environment information to data on its way to the database.

If an Adorn transformation is configured to work on the `Person` table by adding the currently logged-in user "broberts" into a column called "LastUpdatingUser", then the statement:

```
INSERT INTO Person (ID, FName, LName)

VALUES (6, 'Bob', 'Thompson')
```

becomes

```
INSERT INTO Person (ID, FName, LName, CurrentUser)

VALUES (6, 'Bob', 'Thompson', 'broberts')
```

The values in the new columns are always set for new tuples by extending insert statements. The remaining issue is deciding whether to refresh the values of those columns during an update. We define the Adorn transformation to be given a list of columns to monitor. Whenever a monitored column is updated, the columns added by the Adorn transformation are updated to the current values from the environment. For example, if 'FName' is the only monitored column, then:

```
UPDATE Person SET FName = 'Ted' WHERE ID = 6
```

becomes

```
UPDATE Person SET FName = 'Ted', LastUpdatingUser = 'broberts'

WHERE ID = 6
```

whereas

```
UPDATE Person SET LName = 'Lassiter' WHERE ID = 6
```

remains unaltered.

Figure 4.6 shows a single form in a forms-based application. This form does not display any information about data that is added or altered by the channel. In Figure

Figure 4.6: Example form in an application without the effects of application-specific transformations (base case)

4.7(a), the form shows a new field, Last Updating User (among other changes), just beneath the original form, which is the result of using an Adorn operator in the channel. The effect of the Adorn transformation on an application or query interface is straightforward: For the current form, tell the application that additional fields are available and what their data types are. The application can then decide whether and how to display the information. In the figure, the added column of data (Last Updating User) appears as a text box because its domain is "string". The channel that yields the effects shown in Figure 4.7 is shown in Figure 4.8

Formalism: The formal declaration for Adorn is $Adorn(T, E, \vec{A}, \vec{M})$, for table T, some function E that accepts zero inputs and produces the values for the new columns, and a set of names $\vec{A}$ to give the new columns to hold the values that E produces. E must have the same number of outputs as names in $\vec{A}$. $\vec{M}$ is the set of columns the operator monitors for updates. Table 4.6 formalizes the action of the Adorn operator on Guava
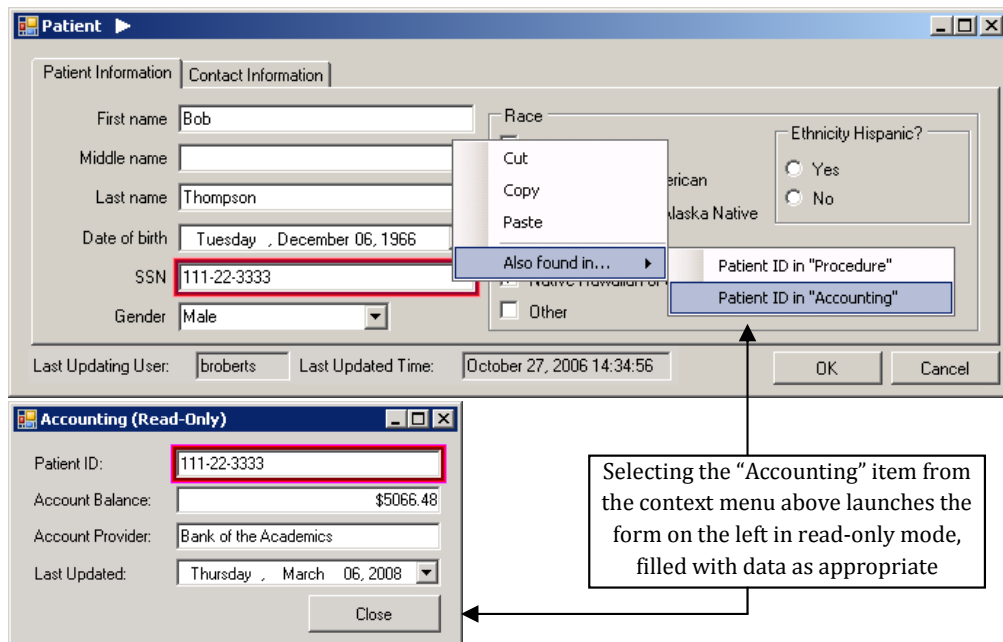
(a) Two forms from an application, but with visible changes from Adorn (Last Updating User), Audit (Last Updated Time), and Column Equate (context menu on SSN field)
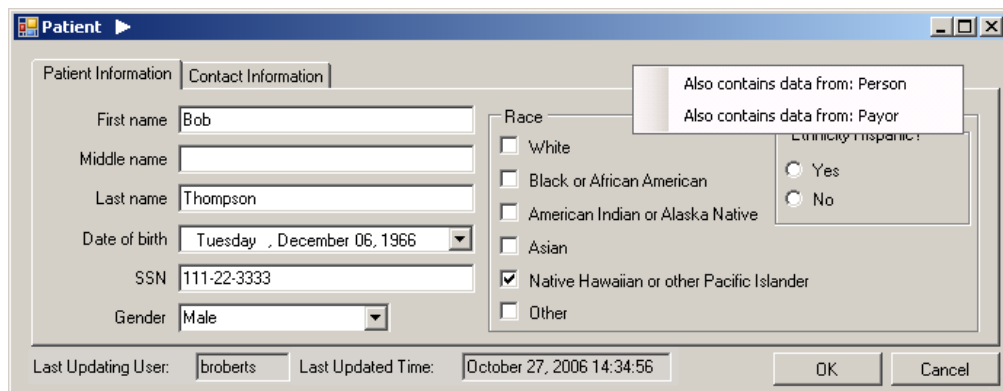
(b) The same form again, but with a context menu showing the effects of Table Equate

Figure 4.7: Examples of forms from an application, augmented with the effects of application-specific transformations and assertions; the Table Equate and Column Equate transformations are introduced in Section 4.3

| Adorn (Patient, CurrentUser, LastUpdatingUser, {All fields})) | ColumnEquate (Patient, SSN, Accounting, Patient ID) | ColumnEquate (Patient, SSN, Procedure, Patient ID) | TableEquate (Patient, Payor) | TableEquate (Person, Patient) | Audit(Person, LastUpdatedTime, EndTime, {}, {}) |

∧(Patient) - > ∧(Person) - - - - - - - - - - - - >

Change List

| | NewC(LastUpdatingUser, String) | RedC(SSN, Accounting, Patient ID) | RedC(SSN, Procedure, Patient ID) | RedT (Payor) | RedT (Person) | NewC(LastUpdatedTime, DateTime) |

Figure 4.8: The channel used by the forms in Figure 4.7, and how it responds to a change spike

Figure 4.9: An example query interface derived from the form in Figure 4.6

Table 4.6: Encapsulating the action of the Adorn transformation. Statements that are not listed are unaffected by the transformation

Statement	$\mathbf{Adorn}\,(T, E, \vec{A}, \vec{M})$
Query T_i^{query}	$T_i = T \land$ There are no references to columns $\vec{A}$ in any other operator in the current query $\implies \pi_{\vec{Cols_{in}(T)}} T^{query}$
Insert $I(T_i, \vec{C}, Q)$	$T_i = T \implies I(T_i, \vec{C} \cup \vec{A}, Q \times E_{now})$, where E_{now} is the value of E at transformation time.
Update $U(T_i, \vec{F}, \vec{C}, Q)$	$T_i = T \land \vec{C} \cap \vec{M} \neq \varnothing \implies U(T_i, \vec{F}, \vec{C} \cup \vec{A}, Q \times E_{now})$, where E_{now} is the value of E at transformation time.
Add Table $AT(T_i, \vec{C}, \vec{D}, \vec{K})$	$T_i = T \land \vec{C} \cap \vec{A} \neq \varnothing \implies$ Throw error (duplicate columns) $T_i = T \land \vec{C} \cap \vec{A} = \varnothing \implies AT(T_i, \vec{C} \cup \vec{A}, \vec{D} \cup \mathbf{Outputdomains}(E), \vec{K})$
Add Column $AC(T_i, C, D)$	$T_i = T \land C \in \vec{A} \implies$ Throw error (duplicate column)
Drop Column $DC(T_i, C)$	$T_i = T \land C \in \vec{M} \implies$ Throw error (cannot remove monitored column)
Change Spike $\land(T_i)$	$T_i = T \implies NewC(c, d)$ for each column $c \in \vec{C}$ and its corresponding domain d from the range of E

statements. In particular, note that Adorn responds to a change spike by reporting the columns that have been added with their data types.

In Table 4.6, we see that the Adorn operator adds the current value of the adorning function during insert (line 2) and update (for monitored columns, line 3), checks for duplicate column names for added columns (line 4), and responds to a change spike by describing the adorning elements (last line).

4.2.2 Lookup

At first glance, the Lookup transformation appears to be very much like the Apply transformation: For each row in a table, take the values in a column set $\vec{C_{in}}$ and perform an

invertible transformation that yields values for another column $\vec{C_{out}}$. Whereas the Apply transformation employs an invertible function, written by the developer, the Lookup transformation uses a table in the database. The lookup table must have two uniqueness constraints imposed on it, enforced by unique keys or triggers. The Lookup transformation leverages the uniqueness constraints to create an invertible mapping; if table T has uniqueness constraints on both column sets $\vec{C_1}$ and $\vec{C_2}$, then the query $\pi_{\vec{C_2}}\sigma_{\vec{C_1}=\vec{V}}T$ will produce a single $\vec{C_2}$ value vector for each value vector $\vec{V}$ for $\vec{C_1}$. Swapping $\vec{C_1}$ and $\vec{C_2}$ in the query also produces a single value vector for each input vector, and provides the reverse lookup mechanism.

Another difference between Lookup and Apply is, with a Lookup transformation, we know that the transformation is determined by a lookup table rather than an opaque function. Therefore, Lookup transformations can process Rename Element statements issued against the input of the Lookup, whereas Apply transformations would throw an error.

Our description of Lookup has one additional difference from Apply, motivated by the CORI case study. Our description of the Lookup transformation operates on key columns rather than non-key columns, which was the restriction on the Apply transformation. Table 4.7 provides a full description of our Lookup transformation.

Formalism: The Lookup transformation is declared as $Lookup(T, \vec{C_{in}}, \vec{C_{out}}, T^L, \vec{C_{in}^L},$ $\vec{C_{out}^L}, \vec{K}, \vec{D_{out}})$, where T is the name of the table whose column is being replaced, $\vec{C_{in}}$ are the input columns for the lookup, and $\vec{C_{out}}$ are the columns of the output. Table T^L is the name of the lookup table, which must be a table in the physical database; the implication is that T^L is not a table in the natural schema. Columns $\vec{C_{in}^L}$ and $\vec{C_{out}^L}$ are the

pair of columns in T^L with uniqueness constraints defined. Columns $\vec{C_{in}}$ and $\vec{C_{in}^L}$ must have the same domain and cardinality, as must $\vec{C_{out}}$ and $\vec{C_{out}^L}$. $\vec{D_{out}}$ is the list of domains of the output column of the lookup table (necessary because the transformation does not have awareness of the lookup table). The set $\vec{K}$ is the set of key columns of T^L (again, specified to give the operator sufficient information about the lookup table). K cannot overlap with any of the lookup columns.

4.2.3 Audit

The final — and most complex — application-specific transformation that we introduce in this section is Audit. The problems solved by Audit are similar to those that motivate temporal database research [71], with a similar solution: Give all tuples in an audited table two additional date attributes indicating a lifespan. Current data has a null ending time. Whenever data is updated or deleted by the application, the end time of the corresponding tuples is set to the current time and fresh tuples are added if necessary.

Rather than drop any columns from the physical database, the Audit transformation keeps track of a list of "deprecated" columns that are no longer in an audited table. The application may later re-add the dropped column; in that case, all of the data that was available in the dropped column would become visible in the newly-added column.

Table 4.7: Encapsulating the action of the Lookup transformation. Statements that are not listed are unaffected by the transformation (including updates, since we are looking at key columns only)

Statement	Lookup $(T, \vec{C}_{in}, C_{out}, T^L, \vec{C}^L_{in}, \vec{C}^L_{out}, \vec{K}, \vec{D}_{out})$
Query T_i^{query}	$T_i = T \Longrightarrow \rho_{\vec{C}^L_{in} \to \vec{C}_{in}} \pi_{Col_{in}(T) \cup \vec{C}^L_{in} - \vec{C}_{out}} T^{query} \bowtie_{\vec{C}_{out} = \vec{C}^L_{out}} T^{L,query}$
Insert $I(T_i, \vec{C}, Q)$	$T_i = T \Longrightarrow I(T_i, (\vec{C} - \vec{C}_{in}) \cup \vec{C}_{out}, \pi_{\vec{C}(T) \cup \vec{C}^L_{out} - \vec{C}} Q \bowtie_{\vec{C}_{in} = \vec{C}^L_{in}} T^L)$
Delete $D(T_i, \vec{F})$	$T_i = T \wedge \exists_{<c,v> \in \vec{F}} c \in \vec{C}_{in} \Longrightarrow Loop(t, \pi_{\vec{C}^L_{out}} \sigma_{F'} T^L, D(T_i, (\vec{F} - \vec{F'}) \cup \{< \vec{C}_{out}, t >\}))$, where $\vec{F'}$ is the subset of $\vec{F}$ that refer to columns in $\vec{C}_{in}$
Add Table $AT(T_i, \vec{C}, \vec{D}, \vec{K})$	$T_i = T \wedge \vec{C}_{in} \not\subseteq \vec{C} \Longrightarrow$ Throw error (inadequate columns) $\newline$ $T_i = T \wedge \vec{C}_{in} \subseteq \vec{C} \wedge \vec{C}_{in} \not\subseteq \vec{K} \Longrightarrow$ Throw error (inadequate columns) $\newline$ $T_i = T \wedge \vec{C}_{in} \subseteq \vec{C} \wedge \vec{C}_{in} \subseteq \vec{K} \Longrightarrow AT(T_i, (\vec{C} - \vec{C}_{in}) \cup \vec{C}_{out}, (\vec{D} - domains(\vec{C}_{in})) \cup \vec{D}_{out}, (\vec{K} - \vec{C}_{in}) \cup \vec{C}_{out})$
Add Element $AE(T_i, C, E)$	$T_i = T \wedge C \in \vec{C}_{in} \Longrightarrow Error(\{(E)\} - \pi_{C^L} T^L)$ where C^L is the corresponding column in the lookup table to C (verify that the element already lives in the lookup table)
Rename Element $RE(T_i, C, E_n, E_o)$	$T_i = T \wedge C \in \vec{C}_{in} \Longrightarrow Loop(t, \pi_{\vec{K}} \sigma_{C^L = E_o} T^L, U(T^L, < \vec{K}, t >, \{C^L\}, \{E_n\}))$ where C^L is the corresponding column in the lookup table to C (update the lookup table with the change, leave output table alone)
Drop Element $DE(T_i, C, E)$	$T_i = T \wedge C \in \vec{C}_{in} \Longrightarrow Loop(t, \pi_{\vec{C}^L_{out}} \sigma_{C^L = E} T^L, D(T, < \vec{C}_{out}, t >))$ where C^L is the corresponding column in the lookup table to C (drop all matching tuples)
Foreign Key $FK(\vec{F}[T_i.\vec{X} \to \vec{G}].B.\vec{Y})$	$T_i = T \wedge (C_{in} \in \vec{X} \vee \exists_{<c,v> \in \vec{F}} c \in \vec{C}_{in}) \Longrightarrow$ Treat as $Check(Q_1 \subseteq Q_2)$ and process queries $\newline$ $B = T \wedge (C_{in} \in \vec{Y} \vee \exists_{<c,v> \in \vec{G}} c \in \vec{C}_{in}) \Longrightarrow$ Treat as $Check(Q_1 \subseteq Q_2)$ and process queries

Audit exposes a field of type DateTime in response to a change spike. This field acts as a reference to the beginning of the liveness interval for the tuple of interest. When displayed on a form in an application, this field will show the time and date when the tuple was last inserted or updated. However, this field acts as a test on the lifespan when used in a filtering condition (used in a σ operator). So, in a query interface, the user can ask questions like, "what was the data like at time $\mathcal{T}$ for this table?" Audit will also expose the list of deprecated columns in response to a spike.

Our implementation of the Audit transformation is meant to solve a specific issue: ensure that no data is ever deleted from the tables in a database. It is not intended to implement all of the capabilities of a natively temporal system [45], but only what is necessary to keep historical information. For instance, we do not coordinate between Audit transformations to perform any kind of temporal join in the query processing, such as a global "as of time X" query; our Audit transformation supports "as of" queries on a table-by-table basis. This particular implementation of Audit was based on our interaction with CORI; most of their queries of interest are over current data, and the rare temporal queries that they write tend to be of the form "Show me the following data items for all patients at time T", which do not require a temporal join.

The example form shown in Figure 4.7(a) contains a field called Last Updated Time, which corresponds to the new, temporal attribute added by the Audit operator. The query interface shown in Figure 4.9 contains the same field, as well as a field Insurance Carrier that appears in Audit's deprecated list, meaning it was once in the application but deleted.

Formalism: The Audit transformation is declared as $Audit(T, B, E, \vec{DC}, \vec{DD})$, where

Table 4.8: Encapsulating the action of the Audit transformation. Statements not listed in the table are unaffected by the transformation

Statement	Audit $(T, B, E, \vec{DC}, \vec{DD})$
Query T_i^{query}	$T_i = T \wedge$ There are no references to columns $T.B$ in any operator in the current query $\implies \pi_{Cols_{in}(T)}\sigma_{E=null}T^{query}$
$\sigma_{T_i.B=V}Q$	$T_i = T \implies \sigma_{B \leq V \wedge (E \geq V \vee E=null)}Q$
$\sigma_{T_i.B>V}Q$	$T_i = T \implies \sigma_{B>V \vee (B \leq V \wedge (E>V \vee E=null))}Q$ (Same for $\geq$)
$\sigma_{T_i.B<V}Q$	$T_i = T \implies \sigma_{E<V \vee (B<V \wedge (E \geq V \vee E=null))}Q$ (Same for $\leq$)
$\sigma_{T_i.B \neq V}Q$	$T_i = T \implies \sigma_{B>V \vee E<V}Q$
Insert $I(T_i, \vec{C}, Q)$	$T_i = T \implies I(T, \vec{C} \cup \{B, E\}, Q \times \{CurrentTime, null\})$
Update $U(T_i, \vec{F}, \vec{C}, Q)$	$T_i = T \implies$ $I(T, Cols_{in}(T_i) \cup \{B, E\}, (\pi_{Cols_{in}(T) \cup \{B\}}\sigma_{\vec{F}}T) \times \{CurrentTime\})$, followed by $U(T, \vec{F} \cup \{E = null\}, \vec{C} \cup \{B\}, Q \times \{CurrentTime\})$
Delete $D(T_i, \vec{F})$	$T_i = T \implies U(T_i, \vec{F} \cup \{E = null\}, \{E\}, \{CurrentTime\})$
Add Table $AT(T_i, \vec{C}, \vec{D}, \vec{K})$	$T_i = T \wedge \vec{C} \cap (\{B, E\} \cup \vec{DC}) \neq \varnothing \implies$ Throw error (duplicate columns) $T_i = T \wedge \vec{C} \cap (\{B, E\} \cup \vec{DC}) = \varnothing \implies AT(T_i, \vec{C} \cup \{B, E\} \cup \vec{DC}, \vec{D} \cup \{DateTime, DateTime\} \cup \vec{DD}, \vec{K} \cup \{B\})$
Add Column $AC(T_i, C, D)$	$T_i = T \wedge C \in \{B, E\} \implies$ Throw error (duplicate column) $T_i = T \wedge C \in \vec{DC} \implies$ Check if D matches the column's old domain in $\vec{DD}$. If so, drop statement and remove the column from $\vec{DC}$ and its domain from $\vec{DD}$. If not, throw error (domain conflict with deprecated column).
Rename Column $RC(T_i, C_o, C_n)$	$T_i = T \wedge C_n \in \{B, E\} \cup \vec{DC} \implies$ Throw error (duplicate column)
Drop Column $DC(T_i, C)$	$T_i = T \implies$ Add C to $\vec{DC}$, add $Domain(T.C)$ to $\vec{DD}$, and drop the statement
Change Spike $\wedge(T_i)$	$T_i = T \implies NewC(B, DateTime)$, followed by $NewC(c, d)$ for each $c, d \in \vec{DC}, \vec{DD}$

T is the name of the table to audit and *B* and *E* are the names of the begin and end times-tamp columns respectively. $\vec{DC}$ is a list of the columns that have been dropped from the table (but still exist in the database), and $\vec{DD}$ are the domains of those columns. From a user perspective, we expect that at the time a developer places an Audit transforma-tion in the channel, the list of deprecated columns is empty (unless the developer has some specific prior knowledge), and that the transformation will maintain the list of deprecated columns over the lifespan of the application.

For query processing, if column *B* is not referenced in the query in a select oper-ator σ, then the transformation assumes that the query will only retrieve current data. Therefore, query *Q* becomes query $\sigma_{E=null}Q$. Otherwise, the query treats the reference to *B* as a test on the lifespan of the query. For instance, the filter $\sigma_{B=V}$ becomes the filter $\sigma_{B \leq V \wedge (E \geq V \vee E=null)}$ — a test to see if the value *V* falls within a tuple's lifespan. Each σ operator is shown with a single condition $B\theta V$ to simplify the exposition in Table 4.8.

A reminder: This implementation of the Audit transformation is only one possibility. We describe our thoughts on implementing new application-specific transformations, including alternative implementations of Audit, in Chapter 7.

4.2.4 Application-Specific Transformations and Equivalences

Like the seven transformations in Chapter 3, application-specific transformations can participate in two classes of equivalence relationships: commutativity and invertibility. The trivial commutative case still holds, i.e., that two transformations can commute if they do not appear together in any table's trace. This property holds even between a physical design transformation and an application-specific transformation. The visible

segment assumption still applies, and is necessary so that we can keep the language of change items simple (consider the process of pushing a New Column change item through a Pivot or Unpivot, where in one case, the new column would become a new element after passing through a Pivot, and in the other case, the new column would invalidate the action of an Unpivot by having too many value columns).

There are also non-trivial commutativity equivalences involving application-specific transformations. Here are two of them:

Theorem 4.2.1 $[Adorn(T, E, \vec{A}, \vec{M}), Adorn(T, E', \vec{A'}, \vec{M'})] \equiv$

$\quad [Adorn(T, E', \vec{A'}, \vec{M'}), Adorn(T, E, \vec{A}, \vec{M})]$, provided $\vec{A} \cap \vec{M'} = \varnothing$ and $\vec{A'} \cap \vec{M} = \varnothing$

Theorem 4.2.2 $[Audit(T, B, E, \vec{DC}, \vec{DD}), HPartition(T, C)] \equiv$

$\quad [HPartition(T, C), \forall_{t \in domain(T.C)} Audit(t, B, E, \vec{DC}, \vec{DD})]$

The first theorem states that Adorn transformations can commute if the second Adorn transformation is not monitoring changes from the first Adorn. The second theorem states that Audit commutes through HPartition, resulting in all of the partitioned tables being audited.

One property that the application-specific transformations do not share with the physical design transformations is closure under inverses. Chapter 3 demonstrated that each of the seven physical design transformations has an inverse that can also be expressed as a physical design transformation (with the small exception of HMerge, for which this property is only true if the input schemas are union-compatible, which we address shortly). Said a different way, each of the seven physical design transformations can be "undone" by applying another transformation, which is an important property

when considering how to deal with evolving channels (a topic we consider in Chapter 6). This property is a product of the fact that physical design transformations are information-preserving and restructure data in an invertible way.

The same cannot be said about application-specific transformations in general. The Lookup transformation does have a natural inverse:

Theorem 4.2.3 $[Lookup(T, \vec{C_{in}}, \vec{C_{out}}, T^L, \vec{C_{in}^L}, \vec{C_{out}^L}, \vec{D_{out}}),$

$$Lookup(T, \vec{C_{out}}, \vec{C_{in}}, T^L, \vec{C_{out}^L}, \vec{C_{in}^L}, \vec{domains}(\vec{C_{in}}))] = \epsilon$$

To undo a Lookup, one just performs the reverse Lookup procedure. However, inverting an Adorn or Audit cannot be done in an information-preserving way. Because these transformations add data, their "inverse" transformation necessarily removes data; so far, we have not considered any transformation that removes data because it is by definition not information-preserving, and therefore cannot appear in a channel.

The inverse of $Adorn(T, E, \vec{A}, \vec{M})$ would be the transformation $DropProject(T, \vec{A})$. DropProject simply drops the referenced columns; clearly, this transformation is not information-preserving.

The inverse of $Audit(T, B, E, \vec{DC}, \vec{DD})$ is a pair of transformations, $[Filter(T, < E, null >), DropProject(T, \vec{DC} \cup \{B, E\})]$. The DropProject transformation we have seen used with the Adorn transformation; the Filter transformation is exactly the relational σ operator, in this case eliminating all rows where the value for the lifespan end-value E is not null.

The following set of transformations are not information-preserving, but are useful in defining inverses of transformations that are information-preserving:

- DropProject

- Filter

- Union

These transformations also have clear semantics, so for brevity, we omit their full definition in terms of transformation of statements. Adding these transformations to our language allows us to define inverses of transformations where we could not before, including the generalized operators from the previous section:

- The inverse of PPartition is Union (relying on duplicate elimination to remove redundant rows)

- The general inverse of HMerge (without assuming union compatibility) is HPartition, with DropProjects added to remove columns that were not present in a table originally

- The inverse of GVPartition is a sequence of VMerge transformations, preceded by DropProjects to remove redundant columns

Because these transformations are not information-preserving, we do not allow them to be placed in a channel by a developer. Therefore, defining the inverse of Adorn to be an instance of DropProject does not provide any benefit in terms of channel optimization, because these two transformations will never be present in the same channel. We will outline an alternative use for inverse transformations in Chapter 6, the chapter on schema evolution.

4.3 CORRESPONDENCE ASSERTIONS

The final class of new transformations we consider are called *correspondence assertions*. A correspondence assertion is a judgment made by the developer regarding relationships that exist in the extent of a schema. For instance, consider an application that includes two forms, one called "Student" and another called "Person". In the natural schema for this application, there will be a table for each of these two forms. However, the developer may decide that, for the purposes of the application and the database, students and people are interchangeable. In other words, every time a student is added, that student should appear as a person in the database, and vice versa.

This kind of relationship cannot be expressed in the application using any modern form-building library, and cannot be expressed in the relational model. It can, however, be expressed as a channel transformation. A channel transformation operates by re-writing statements with respect to a central purpose; in this case, we can create a transformation that accepts student changes and alters entries in Person accordingly. Or, the transformation can effectively make Student and Person both access the same underlying base table (which is how our version of the transformation is implemented).

What separates this class of transformation from the other transformations we have considered so far is how these transformations respect information preservation. With physical design or application-specific transformations, information preservation means that local updates against a table have a predictable effect on that table, and also no effect on any other table when viewed in the natural schema. With correspondence assertions, the same is true — but only with respect to the assertion being made. For instance, if a developer asserts that table T and table T' have the same extent, then an insert, update,

or delete against T will have an effect on the rows of T', but that effect is expected.

For the rest of this section, we consider two correspondence assertions: Column Equate and Table Equate. We describe what each assertion is meant to do, and formally define its effect as a channel transformation as we did with other transformations. That is, we show how these two assertions modify Guava statements. Like the application-specific transformations, correspondence assertions can have an effect on data that applications accessing the natural schema may be interested in; therefore, correspondence assertions will respond to a change spike by adding change items to the response. Thus, we assume that correspondence assertions occur in the visible segment of the channel. However, the language of change items returned in response to a spike expands to include two new possibilities:

- $RedC(C, C', T')$: Column C in spiked table T is redundant; identical data can be found in column C' in table T'.

- $RedT(T')$: Table T is redundant and T' holds objects in the same entity set.

4.3.1 Column Equate

We define the Column Equate transformation to eliminate redundant information. Specifically, if two different columns X and Y are equated using this transformation, then this transformation ensures that the value of X always propagates to Y, eliminating the need for Y to be present in the physical database. This situation may arise in an application where two different controls, perhaps on different forms, show the same data (e.g., a person's date of birth) and a change to the value of one must be reflected in the other. This results in redundant columns in the natural schema. The Column Equate transformation

Person (Input Schema):

ID	Name	DOB
1	Bob	11/27/75
3	Sally	08/15/64

Procedure (Input Schema):

PID	ID	ProcType	PtDOB
1	101	EGD	11/27/75
3	102	EGD	08/15/64
1	103	ERCP	11/27/75

Column Equate (Person, DOB, Procedure, PtDOB)

Person (Output Schema):

ID	Name	DOB
1	Bob	11/27/75
3	Sally	08/15/64

Procedure (Output Schema):

ID	PID	ProcType
101	1	EGD
102	3	EGD
103	1	ERCP

FK

FK
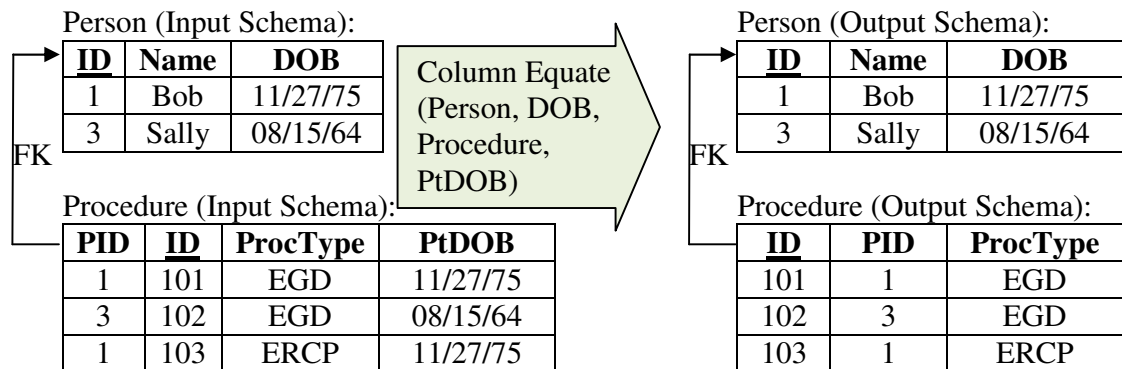
Figure 4.10: An example of the Column Equate transformation acting on a concrete instance

eliminates one of the columns and, thus, the redundancy. An example of Column Equate in action is shown in Figure 4.10.

One cannot equate pairs of columns arbitrarily. If one equates column X with column Y and they happen to be in the same table, or if they are in different tables that share a 1:1 relationship, there is no problem; for each row in the input schema holding a value for X, there is a unique row holding the Y value and vice versa. More generally, so long as each Y value can be mapped to a unique X value, Y can be dropped safely even if a single X value is associated with multiple rows in Y's table.

Any column that is dropped by a Column Equate transformation is not visible to any transformations that appear later on in the channel. For instance, if one equates column X with column Y, with column Y appearing second and therefore dropped, one cannot subsequently equate Y with column Z. One can instead equate column X with column Z, creating an implied relationship between Y and Z via transitivity.

Even though Column Equate transformations drop redundant columns, the columns

are still present in the natural schema and thus the applications. The user of an application or query interface may want to know if a widget on the screen is redundant, and if so, find all of the places in the application where the same data can be found. For example, a user may come across a read-only version of a field such as "date of birth" and want to find the form that contains an editable version.

The form in Figure 4.7(a) shows an example of a field that is redundant. The SSN field has a small halo around it, indicating that it can be found in other forms. Right-clicking the SSN control brings up a context menu that shows all of the other controls that have been equated with the SSN control. One can select any one of the equivalent controls from the menu to display the corresponding form in order to see the equivalent control in context. Note that the SSN data is not actually called SSN in the other tables; using information provided by Column Equate, the user now knows that SSN and Patient ID are used interchangeably in the application.

Formalism: Column Equate is declared as $ColumnEquate(T, X, T', Y)$, for tables T and T' with columns X and Y respectively. Column Y cannot be a key column, since it will get eliminated. Also, columns X and Y must have the same domain, and be *column-equatable*, meaning that they satisfy the following property:

Definition: Two columns X and Y in tables T and T', respectively, are *column-equatable* if T and T' are the same table or if there exists a unique path of 1:1 or 1:n relationships from T to T'. In this situation, column Y is considered *droppable* because it can be uniquely associated with a value for X along the relationship path.

Definition: Given two column-equatable columns X and Y in different tables T and T', the *join path* for X and Y (denoted $\mathcal{J}_{XY}$) is a relational algebra expression $T \bowtie_{cond_1}$

$T_1 \bowtie_{cond_2} T_2 \bowtie_{cond_3} \cdots \bowtie_{cond_n} T'$ that represents the unique way to join table T to table T' across relationships. Because we require there to be a unique join path for this particular flavor of Column Equate, the join path is inferred from the input schema of the transformation and need not be specified by the user. That we infer the join path rather than requiring the developer to provide one was an implementation decision.

To give an example of how the Column Equate formalism works, consider an insert statement $I(T_i, \vec{C}, Q)$. If the insert places values into a column Y that is being dropped, then we want to make sure that the value is propagated to its rightful place. So, after the rest of the insert statement (without Y) has been issued, the transformation creates a query Q' that joins the original query Q against the table that will hold the values in column Y along the join path. This query Q', in effect, takes each value in Y and finds the key of the row that will hold that value (we know that each row will exist because of the one-to-many relationship of the column-equatable property). Finally, we loop through the results of Q' and translate the key-value pairs into update statements to set the new values.

Inverse: An inverted $ColumnEquate(T_1, C_1, T_2, C_2)$ is $ColumnCopy(T_1, C_1, T_2, C_2)$, where $ColumnCopy$ is the transformation that takes the data in column C_1 from table T_1, follows the join path to T_2, and creates a column in T_2 called C_2 that holds a copy of the data from C_1. $ColumnCopy$ has a straightforward definition as a channel transformation, so we omit its definition for brevity. Note that the result of this inverse transformation is exactly the case where columns C_1 and C_2 hold identical data and may be considered redundant.

Table 4.9: Encapsulating the action of Column Equate.

Statement	ColumnEquate (T_1, C_1, T_2, C_2)
Query T_i^{query}	$T_i = T_2 \implies \pi_{Columns_{in}(T_2)-\{C_2\}\cup\{C_1\}}\mathcal{J}_{C_1 C_2}$
Insert $I(T_i, \vec{C}, Q)$	$T_i = T_2 \wedge C_2 \in \vec{C} \implies I(T_2, \vec{C}-\{C_2\}, \pi_{\vec{C}-\{C_2\}}Q),$ $Loop(t, \pi_{\mathbf{Keys}(T_1)\cup Q.C_2}(Q \bowtie_{\mathbf{Keys}(T_2)} \mathcal{J}_{C_1 C_2}), U(T_1, \mathbf{Keys}(T_1) = \pi_{\mathbf{Keys}(T_1)}t, \{C_1\}, \pi_{C_2}t)$
Update $U(T_i, \vec{F}, \vec{C}, Q)$	$T_i = T_2 \wedge C_2 \in \vec{C} \implies U(T_2, \vec{F}, \vec{C}-\{C_2\}, \pi_{\vec{C}-\{C_2\}}Q),$ $Loop(t, \pi_{\mathbf{Keys}(T_1)}(\sigma_{\vec{F}}\mathcal{J}_{C_1 C_2}), U(T_1, \mathbf{Keys}(T_1) = \pi_{\mathbf{Keys}(T_1)}t, \{C_1\}, \pi_{C_2}Q)$
Add Table $AT(T_i, \vec{C}, \vec{D}, \vec{K})$	$T_i = T_2 \wedge C_2 \in \vec{K} \implies$ Throw error (cannot have a redundant key column) $T_i = T_2 \wedge C_2 \notin \vec{C} \implies$ Throw error (redundant column must exist) $T_i = T_2 \wedge C_2 \in \vec{C} \wedge C_2 \notin \vec{K} \implies AT(T_2, \vec{C}-\{C_2\}, \vec{D}, \vec{K})$
Drop Column $DC(T_i, C)$	$T_i = T_2 \wedge C = C_2 \implies$ Throw error (cannot drop redundant column)
Change Spike $\wedge(T_i)$	$T_i = T_1 \implies RedC(C_1, C_2, T_2)$ $T_i = T_2 \implies RedC(C_2, C_1, T_1)$
Redundant Column $RedC(C, C', T')$	$T' = T_1 \wedge C' = C_1 \implies RedC(C, C', T'), RedC(C, C_2, T_2)$ (Transitivity)

4.3.2 Table Equate

Two tables in a relational schema Person and Patient may refer to the same entity set. However, the columns in the two tables may not be the same. This scenario corresponds to a situation where, for instance, two different forms in an application present data on People; however, one form focuses on demographics information, whereas the other form focuses on clinical background. The database designer may wish to simply store the data for both forms in the same table. In addition, a new person added via one form should be available for viewing and editing in the second form. We call the assertion that supports this transformation *Table Equate*. An example of Table Equate appears in Figure 4.11(a), and the consequence of inserting a new row into an equated table appears in Figure 4.11(b).

Table 4.10: Encapsulating the action of Table Equate. The symbol $T_{\neq}$ refers to whichever of T_1 or T_2 is not table T_i, and $Cols_{in}(T)$ refers to the list of columns of T before the transformation is applied

Statement	TableEquate (T_1, T_2)
Query T_i^{query}	$T_i = T_1 \implies \pi_{Cols_{in}(T_1)} T_1^{query}$ $T_i = T_2 \implies \pi_{Cols_{in}(T_2)} T_1^{query}$
Insert $I(T_i, \vec{C}, Q)$	$T_i = T_2 \implies I(T_1, \vec{C}, Q)$
Update $U(T_i, \vec{F}, \vec{C}, Q)$	$T_i = T_2 \implies U(T_1, \vec{F}, \vec{C}, Q)$
Delete $D(T_i, \vec{F})$	$T_i = T_2 \implies D(T_1, \vec{F})$
Add Column $AC(T_i, C, D)$	$T_i \in \{T_1, T_2\} \wedge C \notin Cols_{in}(T_{\neq}) \implies AC(T_1, C, D)$ $T_i \in \{T_1, T_2\} \wedge C \in Cols_{in}(T_{\neq}) \wedge D = Domain(T_{\neq}.C) \implies$ Drop statement (no effect) $T_i \in \{T_1, T_2\} \wedge C \in Cols_{in}(T_{\neq}) \wedge D \neq Domain(T_{\neq}.C) \implies$ Throw error (domain conflict)
Rename Column $RC(T_i, C_o, C_n)$	$T_i \in \{T_1, T_2\} \wedge C_o \notin Cols_{in}(T_{\neq}) \wedge C_n \notin Cols_{in}(T_{\neq}) \implies RC(T_1, C_o, C_n)$ $T_i \in \{T_1, T_2\} \wedge (C_o \in Cols_{in}(T_{\neq}) \vee C_n \in Cols_{in}(T_{\neq})) \implies$ Throw error (unexpected side effects)
Drop Column $DC(T_i, C)$	$T_i \in \{T_1, T_2\} \wedge C \notin Cols_{in}(T_{\neq}) \implies DC(T_1, C)$ $T_i \in \{T_1, T_2\} \wedge C \in Cols_{in}(T_{\neq}) \implies$ Drop statement (no effect)
Change Spike $\wedge(T_i)$	$T_i = T_1 \vee T_i = T_2 \implies$ Change to $\wedge(T_1)$ on its way through. On its way back to the application, add $RedT(T_{\neq})$, followed by $RedC(C_d, C_d, T_{\neq})$ for each column C_d common between T_1 and T_2
Redundant Column $RedC(C, C', T')$	$T' = T_1 \wedge C' \notin Cols_{in}(T_1) \wedge C' \in Cols_{in}(T_2) \implies RedC(C, C', T_2)$ $T' = T_1 \wedge C' \in Cols_{in}(T_1) \wedge C' \in Cols_{in}(T_2) \implies RedC(C, C', T_1)$, followed by $RedC(C, C', T_2)$
Redundant Table $RedT(T')$	$T' = T_1 \implies RedT(T_1)$, followed by $RedT(T_2)$

Person (Input Schema):

ID	Name	City
1	Bob	NYC
3	Sally	Toronto

Patient (Input Schema):

ID	Name	Provider#
1	Bob	53
3	Sally	19

Table Equate (Person, Patient)

Person (Output Schema):

ID	Name	City	Provider#
1	Bob	NYC	53
3	Sally	Toronto	19

(a)

Person (Input Schema):

ID	Name	City
1	Bob	NYC
3	Sally	Toronto
4	Paul	Sydney

Effect of Insert

Patient (Input Schema):

ID	Name	Provider#
1	Bob	53
3	Sally	19
4	Paul	NULL

(b)

Figure 4.11: An example of the Table Equate transformation acting on a concrete instance (a), and the consequences of inserting a new row into one of the equated tables (b)

Similarly to Column Equate, if tables T_1 and T_2 are considered to represent the same entities, there is no need to keep both tables; the columns from both tables are collected into a single table T_1. Our implementation assumes that if the same column name appears in both tables, then the two columns hold the same data, in effect creating an implicit Column Equate between them. Also, as with Column Equate, one cannot simply equate any two tables. The criteria for which pairs of tables one may equate are listed below in the formalism section.

To any form in the application layer, the Table Equate transformation exposes the list of other forms that hold data about the same entity. In the example above, if the

user is viewing the form called Patient, the application has the option to display a link to the other forms that refer to the same entity set. In Figure 4.7(b), the form gives a context menu when the user clicks anywhere on the form where there are no controls. The resulting menu displays a list all of the names of other forms that show the same data; clicking on any of the items in the menu brings up an instance of the other form so the user can view the other fields that are available and see what data is in those fields for the current entity. For the example in Figure 4.7(b), the Patient form holds entities that are also present in the Person form and the Payor form.

Formalism: The formal declaration for Table Equate is $TableEquate(T_1, T_2)$, for tables T_1 and T_2. The two tables must be *table-equatable*, which means that the following conditions hold:

- The number of primary key columns is the same for each table.

- Corresponding key columns of the two tables have the same names and domains.

- If column C exists in both tables and also participates in a foreign key in both tables, the foreign keys cannot conflict. Since the two columns will be collapsed into a single column C in the result, there must be an unambiguous target for the foreign key.

Table 4.10 describes the action of Table Equate on the various kinds of Guava statements. In general, any query or DML statement that references table T_2 is changed so that it references T_1. When DDL statements are processed, Table Equate tests to see if there are any domain conflicts that arise when adding columns, verifies that there are no

unexpected side effects, and ensures that the columns of the output table T_1 is always the union of the columns of input tables T_1 and T_2.

For instance, consider the case of $RC(T_2, FName, FirstName)$, a rename column statement. We assert that if one renames a column or field in the application, the user does not intend for there to be any effect on the database beyond renaming the column. However, if column FirstName already exists in table T_1, data from both $T_2.FName$ and $T_1.FirstName$ would both end up being placed in $T_1.FirstName$. If both of these columns already hold data, throw an error.

Finally, Table Equate will change the target of a change spike if it refers to the table that is being dropped. $TableEquate(T_1, T_2)$ will change the spike $\wedge(T_2)$ into $\wedge(T_1)$ as it passes through the channel so that it no longer refers to a table that does not exist.

Inverse: The inverse of the transformation $TableEquate(T_1, T_2)$ is the set of transformations $[Copy(T_1, T_2), DropProject(T_1, \vec{C_1}), DropProject(T_2, \vec{C_2})]$, where $Copy$ is the transformation that simply copies the data from T_1 into T_2, $\vec{C_1}$ is the set of columns in T_2 that are not in T_1, and $\vec{C_2}$ is the set of columns in T_1 that are not in T_2. In effect, the inverse of Table Equate restores the two tables, where any columns in common between the two tables hold the same data.

4.4   RELATED WORK

As mentioned before, the Audit transformation is inspired by work on temporal database systems. The current implementation of the Audit transformation assumes that the software runs on a standard DBMS, i.e., one that has not been augmented with temporal

capabilities on either the data [45] or schema [81] level. In the event that the underlying database actually has temporal support, we would like to be able to have Audit (or a more powerful, fully-temporal semantic operator yet to be created) rely on temporal functionality of the database. For instance, if the database handles row-level timestamping but does not handle temporal schema changes, Audit will refrain from modifying DML statements. Audit should still respond the same way to semantic spikes.

Spaccapietra et al present a high-level modeling construct called an Interschema Correspondence Assertion (ICA) to describe the relationship between the scope of two entity sets [72]. With an ICA, one can describe the relationship between two entity sets as equal, a subset, intersecting, or disjoint. The TableEquate transformation works much like an ICA with an equate relationship, and a ColumnEquate transformation works similarly to the "With Corresponding Attribute" clause of an ICA.

There is a wide variety of other research projects besides ICA's that consider establishing relationships within a relational schema or between relational schemas, sometimes called schema matching [64]. These projects (and ICA's as well) are generally used in an information integration context, where heterogeneous schemas from different sources are brought together into a single, virtual schema, and a database expert subsequently identifies portions of the new integrated schema that refer to the same entities, entity sets, or attributes. Using Guava, assertions may be useful or necessary within a single relational schema outside of any information-integration context (for instance, consider the Student-and-Person example at the beginning of Section 4.3).

4.5   SUMMARY AND IMPLEMENTATION STATUS

In this chapter, we considered ways to extend the expressive power of the channel transformation language. The extensions provided in this chapter, combined with the original seven transformations, are sufficiently expressive to cover the scenarios described in the case studies in Chapter 3. Our new transformations can be grouped into three categories: generalizations of existing transformations, transformations that serve a particular application-specific business purpose, and transformations that serve as a correspondence assertion between schema elements in the channel's input schema. These transformations still satisfy the information-preservation property introduced in Chapter 3. An application can discover what changes are made within the channel by application-specific transformations and correspondence assertions by issuing a change spike. Of the transformations introduced in this chapter, we have thus far implemented the Audit and Adorn transformations.

Chapter 5

FORMAL PROOFS OF CORRECTNESS

The seven physical design transformations are defined in Chapter 3 in terms of how they act upon queries, DML updates, and DDL statements. Each of these transformations has several possible but well-understood semantics in terms of how it acts on instances of relations. For instance, the Pivot transformation is well-understood in online analytical processing [16] and federated databases [82], while partitioning and merging, both horizontally and vertically, are often used in physical design [1]. However, each transformation has several potential variants in the database literature. The Vertical Partition transformation, for example, may or may not eliminate rows in either output table that have all null values for non-key columns, depending on the needs of the application. In this chapter, we demonstrate that our definitions of the action of each physical design transformation on statements respects a particular instance-at-a-time semantics, which we introduced informally in Chapter 3 but introduce formally here. We also formally prove that each transformation satisfies the information preservation property as defined in Section 3.2.

We first document the standard definition of each physical transformation, in terms of its effect on a fully materialized database instance, in Table 5.1. Specifically, each row of the table describes how a transformation affects the schema and the data in an instance. For each transformation shown in one row in the table, the first column serves

as a reminder of the signature of the transformation's parameters. The entry in the second column describes the effect of the transformation on the schema of the database instance on which the transformation is applied; namely, the entry in the column shows precisely how to compute the output schema of the database instance from the input schema. The final column describes the effect of the transformation on the instance data of the database; just as with the schema, the column entry describes how to compute the output data instance from the input.

For each of the physical transformations, we define its data transformation semantics in terms of extended relational algebra, which was introduced in Section 3.1.1. We use extended relational algebra in the table to define the data transformations because each of the extended operators has well-understood semantics in the database literature. We add one additional operator to our extended algebra in the table; the *outer union* operator ($\uplus$) is similar to union, except that the tables need not be union-compatible. The schema of the outer union is the union of the columns of each of the summands, and input rows are padded with enough nulls to fill out any new columns. For instance, the outer union of A(X,Y) and B(X,Z) has columns (X,Y,Z); input rows of A in the form (x,y) will appear in the output as (x,y,null).

Table 5.1 uses the following notation:

- Anything in **boldface** is a set

- **Tables** refers to the set of tables in the schema

- **Cols**(T) refers to the set of columns in the schema for table T

- **Dom**(C) refers to the set of elements in the domain of column C

- $inst(T)$ refers to the instance of table T

- $name(D)$ refers to the name of the column or table D, returned as a data value

- A subscript of "in", such as $inst_{in}$, refers to the input of the transformation

- A subscript of "out", such as $inst_{out}$, refers to the output of the transformation

In this chapter, we prove properties about the transformations as presented in Tables 3.2 through 3.9. Over the next few sections, we prove the following:

- The query translation formulae as given in Table 3.2 are correct with respect to the data transformations in Table 5.1. That is, we prove that a query will return results after a channel is applied that are the same as the results it would return if there were no channel at all.

- The DML translation formulae as given in Tables 3.3 through 3.9 are correct with respect to the data transformations in Table 5.1. We prove that each transformation is similar to an updatable view in that the changes made in the output schema produce exactly the changes desired in the input schema. In other words, each DML statement on table T is processed by a transformation in such a way that if the transformed statement is executed on the physical database and the instance of T is subsequently retrieved by pushing a query through the transformation, the result of the query is exactly what it would be if the original DML statement were applied to T's pre-transformation, pre-DML-statement image.

Table 5.1: Definition of the action of seven physical design transformations. Any table in the input schema or instance that is not explicitly referenced by the transformation is passed to the output unaffected.

Transformation	Schema Action	Data Action
VPartition $(T, \mathbf{Cs}, T^n)$	$\mathbf{Tables_{out}} = \mathbf{Tables_{in}} \cup \{T^n\}$, $\mathbf{Cols_{out}}(T) = \mathbf{Keys_{in}}(T) \cup \mathbf{Cs}$ $\mathbf{Keys_{out}}(T) = \mathbf{Keys_{in}}(T)$, $\mathbf{Keys_{out}}(T^n) = \mathbf{Keys_{in}}(T)$ $\mathbf{Cols_{out}}(T^n) = \mathbf{Keys_{in}}(T) \cup (\mathbf{Cols_{in}}(T) - \mathbf{Cs})$ Create foreign key from $T^n.\mathbf{Keys_{in}}(T)$ to $T.\mathbf{Keys_{in}}(T)$	$inst_{out}(T) = \pi_{\mathbf{Keys_{in}}(T) \cup \mathbf{Cs}}(inst_{in}(T))$ $inst_{out}(T^n) = \pi_{\mathbf{Cols_{in}}(T) - \mathbf{Cs}}(\sigma_{(C_1 \neq null) \vee (C_2 \neq null) \vee \ldots \vee (C_k \neq null)}(inst_{in}(T)))$ where $\{C_1, C_2, \ldots, C_k\} = \mathbf{Cols_{in}}(T) - (\mathbf{Cs} \cup \mathbf{Keys_{in}}(T))$ Recall: columns $\vec{C}s$ are non-key columns
VMerge (T^L, T^R)	$\mathbf{Cols}(T^L) = \mathbf{Cols}(T^L) \cup \mathbf{Cols}(T^R)$, $\mathbf{Tables_{out}} = \mathbf{Tables_{in}} - \{T^R\}$	$inst_{out}(T^L) = inst_{in}(T^L) \bowtie_{\forall_{K \in \mathbf{Keys_{in}}(T^L)}(T^L.K = T^R.K)} inst_{in}(T^R)$
HPartition (T, C)	$\mathbf{Tables_{out}} = (\mathbf{Tables_{in}} - T) \cup \mathbf{Ds}$ $(\mathbf{Ds} = \mathbf{Dom}(inst_{in}(T.C)))$ $\forall T_r \in \mathbf{Dom}(inst_{in}(T.C)), \mathbf{Cols_{out}}(T_r) = \mathbf{Cols_{in}}(T) - \{C\}$ $\forall T_r \in \mathbf{Dom}(inst_{in}(T.C)), \mathbf{Keys_{out}}(T_r) = \mathbf{Keys_{in}}(T) - \{C\}$	$\forall T_r \in \mathbf{Dom}(inst_{in}(T.C)) \, inst_{out}(T_r)$ $= \pi_{\mathbf{Cols_{in}}(T) - \{C\}}(\sigma_{T.C = name(T_r)}(inst_{in}(T)))$
HMerge $(\mathbf{Ts}, T^{result}, C)$	$\mathbf{Cols_{out}}(T^{result}) = \bigcup_{T \in \mathbf{Ts}} \mathbf{Cols_{in}}(T) \cup \{C\}$ $\mathbf{Keys_{out}}(T^{result}) = \mathbf{Keys_{in}}(T)$ for any $T \in \mathbf{Ts}$ $\mathbf{Tables_{out}} = \mathbf{Tables_{in}} - \mathbf{Ts} \cup \{T^{result}\}$	$inst_{out}(T^{result}) = \biguplus_{T \in \mathbf{Ts}}(inst_{in}(T) \times \{(name(T))\})$, where $\biguplus$ is outer union with respect to column name (as opposed to column position)
Apply $(T, \mathbf{C_1}, \mathbf{C_2}, f)$	$\mathbf{Cols_{out}}(T) = (\mathbf{Cols_{in}}(T) - \mathbf{C_1}) \cup \mathbf{C_2}$	$inst_{out}(T) = \alpha_{\vec{C_1}, \vec{C_2}, f} \, inst_{in}(T)$ Recall: α is the function application query operator
Pivot (T, A, V)	$\mathbf{Cols_{out}}(T) = \mathbf{Keys_{in}}(T) - \{A\} \cup \mathbf{Ds}$ $(\mathbf{Ds} = \mathbf{Dom}(inst_{in}(T.A)))$ $\mathbf{Keys_{out}}(T) = \mathbf{Keys_{in}}(T) - \{A\}$	$inst_{out}(T) = \boxtimes_{\mathbf{Cols_{in}}(T) - \mathbf{Ks}, A, V} \, inst_{in}(T)$ Recall: $\boxtimes$ is the pivot query operator
Unpivot (T, A, V)	$\mathbf{Cols_{out}}(T) = \mathbf{Keys_{in}}(T) \cup \{V, A\}$ $\mathbf{Keys_{out}}(T) = \mathbf{Keys_{in}}(T) \cup \{A\}$	$inst_{out}(T) = \boxtimes_{\mathbf{Cols_{in}}(T) - \mathbf{Keys_{in}}(T), A, V} \, inst_{in}(T)$ Recall: $\boxtimes$ is the unpivot query operator

- The DDL translation formulae as given in Tables 3.3 through 3.9 are correct with respect to the schema transformations in Table 5.1. We prove that the physical database schema will always be the after-image of applying the transformation to the input schema according to Table 5.1, as the input schema of the channel evolves via DDL statements.

- The transformations as defined in Table 5.1 are invertible and, therefore, information preserving.

The definitions (and thus the correctness proofs) of many of the actions of the transformations in Tables 3.2 through 3.9 are similar, both across statements and across transformations. Therefore, we prove all of the properties for one transformation, HPartition, and then provide proofs for a representative sample of the rest of the transformations.

The final section of the chapter addresses the additional transformations from Chapter 4. In particular, we prove a representative sample of information-preserving properties of the application-specific transformations and correspondence assertions, given the transformation's definition for query, DML, and DDL processing. We do not prove correctness for these transformations. The Lookup transformation can be expressed as an instance-at-a-time transformation using a join and a projection, but, in general, the actions of application-specific transformations and correspondence assertions cannot be expressed in terms of instance-at-a-time semantics. For instance, consider the Audit transformation. If Audit had an instance-at-a-time semantics, it would act as a function from an input instance to an output instance. However, a single input instance could correspond to any number of output instances through an Audit transformation, depending on the history of statements that were executed against the input instance. So, there is

no such function that one can associate with Audit. For application-specific transformations and correspondence assertions, we treat the action of each transformation on the various classes of statements as the definitive semantics for that transformation.

5.1 PROOFS OF QUERY CORRECTNESS

To prove query correctness of a channel transformation, we demonstrate that the query unfolding presented in the transformation's definition shown in a row in Table 3.2 respects the definition of the transformation in Table 5.1. Specifically, we prove that the single-table query T for any table T yields the result $inst_{in}(T)$, which is the result one would get when executing the query without a channel. If a transformation processes single-table queries correctly, then all queries will be processed correctly, since queries are represented as relational algebra. Relational algebra has the property that, when given a query, any sub-expression of the query can be substituted with an equivalent expression. In channel query processing, a transformation replaces an expression E over the transformation's input schema with an expression E' over the transformation's output schema. So, if we can prove query correctness for single-table queries, we will have proven query correctness for arbitrary queries by induction, using the single-table queries as the base case. The proofs in this chapter demonstrate that E evaluated over the instance with the input schema will produce the same result as E' over the instance with the output schema, satisfying relational algebra's like-for-like substitution property.

Table 3.2 also contains rules for re-writing expressions that are not single-table references. These rules are provided as simple optimizations, and can be easily derived from the single-table reference case using simple relational algebra equivalences. Therefore,

proving properties using single-table references is sufficient. However, we demonstrate one such equivalence using HPartition; we prove the transformation rule for the query $Q = \sigma_{C_{in}=T}(T_a)$ directly.

For each transformation, we start with the original query $Q = T$, for an arbitrary table T. We then apply the transformation to get a query expressed on the transformation's output schema. Then, we replace all references to tables with their output instances according to Table 5.1. Finally, we use equivalences to show the result is equivalent to $inst_{in}(T)$, the query $Q = T$ applied to the input instance.

5.1.1 HPartition: Single-Table Query

Let the transformation $O = HPartition(T_a, C_{in})$ as defined in the third row of Table 3.2. We show that the transformation O re-writes the input query, $Q = T_a$, into a query that produces equivalent output over the output schema. For any other table $T \neq T_a$, transformation O has no effect on either the query or the table instances, so the proof that the transformation processes these other tables correctly is trivial.

Proposition: $O(T_a)$ evaluates to $inst_{in}(T_a)$.

Note: For this and all subsequent proofs, the justification for each step in the proof follows the step to which the justification applies.

Proof: $O(T_a)$

$= \cup_{t \in \mathbf{Dom}(C_{in})}(t \times \{name(t)\})$

    (push the query through the HPartition transformation according to Table 3.2)

$= \cup_{t \in \mathbf{Dom}(C_{in})}(inst_{out}(t) \times \{name(t)\})$
    (evaluate the query by replacing each table reference with its instance in the output

    schema)

$= \cup_{t \in \mathbf{Dom}(C_{in})} (\pi_{\mathbf{Cols_{in}(T_a)} - \{\mathbf{C_{in}}\}} (\sigma_{T_a.C_{in}=name(t)}(inst_{in}(T_a))) \times \{name(t)\})$
(substitute the output instances in terms of the input instances as defined in the

HPartition row of Table 5.1)

$= \cup_{t \in \mathbf{Dom}(C_{in})} (\sigma_{T_a.C_{in}=name(t)}(inst_{in}(T_a)))$
(we know that *name(t)* is the value in the column C_{in} thanks to the σ operator, so

the project operator and the cross-product with the name constant are inverses)

$= \sigma_{T_a.C_{in}=v_1 \vee T_a.C_{in}=v_2 \vee ... \vee T_a.C_{in}=v_k}(inst_{in}(T_a))$ where $\{v_1, v_2, \ldots, v_k\} = \mathbf{Dom}(C_{in})$
(apply a basic relational algebra equivalence — disjunction in a σ operator and

union are interchangeable)

$= inst_{in}(T_a)$
(because $T_a.C_{in} = v_1 \vee T_a.C_{in} = v_2 \vee \ldots \vee T_a.C_{in} = v_k \equiv true$; since $\{v_1, v_2, \ldots, v_k\} =$

$\mathbf{Dom}(C_{in})$, this condition evaluates if the value of C_{in} is in its own domain, which is

trivially true)

□

5.1.2 HPartition: Query Expression with Select

Let the transformation $O = HPartition(T_a, C_{in})$. We show that the transformation O

re-writes query $Q = \sigma_{C_{in}=T}(T_a)$ for some value T into a query that produces equivalent

output over the output schema.

Proposition: $O(\sigma_{C_{in}=T}(T_a))$ evaluates to $\sigma_{C_{in}=T}(inst_{in}(T_a))$.

Proof: $O(\sigma_{C_{in}=T}(T_a))$

$= T \times \{name(T)\}$

(push the query through the HPartition transformation according to Table 3.2)

$= inst_{out}(T) \times \{name(T)\}$
(evaluate the query by replacing each table reference with its instance in the output

schema)

$= \pi_{\mathbf{Cols_{in}(T_a)}-\{\mathbf{C_{in}}\}}(\sigma_{T_a.C_{in}=T}(inst_{in}(T_a))) \times \{name(T)\}$

(substitute the output instances in terms of the input instances defined in the HPar-

tition row of Table 5.1)

$= \sigma_{T_a.C_{in}=T}(inst_{in}(T_a)))$

(we know that $name(T)$ is the value in the column C_{in} thanks to the σ operator, so

the project operator and the cross-product with the name constant are inverses)

□

5.1.3 VPartition: Single-Table Query

Let the transformation $O = VPartition(T_a, \vec{C}s, T_n)$.

Proposition: $O(T_a)$ evaluates to $inst_{in}(T_a)$.

Proof: $O(T_a)$

$= T_a \sqsupset\!\!\bowtie T_n$

(push the query through the VPartition transformation according to Table 3.2)

$= inst_{out}(T_a) \sqsupset\!\!\bowtie inst_{out}(T_n)$

(evaluate the query by replacing each table reference with its instance in the output

schema)

$= \pi_{\mathbf{Ks}\cup\mathbf{Cs}}(inst_{in}(T_a)) \sqsupset\!\!\bowtie \pi_{\mathbf{Cols_{in}(T_a)}-\mathbf{Cs}}(\sigma_F(inst_{in}(T_a)))$ where F is the condition

that at least one column in $\mathbf{Cols_{in}(T_a)} - (\mathbf{Cs} \cup \mathbf{Ks})$ is not null

(substitute the output instances in terms of the input instances defined in the VPar-

tition row of Table 5.1)

$= (\pi_{\mathbf{Ks}\cup\mathbf{Cs}}(\sigma_F(inst_{in}(T_a))) \bowtie \pi_{\mathbf{Cols_{in}(T_a)}-\mathbf{Cs}}(\sigma_F(inst_{in}(T_a))))$

$\cup (\pi_{\mathbf{Ks}\cup\mathbf{Cs}}(\sigma_{\neg F}(inst_{in}(T_a))) \times (null, null, \ldots, null)$ where the null tuple

has the same number of columns as $\mathbf{Cols_{in}(T_a)} - (\mathbf{Cs} \cup \mathbf{Ks})$

(unfold the definition of left outer join to separate rows on the left that participate

in the join from rows on the left that do not (and are thus padded with nulls))

$$= \sigma_F(inst_{in}(T_a)) \cup (\pi_{\mathbf{Ks} \cup \mathbf{Cs}}(\sigma_{\neg F}(inst_{in}(T_a)))) \times (null, null, \dots, null)$$

(because $\pi_{K \cup A} T \bowtie \pi_{K \cup B} T = \pi_{K \cup A \cup B} T = T$ if $K \cup A \cup B$ is the entire schema for T,

and K is the key of the table)

$$= \sigma_F(inst_{in}(T_a)) \cup \sigma_{\neg F}(inst_{in}(T_a))$$

(because we know that the columns in $\mathbf{Cols_{in}(T_a)} - (\mathbf{Cs} \cup \mathbf{Ks})$ are all null in the

right summand, so projecting those columns away and padding back with nulls is

just the identity mapping)

$$= \sigma_{F \vee \neg F} inst_{in}(T_a) = inst_{in}(T_a)$$

(by basic relational equivalences)

$\square$

5.1.4 HMerge: Single-Table Query

Let the transformation $O = HMerge(\mathbf{Ts}, T_a, C_{out})$.

Proposition: $O(t)$ evaluates to $inst_{in}(t)$, for arbitrary $t \in \mathbf{Ts}$.

Proof: $O(t)$

$$= \pi_{\mathbf{Cols}(t)} \sigma_{C_{out}=t} T_a$$

(push the query through the HMerge transformation according to Table 3.2)

$$= \pi_{\mathbf{Cols}(t)} \sigma_{C_{out}=t} inst_{out}(T_a)$$

(evaluate the query by replacing each table reference with its instance in the output

schema)

$$= \pi_{\mathbf{Cols}(t)} \sigma_{C_{out}=t} \biguplus_{T \in \mathbf{Ts}} (inst_{in}(T) \times \{(name(T))\})$$

(substituting the output instances in terms of the input instances defined in the

HMerge row of Table 5.1)

$$= \pi_{\mathbf{Cols}(t)}(inst_{in}(t) \times \{(name(t))\} \times (null)_{pad})$$

(selecting the one entry in the union corresponding to t; because it is an outer union,

pad with nulls for columns in $\mathbf{Cols}(T_a) - \mathbf{Cols}(t)$)

$= inst_{in}(t)$

(evaluate the project operator; since **Cols**(t) is precisely the columns of $inst_{in}(t)$, this evaluation effectively gets rid of the constant cross-products)

□

5.1.5 A Note Regarding Invertibility

In Chapter 3, we list the inverse transformation for each channel transformation. With one exception, the proofs for these inverse relationships follows directly from the proofs of query correctness as follows. Given a transformation O, parameters P, and inverse transformation $O^{-1}(P^{-1})$, we must demonstrate that $(O^{-1}(P^{-1}) \circ O(P))$ is the identity function for single-table queries, and thus for queries in general. Since query transformation is correct with respect to the transformation semantics given in Table 5.1, then the transformations are necessarily inverses of one another with respect to how they transform database instances.

To demonstrate an invertibility proof, consider the case of *VPartition* and *VMerge*.

Let the transformation $O = VPartition(T_a, \vec{C}s, T_n)$, and the transformation $O' = VMerge(T_a, T_n)$.

Proposition: $O'(O(T_a)) = T_a$.

Proof: $O'(O(T_a))$

$= O'(T_a \supset\!\!\!\bowtie T_n)$

(by transforms according to Table 3.2)

$= T_a$

(by transforms according to Table 3.2)

□

In the reverse direction, we must prove $O(O'(T_a)) = T_a$ and $O'(O(T_n)) = T_n$.

Proposition: $O(O'(T_a)) = T_a$.

Proof: $O(O'(T_a))$

$= O(\pi_{columns_{in}(T_a)} T_a)$

    (by transforms according to Table 3.2)

$= \pi_{columns_{in}(T_a)}(T_a \ ⊐\bowtie \ T_n)$

    (by transforms according to Table 3.2)

$= T_a$

    (by relational algebra equivalences)

$\square$

The proof for the processing of query T_n is virtually identical to the above and is omitted.

The exception to the template above for proving invertibility is *HMerge*. *HMerge* performs an outer union of tables, which means that the tables need not be union-compatible. However, *HPartition* necessarily creates union-compatible tables as output. So, for parameters P with inverse parameters P^{-1}, $HMerge(P^{-1}) \circ HPartition(P)$ is the identity function, while $HPartition(P) \circ HMerge(P^{-1})$ is not. For instance, consider two tables, $T(\underline{A}, B)$ and $T'(\underline{A}, C)$. When these tables undergo an $HMerge(\{T, T'\}, T_a, D)$ the resulting schema is $T_a(\underline{A, D}, B, C)$. Running this through $HPartition(T_a, D)$ yields two tables, $T(\underline{A}, B, C)$ and $T'(\underline{A}, B, C)$. Each output table has one extra column, and that column will always be null since the original table instances lack those columns.

Running the query T through $HPartition(T_a, D) \circ HMerge(\{T, T'\}, T_a, D)$ yields $\pi_{A,B} T$, which will always produce the expected results (projecting away the column C,

which did not exist in the original table). This scenario is precisely what should happen, since HMerge and HPartition are both information-preserving. However, to demonstrate invertibility, we must both produce the expected results AND reduce to the single-table query. In this case, the resulting query performs the additional projection step to achieve the proper results, which indicates that the resulting data instance is not the same as the input instance.

We will address the issue of *HMerge* and invertibility in Chapter 6. Note, however, that if we assume that the input tables to *HMerge* are union-compatible to begin with, there is no issue because the transformations's semantics uses a standard union instead of an outer union, which is exactly the query transformation action of *HPartition* in Table 3.2.

5.2 PROOFS OF DML CORRECTNESS

We demonstrate the correctness of DML translation through a transformation by tracking the changes made to table instances by the DML statements. For instance, we describe the correctness a DML statement against table T as a relational query equivalence that expresses the state of a table after the statement ($inst_{out}^{after}(T)$) in terms of the state of the same table before the statement ($inst_{out}^{before}(T)$), where the "out" is as usual the output schema):

- The evaluation of the statement $I(T, \vec{C}, Q)$ is represented as the equivalence:

$$\pi_{\vec{C}} inst_{out}^{after}(T) \equiv (\pi_{\vec{C}} inst_{out}^{before}(T)) \cup Q.$$

  This equivalence illustrates that the data in the $\vec{C}$ columns of T after the insert is

precisely the data that was in the $\vec{C}$ columns of T ($\pi_{\vec{C}}T$), together with the newly-added rows from query Q.

- The evaluation of the statement $U(T, \vec{F}, \vec{C}, Q)$ is represented as the equivalence:

$$inst_{out}^{after}(T) \equiv (\alpha_{\vec{C},\vec{C},f}(\sigma_{\vec{F}}(inst_{out}^{before}(T)))) \cup (\sigma_{\neg\vec{F}}(inst_{out}^{before}(T))).$$

This equivalence illustrates that the rows in the instance after the update are the rows that match the update conditions updated using the function f, which assigns the new values to updated rows, combined with the rows that are not updated.

- The evaluation of the statement $D(T, \vec{F})$ is represented as the equivalence:

$$inst_{out}^{after}(T) \equiv \sigma_{\neg\vec{F}}inst_{out}^{before}(T).$$

This equivalence illustrates that the rows in the instance after the delete are whatever rows existed before the delete that do not meet the delete conditions.

5.2.1   HPartition: Insert

Let the transformation $O = HPartition(T_a, C_{in})$. We show that the transformation O re-writes insert statement $I(T_a, \vec{C}, Q)$ into statements in the output schema such that pushing the query $\pi_{\vec{C}}(T_a)$ through the channel produces the result $\pi_{\vec{C}}(inst_{in}(T_a)) \cup Q$. For any other table $T \neq T_a$, transformation O has no effect on either the insert statement or the table instances, so the proof is trivial.

Proposition: Let $inst_{out}^{before}(T_a)$ be the result of single-table query T_a in the output schema before an insert, and $inst_{out}^{after}(T_a)$ be the result of the same query in the output schema after the insert has been processed. We show that after statement $O(I(T_a, \vec{C}, Q))$, pushing $\pi_{\vec{C}}(T_a)$ through the channel produces the result $\pi_{\vec{C}}(inst_{in}(T_a)) \cup Q$.

Note: $O(I(T_a, \vec{C}, Q)) = \forall_{t \in \mathbf{Dom}(C_{in})} I(t, \vec{C} - \{C_{in}\}, \pi_{\vec{C} - \{C_{in}\}} \sigma_{C_{in}=t} Q)$. Therefore, for any table

$t \in \mathbf{Dom}(C_{in})$, $\pi_{\vec{C} - \{C_{in}\}}(inst_{after}(t)) = \pi_{\vec{C} - \{C_{in}\}}(inst_{in}(t)) \cup \pi_{\vec{C} - \{C_{in}\}} \sigma_{C_{in}=t} Q$ (follows from our

definition of Inserts).

Proof: $O(\pi_{\vec{C}}(T_a))$

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}(t \times \{name(t)\}))$

    (push the query through the HPartition transformation according to Table 3.2)

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}((\pi_{\vec{C} - \{C_{in}\}} t) \times \{name(t)\}))$
    (introducing the new project operator does not alter the result, since all columns not

    in $\vec{C} - \{C_{in}\}$ are projected away by the outermost projection, and the union operator

    and cross-product are unaffected)

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}((\pi_{\vec{C} - \{C_{in}\}} inst_{out}^{after}(t)) \times \{name(t)\}))$

    (evaluate the query with the after-insert instance)

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}(((\pi_{\vec{C} - \{C_{in}\}} inst_{out}^{before}(t)) \cup \pi_{\vec{C} - \{C_{in}\}} \sigma_{C_{in}=t} Q) \times \{name(t)\}))$

    (write after-insert instance in terms of before-insert image and inserted rows Q)

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}(((\pi_{\vec{C} - \{C_{in}\}} inst_{out}^{before}(t)) \times \{name(t)\}) \cup ((\pi_{\vec{C} - \{C_{in}\}}(\sigma_{C_{in}=t} Q) \times \{name(t)\}))))$

    (cross-product distributes with union)

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}(((\pi_{\vec{C} - \{C_{in}\}} inst_{out}^{before}(t)) \times \{name(t)\}) \cup (\sigma_{C_{in}=t} Q)))$
    (we know that $name(t)$ is the value in the column C_{in} thanks to the σ operator, so

    the project operator and the cross-product with the name constant are inverses)

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}((\pi_{\vec{C} - \{C_{in}\}} \pi_{\mathbf{Cols_{in}(T_a)} - \{C_{in}\}}(\sigma_{C_{in}=t}(inst_{in}(T_a))) \times \{name(t)\}) \cup (\sigma_{C_{in}=t} Q)))$
    (substituting the output instances in terms of the input instances defined in the HPar-

    tition row of Table 5.1)

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}((\pi_{\vec{C} - \{C_{in}\}}(\sigma_{C_{in}=t}(inst_{in}(T_a))) \times \{name(t)\}) \cup (\sigma_{C_{in}=t} Q)))$

    (cascading projections)

$= \pi_{\vec{C}}(\cup_{t \in \mathbf{Dom}(C_{in})}((\pi_{\vec{C}} \sigma_{C_{in}=t}(inst_{in}(T_a))) \cup (\sigma_{C_{in}=t} Q)))$

(we know that *name*(t) is the value in the column C_{in} thanks to the σ operator, so

C_{in} need not be projected away)

$= \pi_{\vec{C}}(\cup_{t\in\mathbf{Dom}(C_{in})}((\sigma_{C_{in}=t}\pi_{\vec{C}}(inst_{in}(T_a))) \cup (\sigma_{C_{in}=t}Q)))$
(select commutes with projection, when none of the columns referred to by the

selection are projected away)

$= \pi_{\vec{C}}(\cup_{t\in\mathbf{Dom}(C_{in})}(\sigma_{C_{in}=t}(\pi_{\vec{C}}inst_{in}(T_a) \cup Q)))$

(select distributes over union)

$= \cup_{t\in\mathbf{Dom}(C_{in})}(\sigma_{C_{in}=t}(\pi_{\vec{C}}inst_{in}(T_a) \cup Q))$

(remove redundant projection)

$= (\sigma_{C_{in}=v_1 \vee C_{in}=v_2 \vee ... \vee C_{in}=v_k}(\pi_{\vec{C}}inst_{in}(T_a) \cup Q))$ where $\{v_1, v_2, \ldots, v_k\} = \mathbf{Dom}(C_{in})$

(disjunction in a σ operator and union are interchangeable)

$= \pi_{\vec{C}}inst_{in}(T_a) \cup Q$

(because $C_{in} = v_1 \vee C_{in} = v_2 \vee \ldots \vee C_{in} = v_k \equiv true$, since $\{v_1, v_2, \ldots, v_k\} = \mathbf{Dom}(C_{in})$)

$\square$

5.2.2 HPartition: Delete

Let the transformation $O = HPartition(T_a, C_{in})$. We show that the transformation O re-writes delete statement $D(T_a, \vec{F})$ into statements in the output schema such that pushing the single-table query T_a through the channel should produce the result $\sigma_{\neg\vec{F}}(inst_{in}(T_a))$. For any other table $T \neq T_a$, transformation O has no effect on either the delete statement or the table instances, so the proof is trivial.

Proposition: Let $inst_{out}^{before}(T_a)$ be the result of query T_a in the output schema before a delete, and $inst_{out}^{after}(T_a)$ be the result of the same query in the output schema after the delete has been processed. We show that after statement $O(D(T_a, \vec{F}))$, T_a pushed through

the channel produces $\sigma_{\neg\vec{F}}(inst_{in}(T_a))$.

Recall: $\exists_{<c,v>\in\vec{F}}c = C_{in} \implies O(D(T_a, \vec{F})) = D(v, \vec{F} - \{< c, v >\})$ (from Table 3.5).

Recall: $\nexists_{<c,v>\in\vec{F}}c = C_{in} \implies O(D(T_a, \vec{F})) = \forall_{v\in\mathbf{Dom}(C_{in})}D(v, \vec{F})$ (from Table 3.5).

Case 1: $\exists_{<c,v>\in\vec{F}}c = C_{in}$

Proof: $O(T_a)$

$= \cup_{t\in\mathbf{Dom}(C_{in})}(t \times \{name(t)\})$

(push the query through the HPartition transformation according to Table 3.2)

$= \cup_{t\in\mathbf{Dom}(C_{in})}((inst_{out}^{after}(t)) \times \{name(t)\})$

(evaluate the query with the after-delete instance)

$= (\cup_{t\in(\mathbf{Dom}(C_{in})-\{v\})}((inst_{out}^{after}(t)) \times \{name(t)\})) \cup ((inst_{out}^{after}(v)) \times \{name(v)\})$
(separate out the entry in the union corresponding to v, the value in the $< C_{in}, v >$

filtering condition)

$= (\cup_{t\in(\mathbf{Dom}(C_{in})-\{v\})}((inst_{out}^{before}(t)) \times \{name(t)\})) \cup ((inst_{out}^{after}(v)) \times \{name(v)\})$
(output instances in the $\mathbf{Dom}(C_{in}) - \{v\}$ tables are not affected by the condition

$< C_{in}, v >$, and since the set of conditions $\vec{F}$ has "and" semantics, the instances are

not affected at all by the delete statement)

$= (\cup_{t\in(\mathbf{Dom}(C_{in})-\{v\})}((inst_{out}^{before}(t)) \times \{name(t)\})) \cup (\sigma_{\neg(\vec{F}-\{<C_{in},v>\})}(inst_{out}^{before}(v)) \times \{name(v)\})$
(because v contains all of the rows that satisfy the condition $< C_{in}, v >$, output rows

in the v table are filtered by all of the other conditions in $\vec{F}$, leaving the rows that

do not satisfy at least one of the conditions $\vec{F} - \{< C_{in}, v >\}$)

$= (\cup_{t\in(\mathbf{Dom}(C_{in})-\{v\})}(\pi_{\mathbf{Cols_{in}(T_a)}-\{\mathbf{C_{in}}\}}(\sigma_{T_a.C_{in}=t}(inst_{in}(T_a)))) \times \{name(t)\}))$

$\cup(\sigma_{\neg(\vec{F}-\{<C_{in},v>\})}(\pi_{\mathbf{Cols_{in}(T_a)}-\{\mathbf{C_{in}}\}}(\sigma_{T_a.C_{in}=v}(inst_{in}(T_a)))) \times \{name(v)\})$
(substituting the output instances in terms of the input instances defined in the HPar-

tition row of Table 5.1)

$= (\cup_{t\in(\mathbf{Dom}(C_{in})-\{v\})}(\sigma_{T_a.C_{in}=t}(inst_{in}(T_a))))$

$$\cup(\sigma_{\neg(\vec{F}-\{<C_{in},v>\})}(\sigma_{T_a.C_{in}=v}(inst_{in}(T_a))))$$

(we know that *name*(*t*) (and *name*(*v*) for the second clause) is the value in the column C_{in} thanks to the σ operator, so the project operator and the cross-product with the name constant are inverses)

$$= (\sigma_{C_{in}\in(\mathbf{Dom}(C_{in})-\{v\})}(inst_{in}(T_a))) \cup (\sigma_{\neg(\vec{F}-\{<C_{in},v>\})}(\sigma_{C_{in}=v}(inst_{in}(T_a)))),$$ where the condition

$C_{in} \in (\mathbf{Dom}(C_{in}) - \{v\})$ is equivalent to the disjunction

$$t = c_1 \vee t = c_2 \vee \ldots \vee t = c_n, \text{ for } \{c_1, c_2, \ldots, c_n\} = \mathbf{Dom}(C_{in}) - \{v\}$$
(by basic relational algebra equivalence — disjunction in a σ operator and union are interchangeable)

$$= (\sigma_{\neg<C_{in},v>}(inst_{in}(T_a))) \cup (\sigma_{\neg(\vec{F}-\{<C_{in},v>\})}(\sigma_{C_{in}=v}(inst_{in}(T_a))))$$

(because $C_{in} \in \mathbf{Dom}(C_{in}) - \{v\} \equiv C_{in} \neq v$)

$$= (\sigma_{(\neg<C_{in},v>)\vee(<C_{in},v>\wedge\neg(\vec{F}-\{<C_{in},v>\}))}(inst_{in}(T_a)))$$
(because union becomes disjunction in select, and nested selects become conjunction)

$$= \sigma_{(\neg<C_{in},v>\vee<C_{in},v>)\wedge(\neg<C_{in},v>\vee\neg(\vec{F}-\{<C_{in},v>\}))}(inst_{in}(T_a))$$

(distribution rule)

$$= \sigma_{\neg<C_{in},v>\vee\neg(\vec{F}-\{<C_{in},v>\})}(inst_{in}(T_a))$$

(annihilation rule)

$$= \sigma_{\neg(<C_{in},v>\wedge(\vec{F}-\{<C_{in},v>\}))}(inst_{in}(T_a))$$

(de Morgan's rule)

$$= \sigma_{\neg\vec{F}}(inst_{in}(T_a))$$

(set algebra)

$\square$

Case 2: $\nexists_{<c,v>\in\vec{F}} c = C_{in}$

Proof: $O(T_a)$

$= \cup_{t \in \mathbf{Dom}(C_{in})}(t \times \{name(t)\})$

(push the query through the HPartition transformation according to Table 3.2)

$= \cup_{t \in \mathbf{Dom}(C_{in})}((inst_{out}^{after}(t)) \times \{name(t)\})$

(evaluate the query with the after-delete instance)

$= \cup_{t \in \mathbf{Dom}(C_{in})}(\sigma_{\neg \vec{F}}(inst_{out}^{before}(t)) \times \{name(t)\})$

(write after-delete instance in terms of before-delete image — since none of the conditions in $\vec{F}$ refer to column C_{in}, this re-writing is syntactically valid and correct)

$= \sigma_{\neg \vec{F}} \cup_{t \in \mathbf{Dom}(C_{in})} ((inst_{out}^{before}(t)) \times \{name(t)\})$

(select commutes with cross-product and union)

$= \sigma_{\neg \vec{F}} inst_{in}(T_a)$

(follows from query equivalence proof in Section 5.1.1)

□

The proof for DML Update follows the same line of reasoning as the proof above and is omitted for brevity.

5.2.3   VMerge: Delete

Let the transformation $O = VMerge(T_a, T_n)$. We show that the transformation O re-writes delete statement $D(T_n, \vec{F})$ into statements in the output schema such that pushing the single-table query T_n through the channel should produce the result $\sigma_{\neg \vec{F}}(inst_{in}(T_n))$. For table T_a, operator O has no effect on the delete statement, so the proof is trivial.

Proposition: Let $inst_{out}^{before}(T_a)$ be the result of query T_a in the output schema before a delete, and $inst_{out}^{after}(T_a)$ be the result of the same query in the output schema after the DML statements have been processed on the output schema. We show that after processing statement $O(D(T_n, \vec{F}))$, pushing $O(T_n)$ through the channel should produce

the result $\sigma_{\neg\vec{F}}(inst_{in}(T_n))$.

Recall: $O(D(T_n, \vec{F})) = U(T_a, \vec{F}, \mathbf{Cols}(T_n) - \mathbf{Keys}(T_n), \forall_{c\in\mathbf{Cols}(T_n)-\mathbf{Keys}(T_n)}null)$.

Proof: $O(T_n)$

$= \pi_{\mathbf{Cols}(T_n)}\sigma_{(\mathbf{Cols}(T_n)-\mathbf{Keys}(T_n))\neq null}T_a$, where $(\mathbf{Cols}(T_n) - \mathbf{Keys}(T_n)) \neq null$ is the condition

　　　that not all of the non-key columns of T_n are null

　(push the query through the VMerge transformation according to Table 3.2)

$= \pi_{\mathbf{Cols}(T_n)}\sigma_{(\mathbf{Cols}(T_n)-\mathbf{Keys}(T_n))\neq null}inst_{out}^{after}(T_a)$

　(evaluate the query with the after-delete instance)

$= \pi_{\mathbf{Cols}(T_n)}\sigma_{(\mathbf{Cols}(T_n)-\mathbf{Keys}(T_n))\neq null}(\alpha_{\mathbf{Cols}(T_n)-\mathbf{Keys}(T_n),\mathbf{Cols}(T_n)-\mathbf{Keys}(T_n),f}(\sigma_{\vec{F}}(inst_{out}^{before}(T_a))))$

　　　$\cup(\sigma_{\neg\vec{F}}(inst_{out}^{before}(T_a))))$, where f is the function that sets the columns in

　　　$\mathbf{Cols}(T_n) - \mathbf{Keys}(T_n)$ to null

　(write after-DML instance in terms of before-DML image)

$= \pi_{\mathbf{Cols}(T_n)}\sigma_{(\mathbf{Cols}(T_n)-\mathbf{Keys}(T_n))\neq null}\sigma_{\neg\vec{F}}(inst_{out}^{before}(T_a))$
　(any row that function f operates on will immediately fail the outer selection con-

　　dition)

$= \pi_{\mathbf{Cols}(T_n)}\sigma_{(\mathbf{Cols}(T_n)-\mathbf{Keys}(T_n))\neq null}\sigma_{\neg\vec{F}}(inst_{in}(T_a) ⟕_{\forall_{K\in\mathbf{Keys}(T_n)}(T_a.K=T_n.K)} inst_{in}(T_n))$
　(substituting the output instances in terms of the input instances defined in the

　　VMerge row of Table 5.1)

$= \pi_{\mathbf{Cols}(T_n)}\sigma_{\neg\vec{F}}(inst_{in}(T_a) ⋈_{\forall_{K\in\mathbf{Keys}(T_n)}(T_a.K=T_n.K)} inst_{in}(T_n))$
　(relational algebra equivalence, transforming outer join into inner join because the

　　selection eliminates all rows that have all nulls on the right)

$= \sigma_{\neg\vec{F}}(inst_{in}(T_n))$

　(relational algebra equivalence, valid because the join is along a foreign key)

□

5.2.4 Pivot: Insert

Let the transformation $O = Pivot(T_a, A, V)$.

Because the transformation of an Insert through a Pivot involves the use of the *Loop* construct, there is no clean representation of the translation from $inst_{out}^{after}(T_a)$ to $inst_{out}^{before}(T_a)$ as in the proofs above. Therefore, we describe the correctness of the transformation informally.

By the nature of the Pivot transformation, transformation O effectively takes table T_a and translates it into a matrix, whose vertical axis is labeled by the values of $\mathbf{Keys}(T_a) - \{A\}$ and whose horizontal axis is labeled by the values of $domain(A)$. So, if one finds the value 5 in the row (i.e., key value) marked 17 and column named Z, then the pre-transform table had a row $(17, Z, 5)$. If one finds the value *null* in the row marked 17 and column marked B, then the pre-transform table had no row for key values $(17, B)$.

When one inserts rows into the pre-transformation table, the result on the pivoted table comes in three steps:

- $Error((\pi_{\mathbf{Keys_{in}}(T_a)}Q \bowtie T_a) \cap \square_{\mathbf{Dom}(A),A,V}(\pi_{\mathbf{Cols_{out}}(T_a)}(Q \bowtie T_a)))$: Check to see if there will be any existing non-null entries in the pivoted table that conflict with added rows, and if so, throw an error. The query Q in the error check statement reads as follows: Find all rows in the pivoted table that have a non-null value in a column c and also a row in Q that wants to enter a value in that position in the matrix, thus overwriting the value.

- $\forall_{c \in \mathbf{Dom}(A)} Loop(t, \sigma_{A=c}Q \bowtie (\pi_{\mathbf{Keys}(T_a)-\{A\}}T_a), U(T_a, < \mathbf{Keys}(T_a) - \{A\}, \pi_{\mathbf{Keys}(T_a)-\{A\}}t >, \{c\}, \pi_V t))$: For each row in Q that corresponds to an existing row in the pivoted

table, find the correct row and column in the matrix and set its value.

- $I(T_a, \mathbf{Keys}(T_a) - \{A\}, \Delta^{\rightarrow}_{\mathbf{Dom}(A),A,V}(Q \overline{\bowtie} (\pi_{\mathbf{Keys}(T_a)-\{A\}} T_a)))$: Find the rows in Q that do not correspond to any existing rows in the pivoted table, pivot those, and add them to the table. This step is trivially correct, given the inverse relationship between pivot and unpivot.

Since no values are overwritten, and pivot and unpivot are inverse operations, performing an unpivot on the resulting matrix will necessarily produce the result as if the original matrix were unpivoted, followed by inserting the rows in Q. □

5.3 PROOFS OF DDL CORRECTNESS

We note that a transformation's action on an AddTable statement is faithful to that transformation's action on an instance schema in Table 5.1. Verifying this fact can be done by inspection.

Consider an AddColumn statement. One can combine the statement $S_1 = AddTable(T, \vec{C}, \vec{D}, \vec{K})$ with the statement $S_2 = AddColumn(T, C_0, D_0)$ to get a single statement $S_3 = AddTable(T, \vec{C} \cup C_0, \vec{D} \cup D_0, \vec{K})$ that creates the table T with the new column already added. To prove the correctness of a transformation's action on AddColumn, we compare passing S_1 followed by S_2 through the transformation against passing S_3 through it and verify that they produce the same results.

5.3.1 HPartition: Add Table

We construct the HPartition Add Table translation from visual inspection of HPartition's schema-translation semantics. Table 5.1 shows that $HPartition(T_a, C_{in})$ has the following effect on schema.

First, the old table T_a is removed. Because we are dealing with an Add Table statement, we need not worry about removing an old table. Next, new tables are created, one for each element in $\mathbf{Dom}(C_{in})$. We add clause $\forall_{c \in \mathbf{Dom}(C_{in})}$ to add one table for each domain value. Finally, the schema of each new table corresponds to the original schema of T_a, with the column C_{in} removed from both the set of columns and the set of keys. Each new table draws its name from the domain element. Thus, we need the statement $AT(c, \vec{C} - \{C_{in}\}, \vec{D} - \{\mathbf{Dom}(C_{in})\}, \vec{K} - \{C_{in}\})$ for each value c in the domain of C_{in}. Thus, we arrive at $\forall_{c \in \mathbf{Dom}(C_{in})} AT(c, \vec{C} - \{C_{in}\}, \vec{D} - \{\mathbf{Dom}(C_{in})\}, \vec{K} - \{C_{in}\})$.

The proofs for Rename and Drop Table are trivial.

5.3.2 HPartition: Add Column

Let the transformation $O = HPartition(T_a, C_{in})$.

Let the table T_a begin with columns $\vec{C}$ and associated domains $\vec{D}$ and key columns $\vec{K}$. Now, we want to add a new column C_0 with domain D_0. This action will result in a new table T_a with columns $\vec{C} \cup \{C_0\}$, domains $\vec{D} \cup \{D_0\}$, and keys $\vec{K}$. This proof demonstrates that adding the final table in its entirety is equivalent to adding the original table then adding the column.

Proposition: $O(AT(T_a, \vec{C} \cup \{C_0\}, \vec{D} \cup \{D_0\}, \vec{K}))$

$= O(AT(T_a, \vec{C}, \vec{D}, \vec{K}))$,

$O(AC(T_a, C_0, D_0))$.

Proof: $O(AT(T_a, \vec{C} \cup \{C_0\}, \vec{D} \cup \{D_0\}, \vec{K}))$

$= \forall_{c \in \mathbf{Dom}(C_{in})} AT(c, \vec{C} - \{C_{in}\} \cup \{C_0\}, \vec{D} - \{\mathbf{Dom}(C_{in})\} \cup \{D_0\}, \vec{K} - \{C_{in}\})$

   (Push the Add Table statement through the transformation)

$= \forall_{c \in \mathbf{Dom}(C_{in})} (AT(c, \vec{C} - \{C_{in}\}, \vec{D} - \{\mathbf{Dom}(C_{in})\}, \vec{K} - \{C_{in}\}), AC(c, C_0, D_0))$
   (DDL equivalence — break the Add Table statements for each new table into an

   Add Table followed by an Add Column)

$= \forall_{c \in \mathbf{Dom}(C_{in})} (AT(c, \vec{C} - \{C_{in}\}, \vec{D} - \{\mathbf{Dom}(C_{in})\}, \vec{K} - \{C_{in}\})), \forall_{c \in \mathbf{Dom}(C_{in})} (AC(c, C_0, D_0))$
   (Move all Add Column statements to the end, past Add Table statements for other

   tables)

$= O(AT(T_a, \vec{C}, \vec{D}, \vec{K})), O(AC(T_a, C_0, D_0))$

   (View the statements in their pre-transformation image)

$\square$

The proofs for Rename or Drop Column follow the same line of reasoning as the

above proof and are omitted.

5.3.3   HPartition: Add Element

Let the transformation $O = HPartition(T_a, C_{in})$.

Let the table T_a begin with columns $\vec{C} \cup \{C'\}$ and associated domains $\vec{D} \cup \{D'\}$ and

key columns $\vec{K}$. Now, we want to add a new element E_0 to the domain D' of column C'.

This action will result in a table T_a with columns $\vec{C} \cup \{C'\}$, domains $\vec{D} \cup \{D' \cup \{E_0\}\}$, and

keys $\vec{K}$. We will consider two cases: where $C' = C_{in}$ and where $C' \neq C_{in}$. This proof

demonstrates that adding the final table in its entirety is equivalent to adding the original

table then adding the element.

Proposition: $O(AT(T_a, \vec{C} \cup \{C'\}, \vec{D} \cup \{D' \cup \{E_0\}\}, \vec{K}))$

$= O(AT(T_a, \vec{C} \cup \{C'\}, \vec{D} \cup \{D'\}, \vec{K})), O(AE(T_a, C', E_0)).$

Case 1: $C' = C_{in}$

Proof: $O(AT(T_a, \vec{C} \cup \{C_{in}\}, \vec{D} \cup \{D_{in} \cup \{E_0\}\}, \vec{K} \cup \{C_{in}\}))$ (note that the variable $\vec{K}$ has

been adjusted to reflect the key columns of the table that are not the partition column)

$= \forall_{c \in \{D_{in} \cup \{E_0\}\}} AT(c, \vec{C}, \vec{D}, \vec{K})$

    (Push the Add Table statement through the transformation)

$= (\forall_{c \in \{D_{in}\}} AT(c, \vec{C}, \vec{D}, \vec{K})), AT(E_0, \vec{C}, \vec{D}, \vec{K})$

    (Separate the table corresponding to E_0)

$= O(AT(T_a, \vec{C} \cup \{C_{in}\}, \vec{D} \cup \{D_{in}\}, \vec{K} \cup \{C_{in}\})), O(AE(T_a, C_{in}, E_0))$

    (View the statements in their pre-transformation image)

$\square$

Case 2: $C' \neq C_{in}$

    (Note: whether C' is part of the set $\vec{K}$ is irrelevant to this proof, so long as C' is not

C_{in}.)

Proof: $O(AT(T_a, \vec{C} \cup \{C'\}, \vec{D} \cup \{D' \cup \{E_0\}\}, \vec{K}))$

$= \forall_{c \in \mathbf{Dom}(C_{in})} AT(c, \vec{C} \cup \{C'\}, \vec{D} \cup \{D' \cup \{E_0\}\}, \vec{K})$

    (Push the Add Table statement through the transformation)

$= \forall_{c \in \mathbf{Dom}(C_{in})} (AT(c, \vec{C} \cup \{C'\}, \vec{D} \cup \{D'\}, \vec{K}), AE(c, C', E_0))$

    (DDL equivalence — separate the new elements into Add Element statements)

$= (\forall_{c \in \mathbf{Dom}(C_{in})} AT(c, \vec{C} \cup \{C'\}, \vec{D} \cup \{D'\}, \vec{K})), (\forall_{c \in \mathbf{Dom}(C_{in})} AE(c, C', E_0))$

    (Push the Add Element statements past Add Table statements for other tables)

$= O(AT(T_a, \vec{C} \cup \{C'\}, \vec{D} \cup \{D'\}, \vec{K})), O(AE(T_a, C', E_0))$

(View the statements in their pre-transformation image)

□

The proofs for Rename or Drop Element follow the same line of reasoning as the above proof and are omitted. We also omit the references to foreign keys in the translation since they do not affect the translated instances, but rather only affect DML statements after the statement translation.

5.3.4 HMerge: Rename Column

Let the transformation $O = HMerge(\vec{Ts}, T_a, C_{out})$.

How HMerge transforms the Rename Column statement $RC(T, C_o, C_n)$ for a table T depends on two conditions:

- Whether the old column name also exists in one of the other merged tables

- Whether the new column name also exists in one of the other merged tables

Therefore, there are four possible cases. We prove two of them here: first, where neither the old or new names exist in other tables, and second, where the old name exists but the new name does not. The other two cases follow similar logic, so we omit them.

Proposition: Without loss of generality, assume that $\vec{Ts} = \{t, t'\}$ and that t' is added first, followed by t. Also, account for the possibility that there may be data in table t by providing data via an insert statement. We prove:

$O(AT(t', \vec{C'}, \vec{D'}, \vec{K})), O(AT(t, \vec{C} \cup \{C_2\}, \vec{D} \cup \{D_1\}, \vec{K})), O(I(t, \vec{C} \cup \{C_2\}, Q))$

$= O(AT(t', \vec{C'}, \vec{D'}, \vec{K})), O(AT(t, \vec{C} \cup \{C_1\}, \vec{D} \cup \{D_1\}, \vec{K})), O(I(t, \vec{C} \cup \{C_1\}, Q)),$

$O(RC(t, C_1, C_2))$.

Case 1: $C_1 \notin \vec{C'}, C_2 \notin \vec{C'}$

Proof: $O(AT(t', \vec{C'}, \vec{D'}, \vec{K})), O(AT(t, \vec{C} \cup \{C_2\}, \vec{D} \cup \{D_1\}, \vec{K})), O(I(t, \vec{C} \cup \{C_2\}, Q))$

$= AT(T_a, \vec{C'} \cup \vec{C} \cup \{C_2\} \cup \{C_{out}\}, \vec{D'} \cup \vec{D} \cup \{D_1\} \cup \{\{t, t'\}\}, \vec{K} \cup \{C_{out}\}),$

$\qquad O(I(t, \vec{C} \cup \{C_2\}, Q))$

(Push the two Add Table statements through the transformation)

$= AT(T_a, \vec{C'} \cup \vec{C} \cup \{C_2\} \cup \{C_{out}\}, \vec{D'} \cup \vec{D} \cup \{D_1\} \cup \{\{t, t'\}\}, \vec{K} \cup \{C_{out}\}),$

$\qquad I(T_a, \vec{C} \cup \{C_2\} \cup \{C_{out}\}, Q \times \{name(t)\})$

(Push the Insert statement through the transformation)

$= AT(T_a, \vec{C'} \cup \vec{C} \cup \{C_1\} \cup \{C_{out}\}, \vec{D'} \cup \vec{D} \cup \{D_1\} \cup \{\{t, t'\}\}, \vec{K} \cup \{C_{out}\}), RC(T_a, C_1, C_2),$

$\qquad I(T_a, \vec{C} \cup \{C_2\} \cup \{C_{out}\}, Q \times \{name(t)\})$

(DDL equivalence — separate out the column to rename)

$= AT(T_a, \vec{C'} \cup \vec{C} \cup \{C_1\} \cup \{C_{out}\}, \vec{D'} \cup \vec{D} \cup \{D_1\} \cup \{\{t, t'\}\}, \vec{K} \cup \{C_{out}\}),$

$\qquad I(T_a, \vec{C} \cup \{C_1\} \cup \{C_{out}\}, Q \times \{name(t)\}), RC(T_a, C_1, C_2)$

(Swap the insert and the rename column, changing the name of a column in the

insert to accommodate for the rename column action)

$= O(AT(t', \vec{C'}, \vec{D'}, \vec{K})), O(AT(t, \vec{C} \cup \{C_1\}, \vec{D} \cup \{D'\}, \vec{K})), O(I(t, \vec{C} \cup \{C_1\}, Q)),$

$\qquad O(RC(t, C_1, C_2))$

(View the statements in their pre-transformation image)

□

Case 2: $C_1 \in \vec{C'}, C_2 \notin \vec{C'}$

Proof: $O(AT(t', \vec{C'}, \vec{D'}, \vec{K})), O(AT(t, \vec{C} \cup \{C_2\}, \vec{D} \cup \{D_1\}, \vec{K})), O(I(t, \vec{C} \cup \{C_2\}, Q))$

$= AT(T_a, \vec{C'} \cup \vec{C} \cup \{C_2\} \cup \{C_{out}\}, \vec{D'} \cup \vec{D} \cup \{D_1\} \cup \{\{t, t'\}\}, \vec{K} \cup \{C_{out}\}),$

$\qquad O(I(t, \vec{C} \cup \{C_2\}, Q))$

(Push the Add Table statements through the transformation)

$$= AT(T_a, \vec{C}' \cup \vec{C} \cup \{C_{out}\}, \vec{D}' \cup \vec{D} \cup \{\{t, t'\}\}, \vec{K} \cup \{C_{out}\}), AC(T_a, C_2, D_1),$$

$$O(I(t, \vec{C} \cup \{C_2\}, Q))$$

(DDL equivalence — separate out the column to rename)

$$= AT(T_a, \vec{C}' \cup \vec{C} \cup \{C_{out}\}, \vec{D}' \cup \vec{D} \cup \{\{t, t'\}\}, \vec{K} \cup \{C_{out}\}), AC(T_a, C_2, D_1),$$

$$I(T_a, \vec{C} \cup \{C_2\} \cup \{C_{out}\}, Q \times \{name(t)\}))$$

(Push the insert statement through the transformation)

$$= AT(T_a, \vec{C}' \cup \vec{C} \cup \{C_{out}\}, \vec{D}' \cup \vec{D} \cup \{\{t, t'\}\}, \vec{K} \cup \{C_{out}\}), I(T_a, \vec{C} \cup \{C_1\} \cup \{C_{out}\},$$

$$(Q \times \{name(t)\})), AC(T_a, C_2, D_1), U(T_a, <C_{out}, t>, \{C_2, C_1\}, \{C_1, null\})$$

(Moving the Add Column past the insert is not trivial — the insert must now insert

into the old column C_1, then move the data from C_1 to C_2 after the face)

$$= O(AT(t', \vec{C}', \vec{D}', \vec{K})), O(AT(t, \vec{C} \cup \{C_1\}, \vec{D} \cup \{D'\}, \vec{K})), O(I(t, \vec{C} \cup \{C_1\}, Q)),$$

$$O(RC(t, C_1, C_2))$$

(View the statements in their pre-transformation image, noting that C_1 is one of the

columns in $\vec{C}'$)

□

5.3.5 Pivot: Add Element

Let the transformation $O = Pivot(T_a, A, V)$.

Proposition: $O(AT(T_a, \vec{C} \cup \{C'\}, \vec{D} \cup \{D' \cup \{E_0\}\}, \vec{K}))$

$$= O(AT(T_a, \vec{C} \cup \{C'\}, \vec{D} \cup \{D'\}, \vec{K})), O(AE(T_a, C', E_0)).$$

There are three cases to consider: adding an element to column A, column V, or one

of the key columns other than A. For any key column other than A, the Add Element

statement passes through a Pivot without alteration, with trivial proof (since key columns

that are not A are retained unaltered by the transformation). We consider the other two cases. As in HPartition, we omit the references to foreign keys in the translation.

Case 1: Add Element to column A.

Proof: $O(AT(T_a, \vec{C} \cup \{A\}, \vec{D} \cup \{D' \cup \{E_0\}\}, \vec{K} \cup \{A\}))$

$= AT(T_a, (\vec{C} - \{V\}) \cup D' \cup \{E_0\}, \vec{D} - \{\mathbf{Dom}(V)\} \cup \{\forall_{a \in D' \cup \{E_0\}} \mathbf{Dom}(V)\}, \vec{K})$

(Push the Add Table statement through the transformation)

$= AT(T_a, (\vec{C} - \{V\}) \cup D', \vec{D} - \{\mathbf{Dom}(V)\} \cup \{\forall_{a \in D'} \mathbf{Dom}(V)\}, \vec{K}), AC(T_a, E_0, \mathbf{Dom}(V))$
(DDL equivalence — pull a column definition out of the Add Table statement, then

add it)

$= O(AT(T_a, \vec{C} \cup \{A\}, \vec{D} \cup \{D'\}, \vec{K} \cup \{D\})), O(AE(T_a, A, E_0))$

(View the statements in their pre-transformation image)

□

Case 2: Add Element to column V.

Proof: $O(AT(T_a, \vec{C} \cup \{V\}, \vec{D} \cup \{D' \cup \{E_0\}\}, \vec{K}))$

$= AT(T_a, (\vec{C} - \{A\}) \cup \mathbf{Dom}(A), \vec{D} - \{\mathbf{Dom}(A)\} \cup \{\forall_{a \in \mathbf{Dom}(A)}(D' \cup \{E_0\})\}, \vec{K})$

(Push the Add Table statement through the transformation)

$= AT(T_a, (\vec{C} - \{A\}) \cup \mathbf{Dom}(A), \vec{D} - \{\mathbf{Dom}(A)\} \cup \{\forall_{a \in \mathbf{Dom}(A)} D'\}, \vec{K}),$

$\forall_{a \in \mathbf{Dom}(A)} AE(T_a, a, E_0)$
(DDL equivalence — pull element E_0 out of all of the new columns from $\mathbf{Dom}(A)$,

then add it using Add Element statements)

$= O(AT(T_a, \vec{C} \cup \{A\}, \vec{D} \cup \{D'\}, \vec{K} \cup \{D\})), O(AE(T_a, A, E_0))$

(View the statements in their pre-transformation image)

□

5.3.6 Unpivot: Drop Column

Let the transformation $O = Unpivot(T_a, A, V)$.

Proposition: $O(AT(T_a, \vec{C} - \{C_0\}, \vec{D} - \{D_0\}, \vec{K})) = O(AT(T_a, \vec{C}, \vec{D}, \vec{K})), O(DC(T_a, C_0))$.

Proof: $O(AT(T_a, \vec{C} - \{C_0\}, \vec{D} - \{D_0\}, \vec{K}))$

$= AT(T_a, \vec{K} \cup \{A, V\},$

$\qquad \{d \in (\vec{D} - \{D_0\}) | col(d) \in \vec{K}\} \cup \{\vec{C} - \vec{K} - \{C_0\}\} \cup \{d | col(d) \in \vec{C} - \vec{K} - \{C_0\}\},$

$\qquad \vec{K} \cup \{A\})$

(Push the Add Table statement through the transformation)

$= AT(T_a, \vec{K} \cup \{A, V\}, \{d \in \vec{D} | col(d) \in \vec{K}\} \cup \{\vec{C} - \vec{K} - \{C_0\}\} \cup \{d | col(d) \in \vec{C} - \vec{K} - \{C_0\}\},$

$\qquad \vec{K} \cup \{A\})$

(Domain D_0 is not a key column's domain)

$= AT(T_a, \vec{K} \cup \{A, V\}, \{d \in \vec{D} | col(d) \in \vec{K}\} \cup \{\vec{C} - \vec{K} - \{C_0\}\} \cup \{d | col(d) \in \vec{C} - \vec{K}\},$

$\qquad \vec{K} \cup \{A\})$

(Column C_0 has the same domain as some other non-key column, since our version

of Unpivot requires non-key columns to have the same domain, and does not add to

the selection for the domain of column V)

$= AT(T_a, \vec{K} \cup \{A, V\}, \{d \in \vec{D} | col(d) \in \vec{K}\} \cup \{\vec{C} - \vec{K}\} \cup \{d | col(d) \in \vec{C} - \vec{K}\}, \vec{K} \cup \{A\}),$

$\qquad DE(T_a, A, C_0)$

(DDL equivalence)

$= O(AT(T_a, \vec{C}, \vec{D}, \vec{K})), O(DC(T_a, C_0))$

(View the statements in their pre-transformation image)

$\square$

5.4 PROOFS OF INFORMATION PRESERVATION

Proving that the physical design transformations are information preserving follows from proving that their query-rewriting algorithms are faithful to their semantics in Table 5.1. In other words, we have demonstrated that, for any physical design transformation O and table T, $O(T) = inst_{in}(T)$.

In this section, we prove that for any application-specific transformation or correspondence assertion O and table T, $O(T) = inst_{in}(T)$. Proofs for TableEquate and Adorn are trivial, so we focus on cases for ColumnEquate and Audit.

5.4.1 ColumnEquate: Insert

Let the transformation $O = ColumnEquate(T_1, C_1, T_2, C_2)$. Consider the case where we insert a single row Q into table T_2 (inserting rows into T_1 is unaffected by transformation O and is thus a trivial case). From Table 4.9, we know that no existing rows of T_2 will be affected. Therefore, without loss of generality we can assume that table T_2 is empty before the insert.

Proposition: $O(T_2) = Q$ (the inserted row).

Proof: $O(T_2)$

$= \pi_{Columns_{in}(T_2) - \{C_2\} \cup \{C_1\}} \mathcal{J}_{C_1 C_2}$ (where the $\mathcal{J}$ symbol represents the join path)

(pushing the query through its definition in Table 4.9)

Note: Out of the join expression $\mathcal{J}_{C_1 C_2}$, we only use the columns from T_2 and the column C_1 from table T_1 in the query answer. If there are any intermediate tables, none of their columns are used in the result. We also know the following:

- After the insert, the rows in $inst_{out}(T_2)$ are exactly $\pi_{Columns_{in}(T_2) - \{C_2\}} Q$.

- Table T_2 is a weak entity (according to the definition of ColumnEquate's parameters), so the new row Q will join with exactly one row $t' \in T_1$ through the join path.

- The *Loop* statement in ColumnEquate's transformation of an Insert will iterate over exactly one row in T_1, and that row is the same row as the previous step (t'). So, t' will have its value of C_1 updated to $\pi_{C_2}Q$.

Continuing the proof from above:

$$= \pi_{Columns_{in}(T_2)-\{C_2\}\cup\{C_1\}}((\pi_{Columns_{in}(T_2)-\{C_2\}}Q) \bowtie \ldots \bowtie t') = Q$$

(by the logic above)

$\square$

5.4.2 Audit: Update

Consider the case where the application issues an update statement on audited table T with conditions $\vec{F}$: $U(T_i, \vec{F}, \vec{C}, Q)$. The effect to the application is that the query $Q_1 = \sigma_{\vec{F}}T$, the query that finds all rows that satisfy the conditions $\vec{F}$, should reflect the new values. In addition, the query $Q_2 = \sigma_{\neg\vec{F}}T$ should remain unchanged. Proving that Q_2 is unchanged is trivial because the logic for update translation only affects rows that satisfy $\vec{F}$.

The translation of the update statement becomes an insert followed by an update; however, all the inserted rows have a non-null end timestamp. When the query Q_1 is passed through the channel, it becomes $\sigma_{E=null\wedge\vec{F}}T$, which means that all of the inserted rows are ignored. What remains are the updated rows, whose non-timestamp columns

are updated in exactly the same fashion as the original update statement. Therefore, Q_1 returns exactly the rows that were changed with the proper changes.

$\square$

5.5 SUMMARY

In this chapter, we introduce proof techniques that allow us to reason about whether channel transformations have correct definitions and are information-preserving. We further prove, using these techniques, that the HPartition transformation is both correct with respect to its classical definition and information-preserving. We use a representative sample of proofs to provide evidence that the other six physical design transformations are also correct and information-preserving. Finally, we use another sampling of proofs to demonstrate that our application-specific transformations and correspondence assertions are information-preserving.

Chapter 6

EVOLUTION IN GUAVA: GENERATING DATABASE UPGRADE SCRIPTS

When a database-backed application moves from one version V_1 to a new version V_2, existing instances of databases that conform to version V_1 must be upgraded so as to function with the new version of the application. Frequently, that upgrade is performed by a script generated by a database developer based on the developer's understanding of what has changed in the application. Thus, this process is typically manual. As a result, database upgrade scripts are potentially error-prone and require frequent testing on multiple different database instances. Note that creating the new version of the application may involve modification of the user interface as well as changes to the physical database.

With Guava, we provide a mechanism to generate provably-correct database upgrade processes automatically, leveraging the fact that channel transformations already understand how to translate DDL statements. In Guava, the developer starts with the natural schema, and applies channel transformations until the desired physical schema is achieved. Thus, the database is the direct result of the user interface specification and the channel. This functional relationship can be thought of as:

$$f(UI, Channel) = DB$$

Therefore, a change to the user interface results in a potential change to the schema

and data of the underlying database:

$$f(UI', Channel) = DB'$$

Then, in a similar fashion, changes made to the channel will have a potential impact on the physical schema and data:

$$f(UI', Channel') = DB''$$

Since the UI and the channel together fully define the physical database, we can describe how to migrate a database from one version to another in terms of the changes made to the user interface and to the channel.

For the UI half of the equation, we characterize the changes that the developer makes to UI to create UI' in terms of DML and DDL statements against the application's natural schema; we then push those changes through the channel (the old channel, not the changed one). In effect:

$$f(\Delta UI, Channel) = \Delta DB$$

where ΔDB applied to DB yields DB'.

For changes to the channel, we run an algorithm that takes as input the old channel *Channel* and the new channel *Channel'* and generates a third channel called an *upgrade channel*, one that maps the physical database DB' to DB''. So, to upgrade a database from one version to another, one pushes a set of changes through the old channel to accommodate changes made to the UI, then pushes the resulting database through an upgrade channel to handle changes made to the channel.

In this chapter, we make the following research contributions:

- We construct a framework that describes changes made to the UI and channel components, and translates those changes into a database upgrade process.

- We evaluate our upgrade framework by using it to describe the changes that were made to a publicly-available software product during a recent upgrade.

For the purposes of this chapter, we make the assumption that upgrades are performed by a single developer, or by developers using a version control system that provides mutually exclusive access to files. In Chapter 7, we consider the case where developers work using a version control system that allows simultaneous editing of files and merging.

6.1 CAPTURING CHANGES TO THE USER INTERFACE

As a developer uses the form builder in Visual Studio using Guava components, Guava captures any changes made to the data-bound controls and records relevant changes in a *version change log* (VCL). For instance, Guava will record that a control is added or that its domains elements have changed, but it will not record that it was moved or resized, since those properties are not relevant to the state of the database. The VCL is a file that is physically stored with the code of the application and packaged with the final software product. The version of the application can be verified against the version of the database whenever the application launches.

Every time a new build of the application is created, a checkpoint *ChkPt*(*X.Y.Z.Build*) is added to the VCL, where *X.Y.Z* is the current version of the application and *Build* is a monotonically-increasing number that increments with each build. Then, every time the application runs (in development or in production), Guava compares the version of

the application against a label in the database, finds the checkpoint corresponding to the database label, and copies the VCL from that point forward into a new location (called the *active VCL* (AVCL)) so Guava may optimize it. The AVCL represents the list of changes that is required to bring the database up to the necessary version.

The VCL is an always-growing list of changes; it may be possible to use equivalences to create a shorter list of changes that is equivalent to the complete list. For instance, if one adds a field, then deletes that field, those two changes could be removed from the log. We apply a collection of equivalences on the AVCL to simplify it, including the following:

- Eliminate like Add-Drop pairs. The statements $AC(F, C, D)$ and $DC(F, C)$ are inverses and cancel each other out, if they appear in that order and no DML statements in the AVCL that reference C occur between them.

- Collapse rename statements. An add column $AC(F, C, D)$ followed by renaming the same column $RC(F, C, C')$ becomes $AC(F, C', D)$.

- If any element DDL statement follows an Add Column statement, they can be combined. For instance, $AC(F, C, D)$ then $AE(F, C, E)$ becomes $AC(F, C, D')$, where the domain D' is the domain D with the element E added to it.

- If a drop table $DT(F)$ appears, all statements that reference F in the current version log prior to the Drop Table statement are eliminated.

Once the AVCL has been optimized, it is ready to be sent to the database. At this point, it is just a collection of DML and DDL statements, which means that it can be sent to the database through the channel, as a transaction.

For the rest of this section, we consider how developer actions produce statements in the VCL.

6.1.1 Atomic Changes

Changes are broken down into two categories: atomic and compound. An atomic change corresponds to a single, natural action in a form designer relating to a data-bound control. For instance, placing a new text box onto a form is an atomic change. Each atomic change generates a corresponding DDL statement to be placed in the VCL in a straight-forward manner:

- Renaming a form F to F' creates a Rename Table statement $RT(F, F')$

- Adding a new control C to a form F creates an Add Column statement $AC(F, C, D)$, where D is the underlying domain of the control C

- Renaming a control C to C' on form F creates a Rename Column statement $RC(F, C, C')$

- Deleting a control C on form F creates a Drop Column statement $DC(F, C)$

- Adding a new item E to the available entries on a finite-domain control C on form F creates an Add Element statement $AE(F, C, E)$

- Renaming one of the entries in a finite-domain control C on form F from E to E' creates a Rename Element statement $RE(F, C, E, E')$

- Deleting an item E from a finite-domain control C on form F creates a Drop Element statement $DE(F, C, E)$

Adding or dropping tables is handled in a different manner. Guava does not monitor whether new classes representing forms are added or deleted from a project, since simply adding a new form class to a project does not actually make it appear in the application. A new form only appears in an application (and, consequently, in the physical storage) once a launch relationship has been established to it from an existing form in the application.

For example, consider the situation where a developer creates a new form F. The developer then drops a button on form F' that launches form F. The result in the VCL is an Add Table statement for form F, followed by a foreign key statement from F to F'. If the developer then drops a button on form F'' that launches form F, the VCL adds a new foreign key statement from F to F''. Thus, we leverage the fact from Chapter 3 that Tier 2 foreign keys are additive, and that they can be specified by multiple, temporally separated statements.

Thus, Guava recognizes changes to a form in a form-building application for not just modifications to the data-bound controls on forms, but also controls that launch forms:

Add one-to-one relationship: The developer adds a control that has a single-launch relationship with its target B to a form T. Add the table T if it does not exist yet. Then, create a new foreign key statement $FK(T.id \rightarrow B.id)$, a foreign key from the key column of the child form T to the parent table B.

Add many-to-one relationship: The developer adds a control that has a multiple-launch relationship with its target B to a form T. Add the table T if it does not exist yet. Then, create a new foreign key column fk in table T if one does not yet exist, and a new foreign key statement $FK(T.fk \rightarrow B.id)$ from the new column to the primary key of the

parent form's table.

Drop one-to-one or many-to-one relationship: The developer deletes a control that launches another form. Create a "drop foreign key" statement $DFK(T.c \to B.id)$, where c is either id or fk depending on the nature of the relationship. Also, drop the table T if this relationship was the last one for the form (in other words, if it is the last occurrence of the table T in the application's g-tree).

Redirect launch: The developer takes a control that currently launches form T and changes the control so that it launches T'. This situation is identical to the Drop Relationship action, followed by an Add Relationship action.

We currently do not support the situation where a one-to-one relationship changes to a many-to-one relationship, or vice versa, or where a single table participates as the child in both one-to-one and many-to-one relationships. This situation may not be difficult, but we have not yet seen a case in software where it is necessary; our experience with CORI and our observations with other GUI software has been that each form is specifically designed to be either a details window for another screen or a complete entity on its own, but not both.

6.1.2 Compound Changes

A compound change, by contrast, represents a higher-level change than those found in Table 3.1. Compound changes consist of two or more atomic ones, but may have some added significance when considered together. A compound change may also involve some moving or changing of data, and thus might create some DML statements in addition to DDL statements.

Of particular interest to us are compound changes called *user interface refactorings*. A UI refactoring is a compound but still incremental update to the user interface that leaves invariant the data elements that are modeled by the UI. Some example UI refactorings are:

Create Details Window: This refactoring takes a single form and distributes its controls between two forms. Figure 6.1 shows an example of the result of this action. In the figure, a button named "Details" launches the new window on the right. This action breaks down into an Add Table statement for the new form (table), an Insert statement to move data from the parent form to the details form, and a sequence of Drop Column statements to remove the now unnecessary columns from the parent table.

Merge Details Window: This refactoring takes two forms that have a one-to-one relationship and merges their controls onto a single form. This action is the inverse from the Create Details Window action. The Merge Details Window action comprises a sequence of Add Column statements (to add the controls from the details window to the main one), an Update statement (to move data from the details window to the main one), and finally a Drop Table statement to eliminate the details form.

Move Field: The Move Field compound change takes a data-bound control off of one form and places it on another form. There must be a one-to-one relationship between the two forms, such as the relationship between a form and a details window. This way, there is an unambiguous way to move data from the old control to the new one. This action breaks down into an Add Column in the destination form, an Update statement to move the data, and a Drop Column statement to remove the column from the old table.

Alter Field Domain: This compound change allows the developer to change the

elements in a finite domain field, such as a radio list. It may also be used when the developer wants to change a control from one atomic-domain type to another, such as from a drop-down list to a free-text field. If the translation of values from the old domain to the new one is not trivial, the developer may specify a function to transform the values. This compound change comprises an Add Column statement to create a temporary place to store the new data, an Update to do the value translation from the old domain to the new, a Drop Column to get rid of the old column, and a Rename Column so that the column holding the new domain values takes on the old column's name.

The Alter Field Domain refactoring differs from the atomic Add-Rename-Drop Element statements in that the Alter Field Domain action describes how to migrate the data from the old domain to the new. The Drop Element statement $DE(F, C, E)$, for instance, has the semantics of setting all of the C values to *null* for rows in the table F whenever the value E is present. With an Alter Field Domain action, the developer has the option of specifying what the new value for C would be for these rows. The Rename Element statement does describe how to migrate data by changing all instances of element E_{old} into instances of element E_{new}; however, the transformation is limited to a one-to-one element migration.

Alter Field Domain can be extended to multiple controls, both in the before and after state of the refactoring. The only difference is that the number of Add Column statements (and Drop Column statements) must be the same as the number of controls in the output (and input) of the refactoring, and that there is a function defined for each of the controls in the output. One example of a multiple-control Alter Field Domain refactoring would be changing a single drop-down list into a collection of check boxes.

Map Complex Domain: This compound change allows the developer to change a control from one type to another, but where either the old control or the new control do not have atomic type, but rather have entity type (for instance, consider the grid control in Figure 2.3). This case is similar in usage pattern to the Alter Field Domain action in that the developer can specify a function (or collection of functions) to move data from the old control to the new. The difference in usage is that the developer must specify a function for each output column to populate.

Behind the scenes, the Map Complex Domain refactoring produces output similar to the Alter Field Domain action, except with tables instead of columns. The result is an Add Table statement if the table does not yet exist, a Tier 1 Foreign Key statement to link the table to the containing form's table, and an Insert statement to populate the table. Finally, cleanup of the old control includes a Drop Foreign Key statement, and either a Drop Table statement or a Delete, depending upon whether any other controls use that table, discovered by searching through the application's g-tree.

Figure 6.1 presents two different form representations of the same data. One gets from the form in Figure 6.1(a) to the forms in Figure 6.1(b) by applying three refactorings: create a details window (split the form in two), and two alterations of field domain (date of birth, ethnicity). Since all of the typical motions in a form builder map to atomic actions, the IDE needs an additional way for a the developer to specify a directive that a compound change is needed. Figure 6.2 demonstrates a graphical way that the developer may specify a compound change in a form-based IDE; in the case of the figure, the developer can refactor the Date of Birth field on the form into other classes of control with different underlying data types.

(a) One form in a user interface

(b) The same form after 3 refactorings; clicking on the Details button launches the form on the right

Figure 6.1: Refactoring a user interface; the forms in both (a) and (b) model the same information in different ways
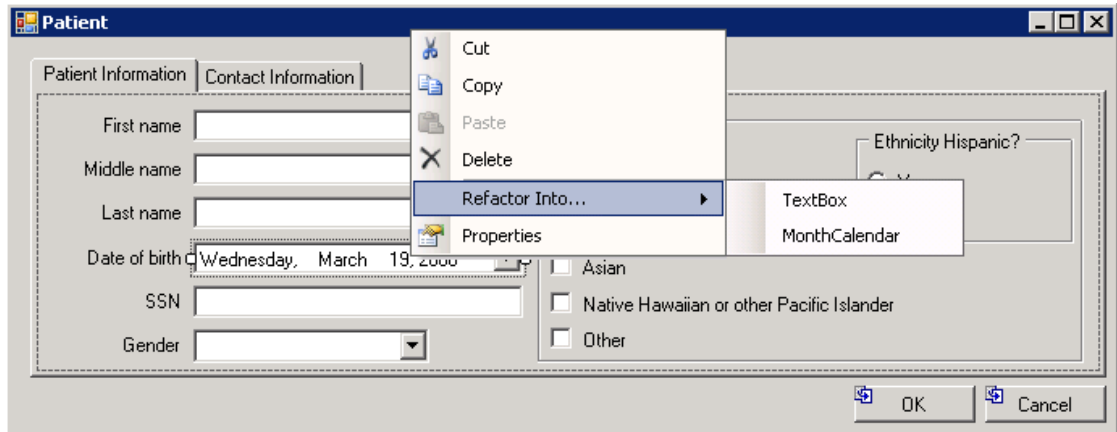
Figure 6.2: Using the form builder to perform a refactoring from Figure 6.1

6.2 EVOLVING CHANNELS

In addition to tracking changes to the user interface, Guava must also track changes made to a channel. In this section, we outline a strategy for translating changes to a channel into changes to a database instance. We also describe a way to handle database refactoring that involves channel evolution.

6.2.1 Comparison Approach

Our approach for translating channel changes into database updates involves comparing the before and after states of the channel. As shown in Figure 6.3, the *comparison approach* relies on identifying the point of commonality closest to the beginning of the channel, whereby the before and after states of the channel are identical up to that point. The algorithm for the comparison approach is as follows, where C_b is the before-image of the channel, and C_a is the after-image:

- Starting from the beginning of both C_b and C_a, move through the channel transformations one-at-a-time until there is a difference between the two.

- In the rest of the channel, determine if there are any pairs of transformations (one from each channel) that are identical; if there are, attempt to use commutativity equivalences to move them each before the point of commonality to provide a maximal prefix in common. For instance, in Figure 6.3, the transformation $O7$ appears in both channels, but if $O7$ and $O6$ cannot commute, then it cannot be moved earlier in the channel. If $O7$ can commute with earlier transformations, enough to appear before both $O4$ and $O4'$, then we do so.

- Construct two channels C_b' and C_a', where C_b' is the inverse of the portion of C_b after the point of commonality and C_a' is the portion of C_a after the point of commonality. In effect, construct a channel that brings the old database instance back to the point of commonality, and another channel that pushes the database instance from the point of commonality through the new portion of the channel.

- Create the channel C_u which is the concatenation of C_b' and C_a'. The channel C_u is called an *upgrade channel*.

- Create copy inserts (introduced in Chapter 3) for each table in the old database schema and push them through the upgrade channel to move data from the old schema to the new.

This approach has the benefit of being straightforward from the framework we have already laid out for channels; since each transformation has an inverse that can also be expressed as a list of transformations, there is always an upgrade channel. However,

All transformations have inverses

Reverse the channel to the point of commonality

Forward the new channel from that point

$O7^{-1}$
$O6^{-1}$
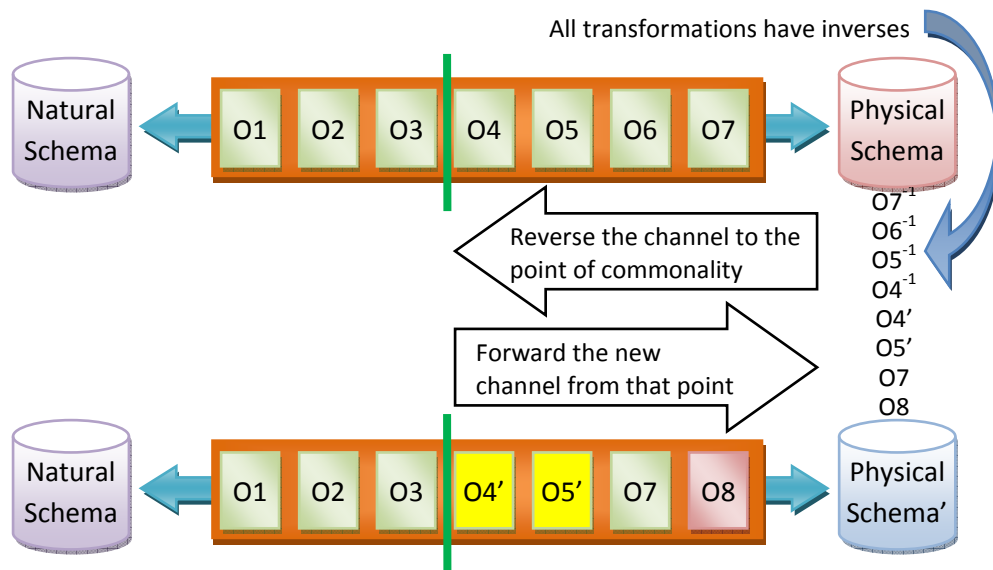$O5^{-1}$
$O4^{-1}$
$O4'$
$O5'$
$O7$
$O8$

Figure 6.3: Translating changes to a channel into changes to a database by comparing the channel against its state before any changes

the resulting upgrade channel is potentially lossy in the presence of application-specific transformations. Consider the case of the Adorn transformation, which adds environment data to tuples as they pass through. The inverse transformation is DropProject, which drops columns from tuples and is guaranteed to be lossy. In Figure 6.3, if transformation $O7$ is the Adorn transformation, all of the data in the Adorned columns will be dropped as it passes through the upgrade channel and readded later, almost certainly to values that are different from the original instance. This behavior is incorrect and undesirable.

Transformations whose inverses are information-preserving do not suffer from this problem, so only the application-specific transformations are troublesome. The inverse of HMerge involves a DropProject, but the columns being dropped necessarily contain

null values. Our expectation is that most of the time the developer will be adding new transformations to the existing set rather than modifying or deleting them. In this case, no loss of information will occur because the existing application-specific transformations occur before the point of commonality. We consider an alternative approach to channel evolution involving capturing channel changes in Chapter 7.

6.2.2 UI Refactoring and the Channel

When we introduced user-interface refactorings, we described them in terms of DML and DDL statements that get executed against the natural schema. In other words, we assume that the channel remains unchanged. For instance, consider the case in Figure 6.4, where we issue a Create Details Window refactoring resulting in the two forms at the top of the figure. In Figure 6.4(a), that refactoring manifests itself as an Add Table statement, an Insert statement, and some Drop Column statements for the columns moved to the details window.

Figure 6.4(b) demonstrates an alternative way to translate the Create Details Window refactoring. Rather than generating update statements to be placed in the VCL, this alternative creates a new channel transformation and prepends it to the channel. The new transformation — in this case, a VMerge — offsets the effects of the UI refactoring, leaving both the rest of the channel (i.e., the pre-existing channel) and the physical database unchanged. The Create Details Window refactoring acts exactly like a vertical partition transformation on a form, so adding a VMerge to the channel adds the inverse transformation to the channel to ensure no net effect.

Not every refactoring can be translated in this way. For instance, the Alter Field

(a) Refactoring option 1: translate refactoring into updates through channel

(b) Refactoring option 2: translate refactoring into a channel transformation
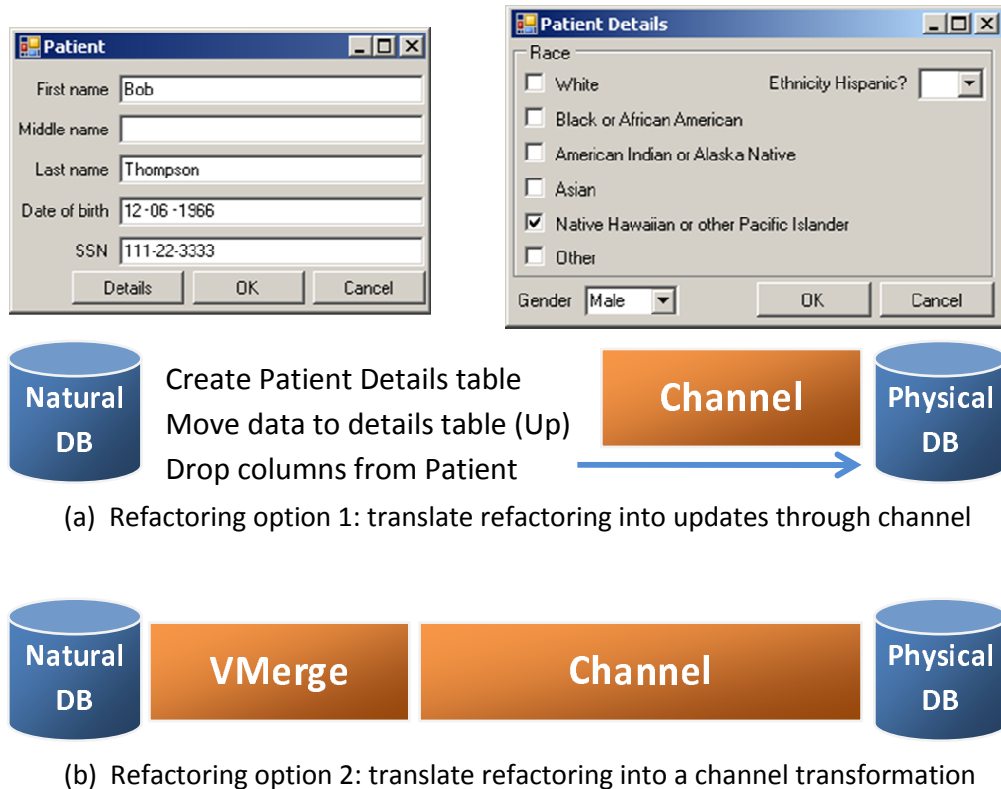
Figure 6.4: Two different options for translating user interface refactorings

Domain transformation may include a non-invertible function that cannot be offset by an Apply transformation. In addition, for those refactorings that can either translate as updates or as a channel transformation, it is left to the developer to decide which is the appropriate choice based on physical-design requirements and performance.

6.3 CASE STUDY

During the writing of this dissertation, CORI has been in the process of developing version 4.1 of its software. The first version, 4.0.23, was completely different from the 3.x versions of the software in that version 4.0.23 was re-written from scratch. Also,

the 3.x versions of the software were written in Delphi, whereas the 4.x versions of the software are written in C#. Version 4.1 is the second version of the current incarnation of the software, and features incremental changes above the 4.0.23 version. Many of the changes in the software between the two 4.x versions are functional in nature and do not affect the data content of the application or data persistence; for instance, the new version fixes a number of issues with PDF generation and faxing of reports. However, several of the changes between the two versions directly affect the data content of the user interface.

This case study demonstrates that our approach — capturing changes made to the UI and to the channel independently — is sufficient to describe the changes made between versions of the CORI software.

6.3.1 Changes to Data Content of the GUI

In this subsection, we enumerate the kinds of changes that developers are making to CORI for the 4.1 version of the software. This list is not comprehensive, but it is representative because it includes examples of each type of change made.

Rename a form: There are at least two instances of forms that were renamed. The form `ColAdenamatousPolypDetail` was renamed to `ColIndAdenoPolyp`, and the form `ColColorectalCancerDetail` was renamed to `ColIndCRC`. These two changes translate to Rename Table statements against the natural schema.

Alter control types: There is one situation in the software upgrade where the data requirements on a form (`IndicationsDetails`) changed. One of the controls on the form was a radio button array (named "Anemia Type"), where the user could select one

of three options: Iron Deficiency, Pernicious Anemia, and Other Anemia. In the new version of the software, it was determined that the user should be able to select each of these options independently; for instance, the user should be able to indicate Iron Deficiency and Pernicious Anemia simultaneously. To support the new requirements, the old radio-button-array control was removed and three check boxes were put it its place, one for each anemia option.

In the natural schema, this process correlates with an Alter Field Domain refactoring. The functions to define the data propagation from the old control to the new controls are simple; for instance, for Iron Deficiency, the value in the column should be $IronDeficiency = (AnemiaType == \text{“Iron Deficiency”})$ (so that IronDeficiency contains a boolean value, describing whether AnemiaType had the particular value "Iron Deficiency").

New procedure type: The 4.0.23 version of the software supports four different types of procedures: colonoscopy, EGD, ERCP, and flexible sigmoidoscopy. The new version supports one additional procedure type (CAPSULE). The new procedure type requires a number of new screens to be built, including new types of findings that are specific to that procedure. However, some screens are reused as well; surgical history and medical history, for example, are already existing forms that are "details" forms of each of the previous procedure types that will also serve as a details form for the CAPSULE procedure.

The changes associated with the new procedure type correlate with Add Table statements, along with Foreign Key statements to associate the forms together.

6.3.2 Changes to Channel

Of course, the current implementation of CORI does not use Guava as its framework, so any changes that they make to their middleware to accommodate the GUI changes were done manually. In addition, there are no changes that CORI is making in version 4.1 that are motivated strictly by physical-design decisions; all of the changes that will occur to CORI's database between these two versions are strictly due to changes in the information modeled in the CORI UI. As a footnote, it should be noted that CORI has not yet addressed the issue of generating an upgrade script to migrate data from version 4.0.23 to version 4.1, though they will need to before the new version is released.

In this subsection, we describe the changes that CORI would need to make in its channel to accommodate each of the UI updates. Recall that we defined part of the channel associated with version 4.0.23 of the CORI software in Chapter 3.

Rename a form: No changes to the channel need to be made; the Rename Table statements propagate through the system, and unless a particular transformation in the channel has a direct conflict with the new name, the statement should move through the channel without incident.

Alter control types: Updates may need to be made to channel transformations that refer to columns. In this case, any time the column "AnemiaType" appears in an argument to a transformation, it must be replaced by the three columns "IronDeficiency", "PerniciousAnemia", and "OtherAnemia". For instance, looking at Case Study 1 in Chapter 3, we can assume that there are several VPartition transformations that partition procedure data according to columns, including Anemia Type, that will need to be updated with the new columns.

New procedure type: As evidenced by the sample channel in Case Study 1, as well as the proposed FindingMerge transformation, all of the data for the new procedure type will need to be merged into the three procedure tables. To accomplish this task, enough VMerge transformations must be added to the channel to merge any details tables with the primary table for CAPSULE. Then, the primary HMerge transformations that merge the procedures together and the findings together must be augmented to bring in the new procedure and the new procedure-specific findings. Finally, the VPartition transformations that divide the columns according to data type must be updated to include any new columns that CAPSULE or its findings introduce, and the domain-aligning Apply transformations must be augmented to accommodate the same columns. At the moment, a developer would need to perform these channel modifications manually, since the transformations monitor columns and tables by name.

6.4 RELATED WORK

There is a considerable amount of research covering the evolution of database schemas [63], on a wide variety of topics and techniques. In this section, we focus on work related to the two primary contributions of this chapter, namely user interface refactoring and propagation of schema evolution through complex mappings.

Model management [6, 7] is an algebra of operators that take as input instances of data models (i.e., schemas) and mappings between model instances. Example operations in the model management algebra are:

- Match: takes two schemas and returns a mapping between them

- Compose: takes mappings from schema A to schema B and from schema B to

schema C and transitively combines them into a mapping from A to C

- Diff: takes a schema A and a mapping from A to schema B and returns all of the parts of A that do not participate in the mapping

One of the papers on model management [6] frames the schema evolution question as the following: given a base schema S_1, a set of view schemas over the base schema V_1, and an evolved version of the base schema S_2, construct an evolved version of the view schema V_2 that respects the original mapping between S_1 and V_1. Using model management, one can express this process as a sequence of operators, including the three operators listed above. The model management operators are strictly syntactic; matching that requires any additional tools (e.g., ontologies, thesauri) or human expertise are not covered. So, in the Guava context, model management can automatically handle cases of adding or removing tables and columns, but cannot handle element-level operations, renaming, or complex refactorings. Model management also cannot handle mappings between data and schema, and thus cannot handle mappings involving such translations as horizontal partitioning or pivoting.

The PRIMA [58] and PRISM [17] projects, mentioned in the related work of Chapter 3, describe transformations between one version of a database schema and the next from a menu of atomic transformations, with an overlapping (but not equivalent or subsuming) expressive power as Guava. Both of these languages support the scenario where a user poses a query against one version of the schema, and the query is re-written to pull data from all versions of the schema; Guava can be useful in this scenario as well, since an upgrade channel can translate queries posed against one version of a database into an equivalent query against a different version.

The Both-as-View (BAV) mapping language [47] is unique among federated database mapping paradigms [42] in that it explicitly supports schema evolution of both the global schema and the various local schemas [48]. As mentioned in Chapter 3, BAV mappings are expressed as a sequence of discrete transforms. In BAV, incremental changes to either global or local schemas are described as insertion, renaming, or deletion of individual schema elements, similar to Guava. Furthermore, specific changes to a concrete schema are grouped into two categories: those changes that can be expressed as an additional transform to be either appended or prepended to the existing list, and those that require changes to the existing transforms. In both cases, the developer must either specify a new query or modify an existing query to describe how data for the new schema element interacts with existing data from other schema elements. One cannot specify changes at the domain level in BAV.

Refactoring is used in other phases of software development. There is research on refactoring code [25], defined as incremental modification of code while leaving the external interface and behavior invariant. There is also work on refactoring databases [3], defined as incremental modification of a database leaving its behavior and information semantics invariant. However, there is little if any work done to formally investigate refactoring of user interfaces, and we are not aware of other tools that perform this functionality.

Contemporary web application frameworks [67, 73] allow a developer to specify the necessary steps to migrate from one version of a database to another. For instance, a Ruby on Rails migration is a sequence of DDL and DML statements that describe how to upgrade a database from one version to the next, and a corresponding sequence of

DDL and DML statements that describe how to downgrade back to the original version. Applications running on Ruby on Rails can then automatically apply these migrations as necessary to ensure that it is always operating on the correct database version. Unlike Guava, a database developer must specify the content of these migrations manually in both directions.

The approach that we take assumes that the developer creates a user interface using a graphical form builder. An alternative approach to building user interfaces is to use a declarative language, such as HTML, XAML [84] or XUL [86]. If the developer writes the UI code manually rather that using a user interface building tool, there is no way to automatically capture changes to the UI as they happen. However, one can compare two documents written using the same language and break down the differences in terms of atomic units. For instance, Eder and Wiggisser [21] present an algorithm for breaking down changes between two directed acyclic graphs into smaller parts; this algorithm would be applicable because user interfaces specified in UI languages are hierarchical and thus DAG's. Once an algorithm generates a list of atomic changes, pushing it through the channel proceeds as normal. Comparison tools would be unable to support complex changes such as UI refactorings without some extension to recognize patterns that may be joined into a single action.

6.5 SUMMARY AND IMPLEMENTATION STATUS

Manual creation of database upgrade scripts between versions of software is error-prone and requires testing. For applications that use Guava, upgrades to the database instance can be automatically generated. Database upgrade scripts in Guava come in two parts:

Changes to the user interface result in a transaction of statements that are pushed through the channel, and changes to the channel result in an upgrade channel through which the physical database is pushed. We analyzed the changes that are being made to a publicly-available software product and verified that the types of changes that they are making to the data handling in their application can be described using our upgrade framework.

Two of the features described in this chapter have been implemented in our prototype system. First, we have implemented the version control log (VCL). Every time a Guava application launches, it finds a marker in the database containing the version number of the database and finds that version number in the VCL. The application then compiles all of the changes from that point forward in the VCL (the AVCL) into a transaction and pushes it through the channel. Then it updates the version number in the database. Second, we have implemented functionality that can invert a portion of a channel; given a transformation in the channel, we can generate the inverse of the portion of the channel from the given transformation through to the physical database.

Chapter 7

CONCLUSIONS AND FUTURE WORK

This dissertation has presented a method for translating a user interface into a relational schema, and a tool for mapping relational schemas based on business logic and physical database design decisions. These two tools put together provide a way to construct database-backed applications in a user-interface-centric fashion. We described the status of our prototype implementation of Guava tools, and used the software produced by CORI as a case study to demonstrate the efficacy of our tools.

With respect to the research goals presented in Chapter 1, this dissertation makes the following contributions:

Research Goal 1: Develop an automatic method for constructing query interfaces that presents the conceptual model inherent in the user interface and draws usability features from the user interface, and is complete with respect to that conceptual model.

Contributions (Chapter 2):

- We described and formalized a method for constructing a relational schema from the specification for a user interface.

- We described and formalized a method for constructing a query interface that resembles the original user interface and produces queries against the data model referred to in the previous item.

- We created a prototype implementation of the relational schema and query interface generator that constructs those tools from any application running on our graphical widget library.

- We conducted a case study demonstrating that a commercially available software application can be implemented using our prototype implementation.

Chapter 2 also presented a contribution not directly related to this research goal, where the framework that generates a relational schema from a user interface can also be used to provide an addressing scheme for the user interface to uniquely identify data elements in context in an application.

Research Goal 2: Develop an information-preserving schema mapping language that is expressive enough to handle physical design decisions, and whose operational capabilities include transforming queries, data updates, and schema updates.

Contributions (Chapter 3):

- We introduced a generalized relational model that includes new relational algebra operators, schema evolution at the individual element level, and a generalized notion of referential integrity that allows incremental evolution.

- We defined and formalized an algebraic abstraction called a channel transformation that serves as a mapping between relational schemas that can translate queries, DML updates, DDL updates, and referential integrity constraints expressed in one schema into equivalent statements in the other schema.

- We defined and formalized the conditions that must be satisfied for a channel transformation to be information-preserving.

- We defined seven transformations that correlate to physical-design decisions commonly made in both physical database tuning solutions and generally available software.

- We conducted a case study demonstrating that the middleware of a commercially available software application can be implemented using a set of discrete transformations, and that all of those transformations that are motivated by physical design are covered by our set of seven transformations.

- We created a prototype implementation of these seven transformations, and demonstrated that this implementation is performant relative to an existing software middleware.

Chapter 3 also presented contributions not directly related to this research goal:

- We presented commutativity and invertibility rules to describe how to rearrange channel transformations.

- We defined how each of the seven physical-design transformations estimates the physical characteristics of its output schema given characteristics of its input schema.

- We introduced the copy insert method for propagating data from the conceptual schema to the physical schema or vice versa using only insert statement translation.

Research Goal 3: Demonstrate that our mapping language is extensible.

Contributions (Chapter 4):

- We defined new transformations that generalize three physical-design transformations from Chapter 3.

- We defined application-specific transformations that encapsulate business-logic decisions, including three transformations motivated by a case study in Chapter 3.

- We defined new transformations that correspond to the notion of correspondence assertions from the information integration research literature.

Chapter 4 also presented an additional contribution not directly related to this goal: We defined a mechanism called a change spike that allows transformations to communicate information about the operation of the transformation back to a channel's input schema.

Research Goal 4: Build a formal framework within which we can prove properties of our mapping language.

Contributions (Chapter 5):

- We provided mathematical proofs of correctness for our definitions of physical-design transformations relative to alternative definitions of those transformations on instances of data rather than operations.

- We provided sample mathematical proofs of correctness for application-specific transformations, where instance-at-a-time semantics may not be available or convenient.

Research Goal 5: Develop a scheme for handling application evolution, both of the user interface of the application and of the mapping between the user interface and database, such that the resulting database upgrade script is automatically generated.

Contributions (Chapter 6):

- We defined a framework for propagating changes to a user interface specification into an automatically generated script of data and schema upgrade statements.

- We introduced the notion of a user interface refactoring that encapsulates high-level transformations of a user interface, and showed examples of how form designers can support user interface refactoring.

- We defined a framework for propagating changes to a channel into an automatically generated script that translates database instances to use the new version of the channel.

- We carried out a case study that used our framework to characterize the changes made to a commercially available software application.

For the rest of this chapter, we articulate directions for future research in Guava.

7.1 ALTERNATIVE DATA MODELS: GUAVA GUI TOOLS

Our current conception of Guava is that the natural schema and physical database are in the relational model (with the appropriate foreign key extensions). We chose the relational model not because it was required, but rather for practical reasons. First, relational databases are by far the most common choice for data persistence in business applications currently. Also, most programming environments come with API's to work with relational data, either through cursors or through in-memory data structures such as the .Net DataSet object. However, one may want to consider using other data models for these artifacts.

For example, one may prefer to have the artifacts in the XML data model. Since a g-tree is already hierarchical in nature, it is relatively simple to translate a g-tree into an XML schema; therefore, it makes sense to express the natural schema as XML. One may also want to use XML as the physical storage medium. There are several native XML databases under development. In addition, all of the major vendors of relational databases systems support XML, either as a column type (allowing one to store XML in a field within a relational table) or as a collection of documents (allowing one to store XML outside of any relational table).

There are other models that a developer may consider. For instance, one obvious option for object-oriented programming languages is the object-oriented model. Yet another option is RDF (Resource Description Framework), in which all data is stored as subject-predicate-object triples.

To use a data model other than the relational model as the natural schema in Guava, we require the following:

- A data structure that can hold data in the target data model, to act as the natural schema and as a cache for in-memory data

- An algorithm that translates a g-tree into a schema in the target data model

- An algorithm that translates queries expressed in the Guava query interface into queries in the target data model (e.g., XQuery in XML or SPARQL in RDF)
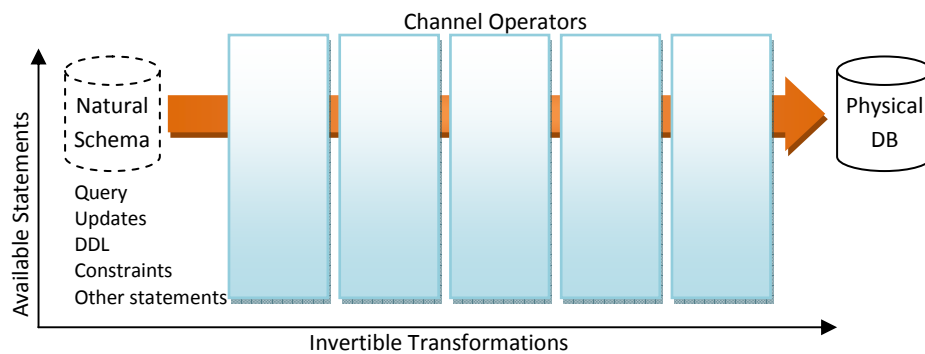
Figure 7.1: An alternative view of a channel, isolating the components necessary to build one independent of the underlying data model

7.2 ALTERNATIVE DATA MODELS: CHANNEL

The channel and transformations as defined in Chapter 3 also use our extended relational model. We chose the relational model for the channel for the same reason that we chose the relational model for the Guava GUI tools, namely that SQL and the relational model are the most prevalent tools used to construct forms-based applications. However, more and more applications are moving in part or in full to a web-based environment, where XML data is more prevalent. Therefore, we consider here how to construct a channel that can operate on a non-relational data model.

There are two primary considerations when designing a channel in a new data model, shown in Figure 7.1 as two orthogonal axes. On one axis is the set of invertible transformations that one can place in the channel. On the other axis is the language of statements that one can issue against the natural schema and that the channel understands. We now treat these axes individually.

On the transformation axis, one must identify transformations of data in the intended

data model that are invertible. Those transformations do not necessarily correspond to the the seven relational transformations we define in this dissertation, though each of those operators may have an analog in another model. For instance, the unpivot transformation in the relational model corresponds roughly to an edge mapping in XML, where the schema of an XML tree becomes data. Just as we describe in Chapter 4 for the relational model, a developer can create application-specific transformations, but just as with the relational model, there may be a small set of well-known transformations in common use that one can pre-define and formalize. In the case of XML, for instance, the following transformations are invertible and have clear motivations in terms of usage:

- *Invert Hierarchy*, in which a tree or forest is restructured so that a different element serves as the root

- *Promote Attribute*, in which an attribute becomes a child element

- *Demote Attribute*, in which an child element with atomic domain, no children, and single cardinality becomes an attribute

- *Partition*, in which a single tree is broken into multiple trees according to specified criteria

One final note about channel transformations and alternative data models: In addition to considering channel transformations whose input and output models are the same, we could also consider transformations from one model to another. One such transformation is an edge mapping, which is the lossless transformation of XML data into a single relation that stores one row per edge in the original XML document.

On the statement axis, one must provide analogs for each of the various classes of statements introduced in Chapter 3. The *Loop* and *Error* statements are recursively defined in terms of other statements, so one must find model-specific instances of the other classes (query, DML, DDL, and constraints). For the XML data model, some of these classes are easier to find analogs for than others.

Queries: The XML model supports several query languages, including XSLT, XPath, and XQuery. XPath is not likely to be powerful enough to remain closed under channel transformation; in other words, some transformations in the channel may be so complex that an input query expressed in XPath may not have an equivalent expression in XPath on the output. Since XQuery is a Turing-complete language, it is likely to be closed under XML channel transformation.

DML Updates: There is not yet a standard data update language for XML, but there are several options provide by both research and industry. For instance, SQL Server 2005 provides an update language for XML that can insert or remove nodes or change atomic data values at a location identified by an XPath expression. Given a set of XML channel transformations, one would need to verify that the update language is closed under those transforms.

DDL Updates: There is a standard schema-definition language, XML Schema, that one can use to establish a schema. There is not yet a standard data-definition language for incremental schema updates in XML, though at least one research project has presented a sound and complete list of primitive updates to XML schemas [27].

The difficulty of schema evolution in XML is that, unlike the relational model, the schema of an XML document does not necessarily have any effect on the storage of

that document. In relational storage, the tuples are stored in a fashion that conforms to the schema, so schema changes to the relational model are tightly coupled to atomic changes in storage structure; in XML, documents are validated against a schema at insertion time. Also, a single XML document can successfully validate against multiple XML schemas. So, the common scenario for XML is that if the schema changes, one issues a completely new schema and validates the documents against the new schema.

In addition, XML schemas are far more expressive than relational schemas; therefore, the range of possible schema changes is much greater. Some schema changes have clear similarities between the relational and XML models. For instance, an "Add Column" statement in the relational model is very similar to an "Add Child Element" in XML (or would be, if such a language existed). However, some possible schema changes in XML have no relational analog, such as "Change Order of Elements" and "Change Cardinality".

Even without a formal schema evolution language, we can emulate one by describing each incremental evolution as a pair of XSLT or XML DML operations: one that transforms the XML Schema document, and one that transforms any XML document that conforms to the old schema into one that conforms to the new one. So, if one develops an internal language that can represent XML Schema evolution primitives, the XML provider may be able to translate those primitives into XSLT statements.

Constraints: XML schemas have an ID-based system of foreign keys, similar to pointers in object-oriented programming. Primary keys (or more specifically, uniqueness constraints) can be expressed as an XPath expression that must always return unique results. It is still an open problem whether these constructs would need to be

generalized to be processed by a channel, and if so, what level of generalization is appropriate. For instance, an XPath expression representing a primary key may be translated by a channel into an XQuery expression that is not strictly XPath. However, similar to pushing foreign keys through relational channels, there may be an intermediate, "Tier 2" query language less expressive than full XQuery that is useful in this situation.

7.3 DEFINING NEW APPLICATION-SPECIFIC TRANSFORMATIONS

The three application-specific transformations presented in Chapter 4 are the ones required to meet the needs of CORI. Other applications will have different requirements for business logic, and may require additional transformations beyond these three. Other applications may also require versions of Adorn, Lookup, or Audit with different algorithms; for instance, an application may require a version of the Adorn transformation where the conditions for refreshing the adorned values are specified as Rule Interchange Format rules [68]. Or, an application may require a version of Audit with different transformation characteristics. Or, an application may require a transformation that cannot be constructed by any combination of the existing transformations presented here (for instance, the Normalize transformation required by InfoSonde, or the CORI transformation FindingMerge).

Fundamentally, a developer can write a new transformation by subclassing the class `Transform`; as demonstrated in the implementation section of Chapter 3, all of our implementations of transformations are written in this fashion. However, for each of the transformations presented in this dissertation, we describe the transformation formally, which allows us to prove that each transformation is information-preserving. We do

not expect the developer to formally describe every new transformation and prove the information-preservation properties. However, it is possible to empirically provide evidence that a channel is information-preserving with respect to a particular schema S. We call this process *channel certification*.

Consider the following sequence of transactions: First, generate the transaction of Guava statements that creates schema S. Then, for each table T in the schema S, generate a sequence of transactions, one transaction per line in the following list:

- Insert statements that create data for the table T

- Insert statements that create data that conflict with existing rows in T (which should fail, due to primary key violation)

- A Rename Table statement, renaming table T to T'

- Update statements that alter data from T'

- Delete statements that remove data from table T' (but not so many deletes as to empty the table)

- A Rename Table statement, renaming the table back to T

- Add, Rename, and Drop Column statements from table T

- Add, Rename, and Drop Element statements from table T

Each transaction should contain a suitably large number of statements that cover the domains of each attribute and in many combinations. After each transaction in the list, issue a query transaction, querying each table in the schema S individually. The query

transaction verifies that the table T (or T', midway through the list) has been altered in the necessary way, and that no other tables have. If there are any foreign keys (first or second tier) defined with table T as the source, then some added steps occur just *before* the previous list of transactions:

- For each foreign key, attempt to create rows in T that satisfy the head of the foreign key but have no counterpart in the target of the foreign key. The attempt should fail.

- For each foreign key, create rows in the target table that satisfy the tail of the foreign key (so that rows can be safely added to T without throwing any more errors).

If a channel passes the tests above, then we say the channel is *minimally certified* with respect to S. We would expect that Guava would automatically generate all of these transactions based upon the input schema of the transformation. The tests empirically demonstrate that the information-preservation properties hold by generating a sample workload and verifying that, at all times, the contents of all tables in S contain exactly the same data as if there were no channel and the natural schema exactly matched the physical schema.

The certification is "minimal" because the set of above tests does not cover every single possible combination of statements. It is possible that a custom-built transformation has some bizarre side effect that does not preserve information for statements with particular values. Since we cannot test every single possible insert statement against S, for instance, the transformation drops statements that have negative values in some

columns. We may be able to improve the coverage of a channel certification by allowing a quality assurance expert to define data distributions for each possible statement to cover potentially known interesting cases.

We can empirically analyze the effect of a transformation on physical characteristics. Given statistics on row counts and value distributions over a transformation's input schema, we can generate a database with a substantial amount of data that matches those characteristics, in a similar fashion to the sample data generator that comes as part of Microsoft Visual Studio Team System Database Edition [54]. We can then push that data through the transformation and measure statistics over the output.

Finally, we can also empirically certify inverse relationships. If the developer specifies that the inverse of a newly-developed transformation T is the sequence of transformations $[T_1, T_2, \ldots, T_n]$ (where the transformations T_i need not themselves be information-preserving, as in the case of Audit), we can empirically demonstrate through sample workloads that $[T, T_1, T_2, \ldots, T_n]$ is equivalent to the empty channel ϵ.

7.4 BEYOND A SINGLE APPLICATION ENVIRONMENT

In this dissertation, we have described an architecture for building software with a single GUI operating over a single data source. Here, we briefly consider directions for using Guava to handle multiple applications accessing a single data source. There is a vast amount of research in the field of information integration that covers the general case of integrating multiple data sources (including Interschema Correspondence Assertions [72], which were the inspiration for correspondence assertion transformations in the channel).
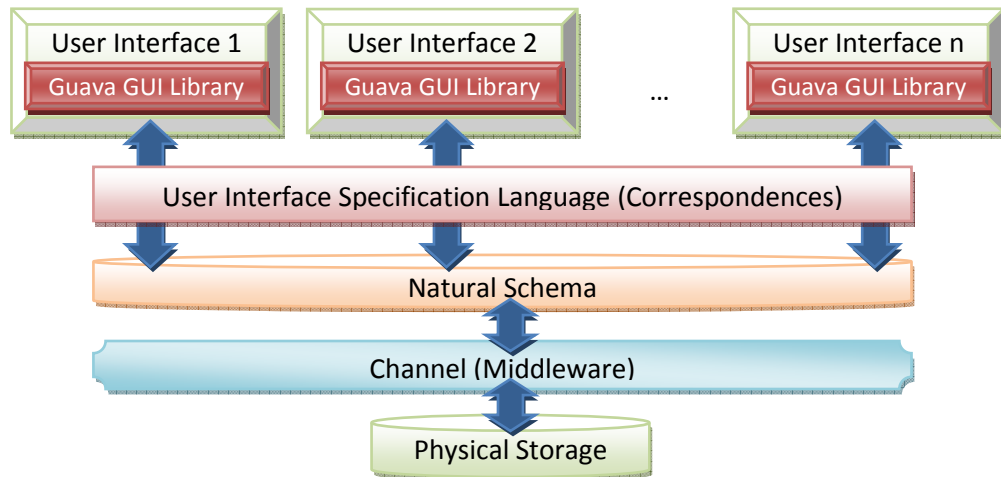
(a) Multiple applications on a single natural schema
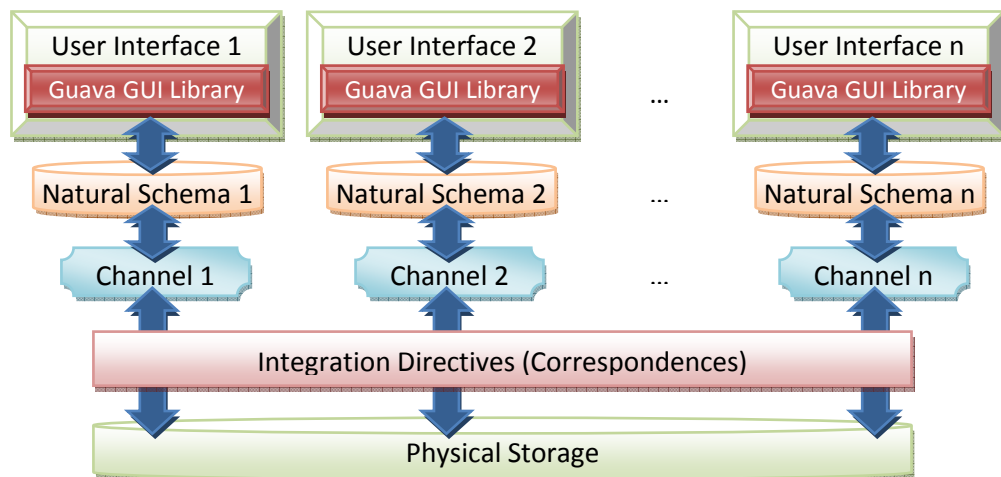
(b) Multiple applications on multiple natural schemas

Figure 7.2: Two different scenarios for using Guava where multiple applications access the same physical database

Figure 7.2 shows two potential ways to handle the multiple application scenario; the primary difference between the two situations is which actor handles the correspondence assertions. In Figure 7.2(a), the many applications share a single natural schema; correspondence assertions are specified by the user interface designer (possibly as an extension to the form builder, like the user interface refactorings). The channel transformations that correspond to the user interface designer's directives appear at the beginning of the channel. This way of specifying correspondences is appropriate for situations where the multiple applications are developed in the same place using the same tools, since specifying a correspondence between heterogeneous environments may require significant additions to each form building application.

In Figure 7.2(b), the many applications each have their own natural schema and channel, and the database designer introduces correspondence assertions to unify their physical schemas. This situation is like the classical information integration problem, where all applications are developed independently and their physical database needs to be unified after the fact.

In the first situation, where correspondence assertions can be specified as directives in a form builder, there is the additional question of how those assertions can be specified, and how they appear to the UI developer in the builder once specified. In both of these situations, the open question is how (or, for that matter, if) one can describe inter-schema correspondences (such as the ICA's from Spaccapietra [72]) beyond Column and Table Equate as channel transformations, including relationships such as inclusion, overlap, or mutual exclusion.

7.5 BEYOND A SINGLE DEVELOPER ENVIRONMENT

The VCL introduced in Chapter 6 works as designed in a single-developer environment. When in a multiple-developer environment, some aspects of the VCL need to be changed:

- The concept of "version" carries a different meaning in a multiple-developer environment, since different people constructing builds simultaneously will have different concepts of what the current build number is

- Since the VCL is itself an artifact in source control, different developers' VCLs will need to be reconciled when checking in source code

The issue of multiple build numbers can be resolved in a number of ways; for instance, we can change the idea of build number so that it is a vector, representing the current build with respect to each developer. Or, the build number in each checkpoint entry may be independently managed by each developer, but the primary version number *X.Y.Z* is managed centrally within source code.

With respect to merging VCL lists, some merging can happen automatically if the artifacts in the VCL do not conflict. The following situations would cause conflicts:

- Entry A from one VCL and entry B from another VCL yield one final result if A comes before B but yield a different result if B comes before A

- Having both entries C and D in a VCL at the same time would cause an error, whereas each individually would not cause an error (for instance, if both entries are Add Column statements adding a column with the same name)

We would expect that, in these cases, conflicts would be resolved in a similar fashion to how conflicts are resolved when code is being merged in source control: a developer is presented with the code that is causing a conflict, and the developer is tasked with making a judgement about how to resolve the conflict. Some conflicts in a VCL merge, however, are the direct result of a conflict in the code; for example, if a VCL conflict arises from two different statements trying to add columns with the same name, that is a direct result of two different controls with the same name trying to be added to the same form, which will cause a build error. Thus, fixing the error in the code would effectively fix the merge problem. The open question here is if conflicts at the VCL level can always be mapped to conflicts at the GUI level, and if so, whether fixing the GUI conflict simultaneously fixes the VCL conflict.

7.6 ADDITIONAL FUTURE WORK

Tables 3.10 and 4.5 describe what effects channel transformations have on table statistics. Also, we introduce commutativity and invertibility rules that one can use to translate channels into equivalent channels. One open question is whether it is possible to construct an automated channel generator and optimizer — given a natural schema and a workload, construct a channel (or, more specifically, the part of the channel pertaining to physical design) that is either optimal for time or for space or both. This process would need to be coupled with a physical design tool [1] to leverage existing work on generating physical structures such as indexes and materialized views. An alternative option would be to take an existing physical design tool and extend its capabilities to be able to work with all of the transformations that the channel can offer.

In Chapter 6, we introduced the comparison approach to translating changes to a channel into changes to a database instance. Just as in the user interface changes, if we assume that there is a tool available to assist the developer with building a channel, that tool may be able to recognize incremental changes to the channel. In the *incremental approach* to channel evolution, an atomic change to a channel translates into a transaction of DML and DDL updates that are pushed through a portion of the channel. For instance, if a VPartition transformation is created at the sixth position from the end of a channel, a set of DML and DDL statements are pushed through only the last five transformations in the channel.

In the incremental approach, each channel transformation must include descriptions for how to translate each of the following actions into DML and DDL:

- Insert Transformation: describe the DML and DDL that is generated when a transformation is inserted into a channel

- Delete Transformation: describe the DML and DDL that is generated when a transformation is deleted from a channel

- Modify Transformation Parameters: describe the DML and DDL that is generated when the parameters for a transformation are altered by the developer; note that this means the transformation must know how to handle modifications to any of its parameters, including parameters that are lists and parameters that are functions

It is an open question as to the best way to integrate incremental changes of the GUI with incremental changes to the channel. One possibility is that they may both somehow be encoded in the VCL simultaneously.

There are emerging benchmarks in the field of schema mappings and matching [2]. It would be informative to compare a channel against other mapping languages and techniques using these benchmarks, once they have been refined and gained more acceptance.

Currently, the only constraints that we have defined in our data model (and, therefore, that our transformations know how to operate upon) are foreign keys and primary keys. We may be able to extend the model and each transformation to include other constraints, including functional dependencies or, more generally, conditional functional dependencies [23].

Finally, we have some hypotheses that we would like to test empirically with regard to users. Because a Guava query interface derives directly from the user interface and because it connects to the database through the same mechanisms as the user interface, we believe that the Guava query interface can give domain experts both productivity and quality gains in their queries. It is an open question whether the Guava query interface can offer these gains; one way to evaluate this hypothesis would be to conduct user studies.

REFERENCES

[1] S. Agrawal, V. Narasayya, and B. Yang. Integrating Veritcal and Horizontal Partitioning into Automatic Physical Database Design. *SIGMOD 2004*, June 2004.

[2] B. Alexe, W. Tan, and Y. Velegrakis. STBenchmark: Towards a Benchmark for Mapping Systems. *VLDB 2008*, Auckland, New Zealand, August 25–28, 2008, 230–244.

[3] S. W. Ambler and P. J. Sadalage. *Refactoring Databases*. Addison-Wesley, publisher. ©2006.

[4] M. Angelaccio, T. Catarci, and G. Santucci. QBD*: A Graphical Query Language with Recursion. *IEEE Transactions on Software Engineering*, October, 1990, 16(10):1150–1163.

[5] F. Bancilhon and N. Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems*, December 1981, 6(4):557–575.

[6] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. *CIDR 2003*, Asilomar, California, January 5–8, 2003, 209–220.

[7] P. A. Bernstein, S. Melnik. Model management 2.0: Manipulating Richer Mappings. *SIGMOD 2007*, Beijing, China, June 12–14, 2007, 1–12.

[8] J. A. Blakeley, D. Campbell, S. Muralidhar, and A. Nori. The ADO.Net Entity Framework: Making the Conceptual Level Real. *SIGMOD Record*, December 2006, 35(4):31–38.

[9] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. *PODS '06*, Chicago, IL, USA, 2006, 338–347.

[10] N. Bruno and S. Chaudhuri. Automatic Physical Database Tuning: A Relaxation Approach. *SIGMOD 2005*, June 2005, 227–238.

[11] Centricity EMR, formerly Logician. http://support.medicalogic.com/.

[12] S. Chaudhuri, V. R. Narasayya, and M. Syamala. Bridging the Application and DBMS Profiling Divide for Database Application Developers. In *VLDB 2007*, Vienna, Austria, September 23-27, 2007, 1252–1262.

[13] Clinical Outcomes Research Initiative. http://www.cori.org/.

[14] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 1970, 13(6):377-387.

[15] Crystal Reports. http://www.businessobjects.com/product/catalog/crystalreports/.

[16] C. Cunningham, G. Graefe and C. A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. *VLDB 2004*, 998–1009.

[17] C. Curino, H. Moon, and C. Zaniolo. Graceful Database Schema Evolution: the PRISM Workbench. *VLDB 2008*, Auckland, New Zealand, August 25–28, 2008, 761–772.

[18] B. Dageville, D. Das, and K. Dias. Automatic SQL Tuning in Oracle 10g. *VLDB 2004*, 1098–1109.

[19] U. Dayal and P. Bernstein. On the Correct Translation of Update Operations on Relational Views. *ACM Transactions on Database Systems*, September 1982, 8(3):381–416.

[20] D. Draheim and G. Weber. *Form-Oriented Analysis*. Springer-Verlag, publisher. ©2005.

[21] J. Eder, K. Wiggisser. A DAG Comparison Algorithm and Its Application to Temporal Data Warehousing. *ER Workshops 2006 (ECDM 2006)*, 217–226.

[22] D. W. Embley. NFQL: the natural forms query language. *ACM Transactions on Database Systems*, June 1989, 14(2):168–211.

[23] W. Fan et al. Propagating Functional Dependencies with Conditions. *VLDB 2008*, Auckland, New Zealand, August 25–28, 2008, 391–407.

[24] S. Flesca, F. Furfaro and S. Greco. XGL: a graphical query language for XML. *Proceedings of the International Database Engineering and Applications Symposium (IDEAS02)*, Washington, DC, USA, 2002, 86–95.

[25] M. Fowler et al. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, publisher. ©1999.

[26] E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, publisher. ©1994.

[27] G. Guerrini, M. Mesiti, and D. Rossi. Impact of XML Schema Evolution on Valid Documents. *Proceedings of the International Workshop on Web Information and Data Management (WIDM), CIKM 2005 Workshop*, Bremen, Germany, November 5, 2005, 39–44.

[28] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 1995, 18(2):3-18.

[29] A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 2001, 10(4):270–294.

[30] J-L. Hainaut. The Transformational Approach to Database Engineering. *Generative and Transformational Tech. in Software Engineering*, 2006, LNCS 4143:89–138.

[31] D. C. Hay. *Data Model Patterns: A Metadata Map*. Morgan Kaufman, publisher. ©2006

[32] M. Hernandez, P. Papotti, and W. Tan. Data Exchange with Data-Metadata Translations. *VLDB 2008*, Auckland, New Zealand, August 25–28, 2008, 260–273.

[33] Hibernate. http://www.hibernate.org/.

[34] B. Howe et al. Quarrying dataspaces: Schemaless profiling of unfamiliar information sources. *ICDE Workshops 2008*, Cancun, Mexico, 270–277.

[35] IBM FileNet Forms Manager. http://www-306.ibm.com/software/data/content-management/filenet-forms-manager/.

[36] IBM Lotus Forms. http://www-306.ibm.com/software/lotus/forms/.

[37] H. V. Jagadish et al. Making Database Systems Usable. *SIGMOD 2007*, Beijing, China, 2007, 13–24.

[38] M. Jayapandian, H. V. Jagadish. Automated Creation of a Forms-based Database Query Interface. *VLDB 2008*, Auckland, New Zealand, August 25–28, 2008, 695–709.

[39] A.M. Keller, R. Jensen, and S. Agrawal. Persistence Software: Bridging Object-Oriented Programming and Relational Databases. *SIGMOD 1993*, 523–528

[40] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On efficiently implementing SchemaSQL on a SQL database system. *VLDB 99*, Edinburg, Scotland, September 1999, 471–482.

[41] J. A. Larson, S. B. Navathe, and R. Elmasri. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. *IEEE Transactions on Software Engineering*, April 1989, 15(4):449–463.

[42] M. Lenzerini. Data Integration: A Theoretical Perspective. *PODS 2002*, 233–246.

[43] J. Logan, J. F. Terwilliger, and L. M. L. Delcambre. Exploiting the User Interface for Tomorrow's Clinical Data Analysis. *Journal on Information Technology in Healthcare*, April 2008, 6(2):138-149. Reprinted from Todays Information for Tomorrows Improvements 2007, an international conference addressing Information Technology and Communications in Health (ITCH 2007).

[44] Logician Data Schema, Release 5. MedicaLogic Inc. (now GE Medical Systems) P/N 2487-04. ©1999.

[45] D. Lomet et al. Transaction Time Support Inside a Database Engine. *ICDE 2006*, Atlanta, Georgia, USA, 2006.

[46] A. Maule, W. Emmerich, and D. S. Rosenblum. Impact Analysis of Database Schema Changes. *ICSE 2008*, Leipzig, Germany, May 10–18, 2008.

[47] P. McBrien and A. Poulovassilis. Data Integration by Bi-Directional Schema Transformation Rules. *ICDE 2003*, 227–238.

[48] P. McBrien and A. Poulovassilis. Schema Evolution in Heterogeneous Database Architectures, a Schema Transformation Approach. *CAiSE 2002*, 2002, 484–499.

[49] S. Melnik, A. Adya, and P. A. Bernstein. Compiling Mappings to Bridge Applications and Databases. *SIGMOD 2007*, Beijing, China, 2007, 461–472.

[50] Microsoft Office Access. http://office.microsoft.com/en-us/access.

[51] Microsoft Office InfoPath. http://office.microsoft.com/en-us/infopath.

[52] Microsoft SQL Server Integration Services.
http://www.microsoft.com/sql/technologies/integration/default.mspx.

[53] Microsoft SQL Server 2005. http://www.microsoft.com/sql/default.mspx.

[54] Microsoft Visual Studio Team System Database Edition.
http://msdn.microsoft.com/en-us/vsts2008/db/default.aspx.

[55] R. J. Miller. Using Schematically Heterogeneous Structures. *SIGMOD 1998*, Seattle, WA, June 1998, 27(2):189–200.

[56] R. J. Miller, L. M. Haas, and M. A. Hernndez. Schema Mapping as Query Discovery. *VLDB 2000*, San Francisco, CA, USA, 2000, 77–88.

[57] R. J. Miller et al. The Clio Project: Managing Heterogeneity. *SIGMOD Record*, 2001, 30(1):78–83.

[58] H. Moon et al. Managing and Querying Transaction-time Databases under Schema Evolution. *VLDB 2008*, Auckland, New Zealand, August 25–28, 2008, 882–895.

[59] P. Mork, P. A. Bernstein, and S. Melnik. Teaching a Schema Translator to Produce O/R Views. *ER 2007*, Aukland, New Zealand, 2007, 102–119.

[60] S. Murthy, L. M. L. Delcambre, and D. Maier. Explicitly Representing Superimposed Information in a Conceptual Model. *ER 2006*, Tucson, Arizona, Nov. 6–9, 2006, 126–139.

[61] A. Papantonakis and P. J. H. King. Gql, a declarative graphical query language based on the functional data model. In *Proceedings of the workshop on Advanced visual interfaces (AVI 94), a SIGCHI workshop*, Bari, Italy, 1994, 113–122.

[62] A. Poulovassilis and P. McBrien. A General Formal Framework for Schema Transformation. *Journal of Data and Knowledge Engineering (DKE)*, October 1998, 28(1):47–71.

[63] E. Rahm and P. A. Bernstein. An Online Bibliography on Schema Evolution. *SIGMOD Record*, December 2006, 35(4):30–31.

[64] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001, 10(4):334–350.

[65] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. *VLDB 2001*, San Francisco, CA, USA, 2001, 381–390.

[66] S. R. Rollinson and S. A. Roberts. Formalizing the Informational Content of Database User Interfaces. In *Proceedings of the 17th International Conference on Conceptual Modeling (ER98)*, Singapore, November 16-19, 1998, 65–77.

[67] Ruby on Rails. http://www.rubyonrails.org/.

[68] Rule Interchange Format (RIF). http://www.w3.org/2005/rules/.

[69] SAS Metadata Server.
http://www.sas.com/technologies/bi/appdev/base/metadatasrv.html.

[70] E. Sciore, M. Siegel, and A. Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM Transactions on Database Systems*, June 1994, 19(2):254–290.

[71] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[72] S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB Journal*, 1992(1):81-126.

[73] SQL Alchemy. http://www.sqlalchemy.org/.

[74] J. F. Terwilliger, L. M. L. Delcambre, and J. Logan. Context-Sensitive Data Integration. In *Proceedings of the EDBT 2006 Workshop on Information Integration in Healthcare Applications (IIHA)*, Munich, Germany, March 26, 2006, 20–31.

[75] J. F. Terwilliger, L. M. L. Delcambre, and J. Logan. The User Interface is the Conceptual Model. *ER 2006*, Tucson, Arizona, USA, November 6–9, 2006, 424-436.

[76] J. F. Terwilliger, L. M. L. Delcambre, and J. Logan. Querying through a user interface. *Journal of Data and Knowledge Engineering (DKE)*, December 2007, 63(3):748–768.

[77] H. Tian et al. NeuroQL: A Domain-Specific Query Language for Neuroscience Data. In *Proceedings of the 2006 EDBT Workshops 2006, Workshop on Query Languages and Query Processing*, Munich, Germany, March 26–31, 2006, 613–624.

[78] D. Tsichritzis and A. C. Klug. ANSI/X3/SPARC DBMS Framework. Report of the study group on data base management systems, AFIPS Press, Arlington, Va., 1977.

[79] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. *The VLDB Journal*, April 1996, 5(2):101–118.

[80] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, and S. Skiadopoulos. A generic and customizable framework for the design of ETL scenarios. *Information Systems*, November 2005, 30(7):492–525.

[81] H. Wei and R. Elmasri. PMTV: A Schema Versioning Approach for Bi-Temporal Databases. *TIME 2000*, Washington, DC, USA, 2000, 143–151.

[82] C. M. Wyss and E. L. Robertson. A Formal Characterization of PIVOT/UNPIVOT. *CIKM 2005*, Bremen, Germany, October/November 2005, 602–608.

[83] C. M. Wyss and F. I. Wyss. Extending relational query optimization to dynamic schemas for information integration in multidatabases. *SIGMOD 2007*, Beijing, China, 2007, 473–484.

[84] XAML. http://www.xaml.net/.

[85] XForms. http://www.w3.org/TR/xforms/.

[86] XUL. http://www.xulplanet.com/.

[87] M. M. Zloof. QBE/OBE: A Language for Office and Business Automation. *IEEE Computer*, 1981, 14(5):13–22.