

# **Language Integrated Incremental Relational Lenses**

*Rudi Horn*

Master of Science by Research  
School of Informatics  
University of Edinburgh

2017



# Abstract

Existing work by Bohannon et al. show that it is possible to define lenses for relational data. The idea is that the user can make updates to a view, and the updated view is translated into updated underlying database tables. These relational lenses suffer from being inefficient on larger databases in traditional web server configurations. This is caused by the fact that entire database tables need to be recomputed, which is expensive and requires the entire tables to be copied to and from the server. The proposed solution is to incrementalize relational lenses, so that changes to a view can be tracked as a change set. This change set to the view is translated into a change set which is applicable to the underlying database. During translation, any information that is required but missing from the change set can be obtained by querying the database server.

We also implement our incremental relational lenses as a Links language extension. By offering the lenses as a language extension they become easy and intuitive to use. It also helps prevent security issues such as injection attacks and bugs by making use of type checking.

We evaluate the performance of our incremental lenses in comparison to a naive version. We also compare different types of changes and how they scale with different lens compositions. Our experiments show that incremental relational lenses are able to outperform our naively implemented relational lenses by a factor of up to 300 for the total execution time and up to 10 times for the time spent querying the database server.

# Acknowledgements

First of all I would like to thank my supervisor Dr. James Cheney, who took the time for weekly meetings and always gave great and helpful feedback.

I would like to thank Wen Kokke, Simon Fowler and Casey Beall for proof reading my thesis on short notice. All of you gave great feedback that helped a lot to improve my thesis.

The work here also would not have been possible without the Links project and so I would like to thank everyone who has worked on it so far. The same goes for the authors of the Relational Lenses paper, as the work presented here heavily relies on it.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Rudi Horn)



# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Contributions . . . . .	4
1.3	Structure of Thesis . . . . .	4
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Lenses . . . . .	7
2.2	Relational Databases . . . . .	8
2.3	Links Programming language . . . . .	9
2.4	Relational Lenses . . . . .	10
2.4.1	Helper Definitions . . . . .	10
2.4.2	Selection Lens . . . . .	12
2.4.3	Join Lens . . . . .	14
2.4.4	Projection Lens . . . . .	16
2.4.5	Composition Lens . . . . .	17
<b>3</b>	<b>Incremental Relational Lenses</b>	<b>19</b>
3.1	Overview . . . . .	19
3.2	Change Set Generation . . . . .	20
3.3	Invariant Properties . . . . .	21
3.3.1	No conflicting output . . . . .	21
3.3.2	Validity of changes . . . . .	22
3.3.3	Changes contain complements . . . . .	22
3.4	Notation . . . . .	23
3.4.1	Phrase Quasiquotation . . . . .	23
3.5	Helper Definitions . . . . .	24
3.5.1	Common Functions . . . . .	24

3.5.2	Sort Function . . . . .	24
3.5.3	Record Functions . . . . .	24
3.5.4	Update Set . . . . .	25
3.6	Select Lens . . . . .	26
3.7	Projection Lens . . . . .	30
3.8	Join Lens . . . . .	32
3.8.1	Notation and Assumptions . . . . .	33
3.8.2	Helper Definitions . . . . .	34
3.8.3	Case Analysis . . . . .	43
3.8.4	Coverage . . . . .	51
3.9	Put Function . . . . .	52
3.9.1	Table Put . . . . .	52
3.9.2	Combined Put . . . . .	54
3.10	Discussion on Correctness . . . . .	55
<b>4</b>	<b>Language Integration</b>	<b>57</b>
4.1	Types . . . . .	57
4.2	Values . . . . .	58
4.3	Syntax Rules . . . . .	58
4.4	Context . . . . .	59
4.5	Evaluation . . . . .	59
4.6	Typing Rules . . . . .	61
4.7	Links Example . . . . .	63
<b>5</b>	<b>Evaluation</b>	<b>67</b>
5.1	Number of Input Rows . . . . .	67
5.1.1	Setup . . . . .	67
5.1.2	Analysis . . . . .	68
5.2	Number of Tables . . . . .	70
5.2.1	Setup . . . . .	70
5.2.2	Analysis . . . . .	70
<b>6</b>	<b>Related Work</b>	<b>73</b>
6.1	Edit Lenses . . . . .	73
6.2	Incremental View Maintenance . . . . .	74
6.3	Language Integrated Query . . . . .	75



<b>7 Conclusion</b>	<b>79</b>
7.1 Problem Summary . . . . .	79
7.2 Proposed Solution . . . . .	80
7.3 Results . . . . .	80
7.4 Future Work . . . . .	81
<b>Bibliography</b>	<b>83</b>



# Chapter 1

## Introduction

### 1.1 Motivation

Relational databases continue to be one of the most used storage back ends for many different types of applications, especially including web servers. These databases offer the advantage that they are highly performant while being easy to use and flexible enough for a range of applications. Most databases allow interaction using a standardized language called Structured Query Language (SQL).

The most common operations which can be used to query databases are projections, selections and joins. A projection takes a view and removes unwanted columns, e.g.  $\pi_{A,B}(S)$  in relational algebra, takes the source table  $S$  and drops all columns except for  $A$  and  $B$ . Selections filter out rows not satisfying a specified predicate and can be used to only return desired rows. An example of a selection is  $\sigma_{B=3}(S)$  which takes the table  $S$  and removes all rows in  $S$  which do not have the value of column  $B$  equal to 3. Finally join operations take two tables and combine all rows which have the same values in certain columns. A special case is the natural join, which joins all rows which have the same values in common columns, e.g.  $R \bowtie S$ .

A view is a modification of one or more input tables to produce an output table and is specified as a combination of different relational operations. Views are used to calculate data in such a way that the data can be used for different scenarios in a given application. Being able to calculate views is a useful concept that helps databases to be efficient. While the forward direction is trivial to calculate

and can be performed by the database server, tracking the changes back to the underlying database is a more difficult problem that is also referred to as the view update problem [1, 2]. Finding a method to work with updatable views would have multiple benefits though, including:

1. **Confidentiality:** Views allow the hiding of sensitive information such as rows belonging to other users or columns that are not supposed to be shown to the user.
2. **Security:** The view prevents changes out of the scope of the view, including different tables or rows that are filtered out by the view. Existing work also specifically designs lenses with the purpose of ensuring security [3].
3. **Ease of use:** The programmer only has to write the code to determine how the view is calculated and does not have to manage the minor details of how updates are applied to the database.

The concept of having updatable views is strongly related to the field of bidirectional transformations [4] and lenses [5]. In the case of relational data, lenses can be assumed to be asymmetric lenses [6], where the source data structure is the underlying database and the target structure is the view. Here lenses consist of a `get` function, which takes the underlying database and calculates the view, and a `put` operation which takes the modified view and the underlying database and produces the updated underlying database. Additionally there are two properties which a lens must satisfy in order for them to be *well-behaved*.

Existing work by Bohannon et al. [11] define a set of composable lenses for projections, selections and joins on relational data. The relational lenses are defined as functions which take the whole underlying database and an updated view and return the entire updated source data structure, while satisfying the well-behavedness properties.

The approach of recomputing the entire underlying data tables by naively following the definitions of relational lenses is infeasible in the traditional web server and SQL database setup. The issue is that web servers normally interface with larger databases, which are most likely located on different servers and difficult to manipulate directly. This means that in order to commit changes, it would be necessary to download the required tables, recompute them and replace them on the server. This process quickly becomes very slow for larger datasets.

The approach taken here, as shown in Figure 1.1, is to try and commit changes made to a view by calculating a change set for the underlying database, which can be applied using the SQL `INSERT`, `UPDATE` and `DELETE` commands. This is done by initially calculating a change set between the original view and a modified view and then tracking the changes through the different lenses. By using a change set it is now longer required to download and replace entire tables. Performing computations on change sets which are much smaller than full database tables reduces the amount of computation and makes it less problematic if algorithms are less efficient.

Determining the correct change set cannot be done without having access to the full database. It is however possible to access this information by querying the database server, which is able to perform these operations efficiently.

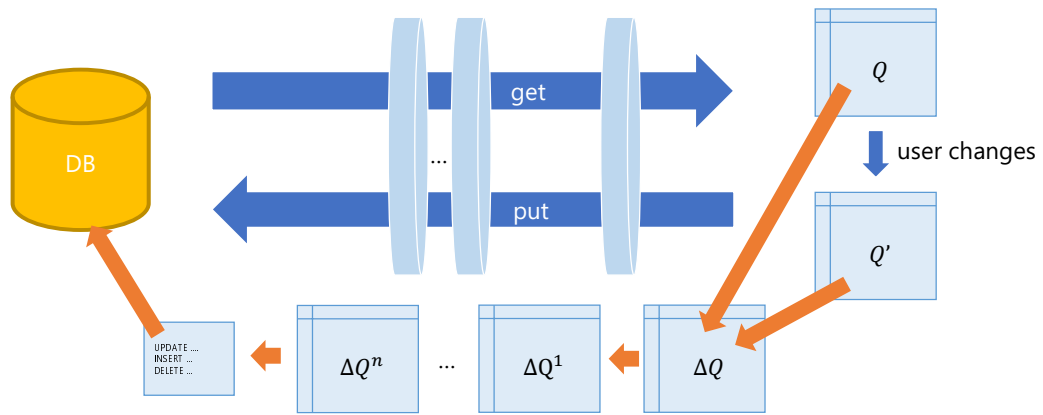


Figure 1.1: How changes to a view can be applied to a database.

One of the further challenges related to web programming is the abundance of languages required for implementing an application. Web applications use Hyper-text Markup Language (HTML) for the document layout, Cascading Style Sheets (CSS) for document styling, JavaScript for client side code, SQL for database communication as well as an additional programming language for server side code. The multitude of languages makes development difficult and one attempt to simplify this problem has been the programming language Links [7]. Links compiles code to JavaScript which can then be executed on the client. It also assists in generating the output page by integrating HTML document structure elements into the host language.

Links also includes functionality for querying database servers in the form of Language Integrated Query (LINQ). Instead of requiring the user to write SQL

queries, Links automatically generates SQL queries that are executed on the database server from Links code [8, 9]. LINQ is a type-safe and easy method to query database servers, without having to become familiar with SQL. These are methods similar to those used in Microsoft .NET's LINQ and Kleisli [20, 19].

We implement incremental relational lenses and show that they are more efficient than non incremental relational lenses. We also show how they can be integrated into Links.

## 1.2 Contributions

Our contribution consists primarily of a formalization of how to process view updates to relational lenses in an incremental fashion. This allows the relational lenses to be used in the traditional web server environment, which would normally be infeasible. There is also no realistic report of an implementation of relational lenses at the time of writing.

Additionally, this formalization has been implemented as a Links programming language extension. The language extension was then used to produce some examples, which allow the performance of the incrementalized version to be compared to the naive implementation.

Finally, using the benchmarks it is possible to show that there is a major performance benefit of using the incrementalized version over the naive implementation. The evaluation also shows that the relational lenses are usable in the traditional web server setup and allow the user to easily define lenses which can be updated efficiently.

## 1.3 Structure of Thesis

Chapter 2 describes existing work necessary to understand the background of this thesis. Section 2.1 contains more information on lenses and the definitions of well-behavedness. Section 2.2 gives short overview of required knowledge on relational databases. Section 2.3 explains the Links programming language in more detail. Section 2.4 describes in detail the existing work on relational lenses,

which form the foundation of the work presented here. Chapter 3 is about how relational lenses are incrementalized. Chapter 4 contains details of incremental relational lenses have been integrated into the Links programming language. Chapter 5 analyzes and discusses the performance of the implemented relational lenses. Further related work is discussed in Chapter 6. This includes work on edit lenses (Section 6.1), incremental view maintenance (Section 6.2) and LINQ (Section 6.3). Finally Chapter 7 draws a conclusion and discusses future work.





# Chapter 2

## Background

### 2.1 Lenses

Lenses define bidirectional transformations between two data structures  $X$  and  $Y$ . Lenses are typically determined by two functions, a **get** operation and a **put** operation. The user specifies the **get** operation, and lenses derive the **put** direction. The signatures of the two functions are usually defined as follows:

$$\text{get} : X \rightarrow Y$$

$$\text{put} : X \times Y \rightarrow X$$

In order to try and ensure correctness, the concept of well-behaved lenses are defined as lenses satisfying two properties. The first property, called **PutGet**, ensures that whatever data we put into a lens is returned unchanged if we try to get it again. The second property, called **GetPut**, ensures that if we put the unchanged output of **get**( $X$ ) into a lens,  $X$  is returned unchanged.

$$\text{get}(\text{put}(x, y)) = y \quad (\text{PutGet})$$

$$\text{put}(x, \text{get}(x)) = x \quad (\text{GetPut})$$

## 2.2 Relational Databases

Relational databases are an extensive topic and there are many things to consider which cannot all be covered here. In order to understand this dissertation it is necessary to understand the basics of what databases, tables and functional dependencies are. These are briefly covered in this section, but for further information it is recommended to consult additional reading material [10].

Relational databases consist out of one or more tables. Tables are sets of rows, which adhere to the same set of columns. An example of this is shown in Figure 2.1.

$R$	$A$	$B$	$S$	$B$	$C$	$D$
	2	1		1	1	5
	3	2		2	2	6
	4	2		3	2	7

Figure 2.1: Two examples of relational tables called  $R$  and  $S$ .

Functional dependencies are constraints between sets of columns. Functional dependencies are written in the form  $X \rightarrow Y$ , meaning that if two rows have the same value for all columns in  $X$  they also need the same values in all columns  $Y$ .

The example in Figure 2.1 assumes the functional dependencies  $A \rightarrow B$  and  $B \rightarrow CD$ . This means that having an additional row in table  $R$  with the values  $(A = 3, B = 4)$  would be invalid, since all rows with the value  $A = 3$  need to have  $B = 2$ . Therefore it is valid to have the second row  $(A = 4, B = 2)$  despite the row  $(A = 3, B = 2)$  being in the table. In the work presented here we describe the signature of the tables as  $R: (\underline{A}, B)$  and  $S: (\underline{B}, C, D)$ , where the underlined part indicates which columns have to define all other columns.

Databases are normally located on different machines than the application using them. As databases can become quite large, it is necessary to be able to retrieve specific subsets in order for them meet application needs efficiently. In practice this is done using the Structured Query Language (SQL). We will mainly use the notation of relational algebra for querying databases, which for the purposes described here can easily be converted into SQL. Specifically, three commonly

used operations which are defined as lenses will be explained here: projections, selections and natural joins.

Figure 2.2 shows examples for the most common relational algebra operations for which relational lenses exist. A projection takes a table and removes specified columns. An example is shown in Figure 2.2a, which takes the table  $S$  and projects it onto the columns  $B$  and  $C$ . Selections filter a table according to some predicate or filter, an example filtering all rows where  $B < 3$  is shown in Figure 2.2b. Lastly, the natural join operation takes two tables and joins them on all common columns  $J$ . In the example shown in Figure 2.2c, two tables  $R$  and  $S$  are joined where  $J = B$ .

$\pi_{B,C}(S)$	$B$	$C$		$\sigma_{B<3}(S)$	$B$	$C$	$D$		$R \bowtie S$	$A$	$B$	$C$	$D$
	1	1			1	1	5			2	1	1	5
	2	2			2	2	6			3	2	2	6
	3	2			2	2	6			4	2	2	6

(a) Projection
(b) Selection
(c) Natural Join

Figure 2.2: Different relational algebra operations.

## 2.3 Links Programming language

The Links programming language is a language designed to unify the many different languages involved in web development. Web applications consist of server side code, which is normally written in one of many of different programming languages, client side code written in JavaScript and database queries formulated in SQL. Links makes use of code generation techniques in order to perform some operations on the client in JavaScript. It also makes use of Language Integrated Query (LINQ) and LINQ to SQL techniques in order to generate queries to be executed on the database.

While implementing relational lenses cannot be done as a library in the Links language, it is possible to extend the language to support this feature. This is because Links does not offer enough flexibility in its type system to be able to type check lenses.

## 2.4 Relational Lenses

This work is largely based on the existing work by Bohannon et al. [11] on relational lenses. Relational lenses define the get and put operations for join, projection and selection lenses in set semantics. Each lens that is defined is equipped with a correctness proof for both GetPut as well as PutGet. The following three sections contain a short overview of the different relational lenses. In all of the definitions,  $I$  refers to the unchanged database while  $J$  refers to the database with the changed view.

### 2.4.1 Helper Definitions

**Functional Dependencies:** One restriction that Bohannon et al. sets is that functional dependencies need to be in tree form [11]. Tree form is defined as being able to split the set of columns  $(X_1, \dots, X_n)$  into a set of disjoint sets, which then form an acyclic directed graph, where each edge represents a functional dependency. This graph must also have the limitation that each node may only appear once, and that each node may only have one edge pointing towards it. This implies that no two functional dependencies may point to the same node and that additionally if one node directly depends on a specific node, then no other node can depend on a part of that node.

In the work presented here we assume that all source tables have a simple primary key that determines all other columns.

We define the two functions, `left` and `right`, which return either the left or the right side of all functional dependencies  $X \rightarrow Y \in F$ :

$$\text{left}(F) = \bigcup_{X \rightarrow Y \in F} X$$

$$\text{right}(F) = \bigcup_{X \rightarrow Y \in F} Y$$

In addition, we require the definition for the roots of a set of functional dependencies. Roots are all nodes of the tree graph which have an outgoing arrow but no incoming arrows and can be defined as follows:

$$\text{roots}(F) = \{X \mid \exists Y. X \rightarrow Y \text{ and } X \cap \text{right}(F) = \emptyset\}$$

### Sort Function:

The sort function, which takes a table/view  $R$  and returns a tuple  $(U, P, F)$ , where  $\text{dom}(R) = U$ ,  $\text{pred}(R) = P$  and  $\text{fd}(R) = F$  [11]. The  $\text{dom}(R)$  function returns the domain of the table/view  $R$ . The  $\text{pred}(R)$  returns the predicate used to query the table/view  $R$ . The  $\text{fd}(R)$  returns the functional dependencies of  $R$ . This sort structure contains sufficient information to perform a query on a database.

**Single Dependency Record Revision:** The single dependency record revision operation takes a functional dependency  $X \rightarrow Y$ , a record  $m$ , and a set of records  $N$ . If  $m$  and an  $n \in N$  have the same value for  $X$ , then it returns  $m'$ , which is calculated by  $m$  by replacing all values for  $Y$  with those from  $n[Y]$ . Otherwise  $m$  is returned unchanged. Both  $m$  and  $n$  are required to have the same domain (specified as  $m : U$  and  $n : U$ ).  $X \rightarrow Y : U$  says that  $X$  and  $Y$  also have to be part of the domain  $U$ .  $N \models X \rightarrow Y$  says that the set of records  $N$  needs to be modelled by the functional dependency  $X \rightarrow Y$ . The additional judgement  $m[X] = n[X]$  states that the records  $m$  and  $n$  have identical values for all columns in  $X$ . Finally  $m \leftarrow +n[Y]$  takes all values for columns  $Y$  and adds them to the record  $m$ .

$$\frac{\begin{array}{c} m : U \quad n : U \quad X \rightarrow Y : U \\ N \models X \rightarrow Y \quad n \in N \\ m[X] = n[X] \quad m' = m \leftarrow +n[Y] \end{array}}{m \xrightarrow[N]{X \rightarrow Y} m'} \quad (\text{C-Match})$$

$$\frac{\begin{array}{c} m : U \quad n : U \quad X \rightarrow Y : U \\ N \models X \rightarrow Y \quad m \notin N[X] \end{array}}{m \xrightarrow[N]{X \rightarrow Y} m} \quad (\text{C-NoMatch})$$

**General Record Revision:** The general record revision operation takes a row  $m$  and a set of records  $L$  and applies all the given functional dependencies to  $m$ . If the set of functional dependencies is empty, then  $m$  is returned unchanged as it means that all functional dependencies have been applied. Otherwise the functional dependencies are split into a set of functional dependencies  $F$  and a

functional dependency  $X \rightarrow Y$ , where  $X$  is in the roots of all functional dependencies. The set of functional dependencies  $F$  needs to be in tree form. Single dependency revision is applied to  $m$  to produce  $m'$ , and then general record revision is applied using  $F$  and  $L$  on  $m'$  to produce  $n$ , which is the result of the general record revision. General record revision is defined using two inductive rules:

$$\frac{m : U \quad L : U}{m \xrightarrow[L]{\emptyset} m} \quad (\text{FC-Empty})$$

$$\frac{\begin{array}{c} L \models F, X \rightarrow Y \quad X \rightarrow Y \notin F \\ F \text{ in tree form} \quad X \in \text{roots}(F, X \rightarrow Y) \\ m \xrightarrow[L]{X \rightarrow Y} m' \quad m' \xrightarrow[L]{F} n \end{array}}{m \xrightarrow[L]{F, X \rightarrow Y} n} \quad (\text{FC-Step})$$

**Relation Revision:** The relation revision operation ( $\leftarrow_F$ ) takes two sets of records  $M$  and  $L$ , and applies general record revision to every record  $m \in M$  using the given functional dependencies  $F$ . Relation revision is defined as follows:

$$M \leftarrow_F L \triangleq \left\{ m' \mid m \xrightarrow[L]{F} m' \text{ for some } m \in M \right\}.$$

**Relational Merge:** The relational merge operation ( $\leftarrow_F^\cup$ ) applies relation revision and then merges the set  $N$  into the result of the relational revision operation. Relational merge is defined as follows:

$$M \leftarrow_F^\cup N \triangleq (M \rightarrow_F N) \cup N.$$

### 2.4.2 Selection Lens

The selection lens is defined as shown in Figure 2.3.  $R$  refers to the table in the input  $I$  which should be filtered, while  $S$  refers to the filtered version of  $R$  in the output  $J$ . The selection lens takes the set  $P$  as a parameter, which defines which records are filtered. Since  $P$  is defined as a set, the filter must be able to be definable by looking at a single row, ruling out aggregations and similar operations that may

depend on other rows. The put direction makes use of record revision in order to apply all functional dependencies of the updated version of  $J(S)$  to all records in  $I(R)$  that are not in the set  $P$ . In addition it merges in all records of  $J(S)$ . Any records that are in the set  $P$  but are not in  $J(S)$  are captured in  $N_{\#}$  and then removed in order to ensure PutGet. The **fd** function returns the functional dependencies of the table  $R$ .

$$\begin{aligned}
\text{get}(I) &= I \setminus_R [S \mapsto P \cap I(R)] \\
\text{put}(J, I) &= J \setminus_S [R \mapsto M_1 \setminus N_{\#}] \\
\text{where} \\
M_1 &= (\neg P \cap I(R)) \stackrel{\cup}{\leftarrow}_F J(S) \\
N_{\#} &= (P \cap M_1) \setminus J(S) \\
F &= \text{fd}(R)
\end{aligned}$$

Figure 2.3: The selection lens definition in set notation.

Figure 2.4 shows an example of how the put operation behaves on a selection lens. In this example the get direction of the lens filters out the last row ( $A = 4, B = 2, C = 2, D = 6$ ). This output is then modified by changing the row ( $A = 3, B = 2, C = 2, D = 6$ ) so that  $C = 6$ . The functional dependencies require all rows with  $B = 2$  to change  $C = 6$  and so the last row in the original table would be updated to ( $A = 4, B = 2, C = 2, D = 6$ ). This would make the row satisfy the predicate  $P$  though, because  $C = 6 > 3$ . If  $\sigma_P(T)$  were recalculated, all three rows would be included in the output. Since this does not satisfy PutGet, the last row needs to be deleted entirely.

The typing rule for the select lens is given as (T-Select). The rule states that the predicate in the output lens is equal to the intersection between the existing lens predicate and the new predicate. There is an additional requirement that says that the predicate for the underlying table needs to ignore the outputs of the functional dependencies. This is required, because otherwise a functional dependency update could cause the underlying tables predicate to not hold anymore.

$$T := R \bowtie S$$

$$P := C > 3 \vee A \leq 3$$

$$\text{put} \left( \begin{array}{c|cccc} T & A & B & C & D \\ \hline & 2 & 1 & 1 & 5 \\ & 3 & 2 & 2 & 6 \\ & 4 & 2 & 2 & 6 \end{array}, \begin{array}{c|cccc} \sigma_P(T) & A & B & C & D \\ \hline & 2 & 1 & 1 & 5 \\ & 3 & 2 & \mathbf{6} & 6 \end{array} \right) \mapsto \begin{array}{c|cccc} T & A & B & C & D \\ \hline & 2 & 1 & 1 & 5 \\ & 3 & 2 & \mathbf{6} & 6 \end{array}$$

Figure 2.4: An example of a selection lens put operation, assuming the functional dependencies  $A \rightarrow B$  and  $B \rightarrow CD$ .

$$\frac{\begin{array}{l} \text{sort}(R) = (U, Q, F) \quad \text{sort}(S) = (U, P \cap Q, F) \\ F \text{ is in tree form} \quad Q \text{ ignores } \text{outputs}(F) \end{array}}{\text{select from } R \text{ where } P \text{ as } S \in \Sigma \uplus \{R\} \Leftrightarrow \Sigma \uplus \{S\}} \quad (\text{T-Select})$$

### 2.4.3 Join Lens

The join lens takes two tables  $R$  and  $S$  from the input  $I$  and performs a natural join to form the table  $T$  in the output  $J$ . The definitions for its **put** and **get** operations are shown in Figure 2.5. An important requirement is that the right table  $S$  is completely defined by the left table  $R$ . In addition, the join lens also takes two sets  $P_d$  and  $Q_d$ , which define from which table a row has to be deleted, if it could be deleted from either. Every row has to either be in  $P_d$  or in  $Q_d$ :  $\forall x. x \in P_d \vee x \in Q_d$ . The put direction makes use of record revision in order to apply all functional dependencies from the output in order to apply to both input tables  $I(R)$  and  $I(S)$  and also merges in the output while doing so to produce  $M_0$  and  $N_0$ . After this it finds all entries that would appear in the output after joining again that need to be deleted as  $L$ , by calculating  $M_0 \bowtie N_0$  and removing all entries already in the output  $J(T)$ . Using  $L$  it determines which of the entries have to be deleted from the left table as  $L_l$  by removing all entries for which another entry in the output exists that contains the right table. Then all records that could be deleted from either of the tables are determined by subtracting all entries in  $L_l$  from  $L$ . Finally, the resulting new table  $R$  is determined as all entries in  $M_0$  without all entries in  $L_a$  that are also in  $P_d$  and without all entries



in  $L_l$  projected onto the columns of the left table  $R$ . Similarly the resulting new table  $S$  is calculated by taking all entries in  $N_0$  and removing all entries in  $L_a$  that are also in  $Q_d$  and projecting it onto the columns of the right table  $S$ .

$$v = \text{join\_template } (R, P_d) (S, Q_d) \text{ as } T$$

$$\text{get}(I) = I \setminus_{R,S} [T \mapsto I(R) \bowtie I(S)]$$

$$\text{put}(J, I) = J \setminus_T [R \mapsto M][S \mapsto N]$$

where

$$(U, P, F) = \text{sort}(R)$$

$$(V, Q, G) = \text{sort}(S)$$

$$M_0 = I(R) \xleftarrow{\cup?}_F J(T)[U]$$

$$N_0 = I(S) \xleftarrow{\cup?}_G J(T)[V]$$

$$L = M_0 \bowtie N_0 \setminus J(T)$$

$$L_l = L \bowtie (J(T)[U \cap V])$$

$$L_a = L \setminus L_l$$

$$M = (M_0 \setminus (L_a \cap P_d)[U]) \setminus L_l[U]$$

$$N = N_0 \setminus (L_a \cap Q_d)[V]$$

Figure 2.5: The join lens definition in set notation.

The typing rule for the join lens is given as (T-Join). The columns of the two lenses are combined and the predicate in the sort structure is taken as the join of the two predicates, while the union of the both functional dependency sets is taken. It also requires that the join key fully defines the right table and the join key is required to be the intersection of the two columns. We assume the case where the relational merge  $\xleftarrow{\cup}_F$  operator is used, along with  $\Phi(U, P, F) = F$  is in tree form and  $P$  ignores  $\text{outputs}(F)$ .

$$\frac{\begin{array}{l} \text{sort}(R) = (U, P, F) \quad \text{sort}(S) = (V, Q, G) \\ \text{sort}(T) = (UV, P \bowtie Q, F \cup G) \quad G \models U \cap V \rightarrow V \\ P_d \cup Q_d = \top_{UV} \quad \Phi(U, P, F) \quad \Phi(V, Q, D) \end{array}}{\text{join\_template } (R, P_d), (S, Q_d) \text{ as } T \in \Sigma \uplus \{R, S\} \Leftrightarrow \Sigma \uplus \{T\}} \quad (\text{T-Join})$$

Bohannon et al. also define an additional operator which can be used instead of the relational merge operator called the squash operator where  $\Phi(U, P, V) = \text{true}$ . The squash operator removes records that conflict with functional dependencies instead of trying to modify them to conform with other functional dependencies. We do not consider the squash operator in the work presented here.

#### 2.4.4 Projection Lens

The projection lens is defined as shown in Figure 2.6. It can be used to drop a column  $A$ , which is defined by another column  $X$  and has a default value  $a$ . In the unchanged database the table  $R$  refers to the table on which the projection is defined, while the table  $S$  refers to the changed table in the changed database  $J$ . The put direction works by finding all unchanged rows by performing a natural join on  $I(R)$  and  $J(S)$ , and then adds all rows that are in  $J(S)$  but not in the input  $I(R)$  with the default value  $a$ . After this, it tries to find a row in  $I(R)$  with the same value for the  $X$  column and uses the  $A$  columns value of that row instead of the existing value.

$$\begin{aligned}
 \text{get}(I) &= I \setminus_R [S \mapsto I(R)[U - A]] \\
 \text{put}(J, I) &= J \setminus_S [R \mapsto M \leftarrow_{X \rightarrow A} I(R)] \text{ where} \\
 M &= (I(R) \bowtie J(S)) \cup (N_+ \bowtie \{\{A = a\}\}) \\
 N_+ &= J(S) \setminus I(R)[U - A] \\
 U &= \text{dom}(R)
 \end{aligned}$$

Figure 2.6: The drop column definition in set notation.

The typing rule for the projection is given as (T-Drop). It says that the functional dependencies of the left table should include  $X \rightarrow A$ , which is removed from the functional dependencies of the right table. It also ensures that  $A$  is in the columns of the left table and is removed from the right table, while any references to  $A$  in  $P$  should be removed.  $P$  also requires that it should be able to partition it into a part that depends on everything except  $A$  and a part depending on  $A$ , which can then be joined. This allows references to  $A$  to be removed from  $P$ . Finally if it is partitioned into the two parts, the default value  $a$  should be in the set  $P[A]$ .

$$\begin{array}{c}
\text{sort}(R) = (U, P, F) \quad A \in U \quad F \equiv F' \cup \{X \rightarrow A\} \\
\text{sort}(S) = (U - A, P[U - A], F') \\
\frac{P = P[U - A] \bowtie P[A] \quad \{A = a\} \in P[A]}{\text{drop } A \text{ determined by } (X, a) \text{ from } R \text{ as } S \in \Sigma \uplus \{R\} \Leftrightarrow \{S\}} \quad (\text{T-Drop})
\end{array}$$

### 2.4.5 Composition Lens

The composition lens takes two lenses  $v$  and  $w$  and composes them into a single lens. The formal definition is shown in Figure 2.7. The **get** functions of the two lenses are simply composed. The put direction requires the unchanged view of  $w$  for  $v$ 's put, and so it is calculated using  $\text{get}_w(I)$ . Once the updated view of lens  $w$  is determined, it can be put into  $\text{put}_w$  along with the unchanged  $I$ .

$$\begin{aligned}
\text{get}_{v,w}(I) &= \text{get}_v(\text{get}_w(I)) \\
\text{put}_{v,w}(J, I) &= \text{put}_w(\text{put}_v(J, \text{get}_w(I)), I)
\end{aligned}$$

Figure 2.7: The composition lens that takes two lenses  $w$  and  $v$  and combines them.

The typing rule for composition states that the domains of the two lenses must coincide ( $\Leftrightarrow$ ) with each other.

$$\frac{v \in \Sigma \Leftrightarrow \Sigma' \quad w \in \Sigma' \Leftrightarrow \Delta}{v; w \in \Sigma \Leftrightarrow \Delta} \quad (\text{T-Compose})$$

In our incremental lenses we define a **put** which recursively calls itself and so no composition lens is required.



# Chapter 3

## Incremental Relational Lenses

### 3.1 Overview

Relational lenses allow an editable view of an underlying relational database to be created. The incrementalized version of relational lenses allows changes to these views to be converted into changes to the underlying database while not requiring the entire database to be loaded in memory, but instead querying the smaller amount of data required to perform a valid update.

The `get` operation of a lens can easily be translated into a single database query. For example assuming we have two tables  $R : (\underline{A}, B, C)$  and  $S : (\underline{C}, D, E)$  we could define the lens  $T : (\underline{A}, B, C, D, E) := R \bowtie S$ . In order to propagate changes made to this view in an incremental fashion, a few more steps are necessary. The first task is to determine what changes have been made to the view, by comparing the changed view to the original view. This allows us to determine which rows have been inserted or removed. The process of generating such a change set is described in Section 3.2.

This change set can then be taken and adapted, so that it reflects the changes to the underlying data. In order to do this, each lens primitive is adapted to propagate changes to the underlying lens or table. The changes go through each lens, one after another, until the change set can be applied to the underlying relational tables.

After propagating the change set through all the lenses, the change set is con-

verted into a set of SQL commands on the underlying database tables. These SQL commands are executed in order to push the changes to the database.

## 3.2 Change Set Generation

Incremental relational lenses work by taking a change set which describes the rows that have been added, changed or deleted and maintaining this change set through the different lenses. The user hands the lenses a modified version of the view which describes what the view should look like. As such the first step required for using incremental relational lenses is to generate a set of changes compared to the unmodified view.

The change set is stored in tuples  $(t, m)$ , where  $t$  is the record and  $m$  is the multiplicity. The multiplicity signifies if a record has been added ( $m = +1$ ), deleted ( $m = -1$ ) or if it is unchanged ( $m = 0$ ). This work makes the assumption that tables / views are not multi sets and that records can only exist once, which is why  $-1 \leq m \leq 1$ . This is the case if all tables have keys. The change set can be generated by finding all record tuples  $t$  which only exist in the modified view and marking them with  $m = +1$ . Then all record tuples  $t$  which only exist in the unmodified view are taken with  $m = -1$  and the rest are taken with  $m = 0$ .

The unmodified rows are kept since they can aid and simplify computations at a later stage. They are not necessarily required however, and it would be possible to remove them.

**Definition** `change_set`  $(T : \text{row set}) (T' : \text{row set}) :=$

$$\{(t, -1) \mid t \in T \setminus T'\} \cup \{(t, 0) \mid t \in T \cap T'\} \cup \{(t, +1) \mid t \in T' \setminus T\}$$

An example of a change set being generated is shown in Figure 3.1.  $T$  is the view calculated by the get direction, and  $T'$  is the modified view. The change set  $\Delta T$  is generated using  $T$  and  $T'$ .

The multiplicity type is defined as follows:

**Type** `mult`  $:= -1 \mid 0 \mid +1$ .

$T$	$A$	$B$	$C$	$D$	$T'$	$A$	$B$	$C$	$D$	$\Delta T$	$A$	$B$	$C$	$D$
	2	1	1	5		2	1	1	5	0	2	1	1	5
	3	2	2	6		3	2	6	6	-1	3	2	2	6
										+1	3	2	6	6

Figure 3.1: Change set of the input set  $T'$  using the unchanged view  $T$ .

The type of a change set entry is defined as:

**Type** `change_entry` := `row * mult`.

The type of a change set is formally defined as:

**Type** `change_set` := `change_entry set`.

## 3.3 Invariant Properties

### 3.3.1 No conflicting output

It is necessary to ensure that the output satisfies all functional dependencies. As such all rows that appear in the output (which are all rows with  $m = 0$  and  $m = +1$ ) need to satisfy all functional dependencies. Examples of conflicting output are shown in (3.1), (3.2) and (3.3).

$\Delta S$	$A$	$B$	$C$
+1	2	2	1
+1	2	1	1 <b>err</b>

(3.1)

$\Delta S$	$A$	$B$	$C$
+1	1	2	1
+1	2	2	2 <b>err</b>

(3.2)

$\Delta S$	$A$	$B$	$C$
0	2	2	1
+1	2	1	1 <b>err</b>

(3.3)

### 3.3.2 Validity of changes

While propagating changes through each lens, we need to ensure that no important information is lost. We cannot assume that change sets are complete, and so we cannot calculate the correct output by simply adding rows with  $m = +1$  and removing rows with  $m = -1$ . Instead, we assume that we always have sufficient information, such that it is possible to determine the correct output given functional dependency constraints and explicit row deletions.

We define an operation which takes a change set and produces the correct output even if the lens change set is not complete. It is possible to determine the correct output by removing all records from the unchanged view which are marked with  $m = -1$  in the change set and to then apply relational merge with all records in the change set  $\Delta N$  where  $m = +1$ . This operation will be referred to as *relational delta merge* and is defined as follows:

$$M \xleftarrow{\Delta \cup}_F \Delta N \triangleq (M - \{t \mid (t, m) \in \Delta N \wedge m = -1\}) \xleftarrow{\cup}_F \{t \mid (t, m) \in \Delta N \wedge m = +1\}.$$

Given the relational delta merge operation, we define the property which all lenses must satisfy for the `delta_put` operation, to ensure that no information is lost during propagation. It is assumed that calling `get` on  $\mathbf{X}$  using a lens  $l$  calculates  $\mathbf{Y}$ , where both are assumed to be a set of tables where all operations are applied componentwise. The property says that if `get` on  $\mathbf{X}$  produces  $\mathbf{Y}$ , then the relational delta merge of  $\Delta \mathbf{Y}$  to  $\mathbf{Y}$  must be the same as calling `get` on  $\mathbf{X}$  after applying the changes in  $\Delta \mathbf{Y}$  to them after passing through the incremental lens in the `put` direction. The property is formally defined as follows:

$$\text{get } l \mathbf{X} = \mathbf{Y} \rightarrow \text{get } l \left( \mathbf{X} \xleftarrow{\Delta \cup}_F \text{delta\_put } l \Delta \mathbf{Y} \right) = \mathbf{Y} \xleftarrow{\Delta \cup}_F \Delta \mathbf{Y}.$$

### 3.3.3 Changes contain complements

We assume that if we have an entry with  $m = +1$ , and there already exists a row in the underlying table, then we need to have that entry marked with  $m = -1$  in the change set.



$$\begin{array}{c|ccc}
S & A & B & C \\
\hline
& 2 & 2 & 1
\end{array} \tag{3.4}$$

$$\begin{array}{c|ccc}
\Delta S & A & B & C \\
\hline
-1 & 2 & 2 & 1 \\
+1 & 2 & 2 & 2
\end{array} \tag{3.5}$$

## 3.4 Notation

### 3.4.1 Phrase Quasiquotation

In order to better explain how queries are formed, quasiquotation type notation is used. While we use relational algebra instead of SQL to query the database, we still use quasiquotation to generate predicates and database modification statements. In this case the symbols  $\ll$  and  $\gg$  are used to start defining a query which is to be executed on the database server. In addition to the quasiquotation, statements can be unquoted and thus diverted to local execution using  $\llbracket$  and  $\rrbracket$ . Assuming the following statement was executed in the environment  $t = \text{"myTable"}; k = \text{"myId"}; a = 5; b = 6$ :

$$\ll \text{SELECT } * \text{ FROM } \llbracket t \rrbracket \text{ WHERE } \llbracket k \rrbracket = \llbracket a + b \rrbracket \gg.$$

This would result in the following actual query, without considering correct escaping:

$$\text{SELECT } * \text{ FROM } myTable \text{ WHERE } myId = 11.$$

We assume that quasiquotation produces values of type **phrase**, which are actually abstract syntax trees. These expressions can be SQL queries as well as boolean expressions used for predicates. In the case of SQL queries, we assume that quasiquotation automatically handles escaping queries and ensuring that no SQL injection attacks happen.

## 3.5 Helper Definitions

### 3.5.1 Common Functions

The function  $\text{dom}(l)$  returns the domain of the view  $l$ , which is the set of all columns in the view. Next, function  $\text{pred}(l)$  returns the predicate  $P$ , which is required to query the view  $l$ . Additionally, we have the function  $\text{fd}$  which returns the functional dependencies of a lens  $l$ .

We also define the function  $\text{key}(l)$ , which returns the minimal key that uniquely identifies a row. This key is assumed to be unique, since the functional dependencies are expected to be in tree form, and thus no functional dependency cycles can exist.

### 3.5.2 Sort Function

We reuse the definition of the sort function, which takes a lens / view  $l$  and returns a tuple  $(U, P, F)$ , where  $\text{dom}(l) = U$ ,  $\text{pred}(l) = P$  and  $\text{fd}(l) = F$  [11]. This sort structure contains sufficient information to perform a query on a database.

### 3.5.3 Record Functions

We first define a function  $\text{match\_on}$ , which takes two rows  $t$  and  $t'$  and compares all columns in  $cols$ .

**Definition**  $\text{match\_on} (t : \text{row}) (t' : \text{row}) (cols : \text{colset}) :=$

$$\bigwedge \{ t[c] = t'[c] \mid c \in cols \}$$

In addition, we define a function  $\text{match\_on\_expr}$  which produces a predicate expression usable in a query to compare all columns in  $cols$  to a record  $t$ . This function returns a value of type **phrase**.

**Definition**  $\text{match\_on\_expr} (t : \text{row}) (cols : \text{colset}) :=$

$$\ll \bigwedge \{ \ll [c] = [t[c]] \gg \mid c \in cols \} \gg$$

We also define a function `compl_rows`, which takes a change entry  $(t, m)$ , a change set  $\Delta R$  and a set of columns  $key$ , and finds all rows with a negated multiplicity and the same values for the given  $key$ .

**Definition** `compl_rows`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set}) (key : \text{colset}) :=$   
 $\{(t', m') \mid (t', m') \in \Delta R, \text{match\_on } t \ t' \text{ cols}, m' = -m\}$

### 3.5.4 Update Set

The invariant property for validity of changes (see Section 3.3.2) insures that the essential information is retained between each `delta_put` step. This essential information can be applied using the relational delta merge operation and consists of rows to remove and containing all necessary functional updates. Another way to describe the functional updates is through an update set. An update set contains the mappings of values for each functional dependency  $f$  in a set of functional dependencies  $F$ . Example (3.6) shows the update set of a given change set  $\Delta R$ .

$$\begin{array}{c|ccc}
 \Delta R & A & B & C \\
 \hline
 -1 & 1 & 2 & 3 \\
 +1 & 1 & 3 & 2 \\
 -1 & 2 & 1 & 3 \\
 +1 & 2 & 1 & 4 \\
 0 & 3 & 4 & 5 \\
 +1 & 4 & 5 & 6 \\
 -1 & 5 & 6 & 7
 \end{array}
 \Rightarrow
 \begin{array}{l}
 A \rightarrow B : \\
 1 \rightarrow 3 \\
 2 \rightarrow 1 \\
 4 \rightarrow 5 \\
 \\
 B \rightarrow C : \\
 3 \rightarrow 2 \\
 1 \rightarrow 4 \\
 5 \rightarrow 6
 \end{array}
 \quad (3.6)$$

The definition of `get_fd_updates` is used, which takes a change set  $\Delta R$  as well as a functional dependency  $f$  and determines all new mappings from `left(f)` to `right(f)`. It does this by finding all changes in the change set  $\Delta R$  where the multiplicity is +1 and then projects them onto `left(f)` and `right(f)`.

**Definition** `get_fd_updates`  $(\Delta R : \text{change\_set}) (f : \text{fundep}) :=$   

$$\left( f, \left\{ \pi_{\text{left}(fd)}(t), \pi_{\text{right}(fd)}(t) \mid (t, +1) \in \Delta R \right\} \right)$$

Using `get_fd_updates`, `get_updateset` is defined which takes a change set  $\Delta R$  as well as the set of functional dependencies  $F$  and calculates the set of `get_fd_updates` for each functional dependency  $f$  in  $F$ .

**Definition** `get_updateset`  $(\Delta R : \text{change\_set}) (F : \text{fundepset}) :=$   

$$\{\text{get\_fd\_updates } \Delta R \ f \mid f \in F\}$$

### 3.6 Select Lens

The select lens takes a table  $R$ , e.g.  $R : (\underline{A}, B, C)$ , and filters out rows that do not satisfy a given predicate  $P$ . Since the select lens only filters entries, the output table type  $T$  has the same type as the input table  $R$ , so for the current example it would be  $T : (\underline{A}, B, C)$ . The output of a select lens is required to satisfy the predicate  $P$ , as it would otherwise not satisfy `get(put( $X, Y$ ))`.

Most changes from the select lens can propagate without much change. There are two special cases which need to be considered, though.

The first is where additional records have to be deleted. An example of this is shown in Figure (3.2). If the user alters the result of the selection lens with the predicate  $P = C > 3 \vee A \leq 3$ , then it is necessary to delete the additional row  $(A = 4, B = 2, C = 2, D = 6)$ . This is because the functional dependency  $B \rightarrow C$  holds, and given the changes  $(A = 4, B = 2, C = 2, D = 6)$  would change to  $(A = 4, B = 2, C = 6, D = 6)$ . This update would cause the row to satisfy  $P$  and `get` would cause the row to appear in the output, which would not satisfy *PutGet*.

The solution is to find all rows on the underlying table that do not satisfy  $P$ , but do after some updated functional dependencies. For this it is necessary to be able to determine what  $P$  would be after all functional dependencies. This is done

	$R$	$A$	$B$		$S$	$B$	$C$	$D$
		2	1			1	1	5
		3	2			2	2	6
		4	2			3	2	7

					$\Delta(\sigma_{C>3\vee A\leq 3}(R\bowtie S))$	$A$	$B$	$C$	$D$	
$\sigma_{C>3\vee A\leq 3}(R\bowtie S)$	$A$	$B$	$C$	$D$	0	2	1	1	5	
	2	1	1	5	$\Rightarrow$	-1	3	2	2	6
	3	2	<del>2</del> <span style="color: red;">6</span>	6		+1	3	2	6	6
						-1	4	2	2	6

Figure 3.2: The correct change set that needs to be produced given the changes in the view on the left, assuming the underlying tables  $R$  and  $S$ .

by replacing each occurrence of a variable in  $P$  with an expression that would calculate its updated value. We make use of the functional dependency update set as described in Section 3.5.4. The example in Figure (3.3) shows the SQL expression for the updated value of the column  $C$ , defined by the given update set.

Update Set:		Updated expression for $C$ :	
$A \rightarrow B$ :		<b>CASE</b>	
$1 \rightarrow 3$		<b>WHEN <math>A = 1</math> THEN 2</b>	
$2 \rightarrow 1$		<b>WHEN <math>A = 2</math> THEN 4</b>	
$4 \rightarrow 5$		<b>WHEN <math>A = 4</math> THEN 6</b>	
		<b>ELSE</b>	
		<b>CASE</b>	
$B \rightarrow C$ :		<b>WHEN <math>B = 3</math> THEN 2</b>	
$3 \rightarrow 2$		<b>WHEN <math>B = 1</math> THEN 4</b>	
$1 \rightarrow 4$		<b>WHEN <math>B = 5</math> THEN 6</b>	
$5 \rightarrow 6$		<b>ELSE <math>C</math></b>	
		<b>END</b>	
		<b>END</b>	

Figure 3.3: An example of how to determine the SQL expression for the column  $C$  given an update set.

We build these case expressions by using the helper definition `var_case_expr` which produces a case statement in the form `CASE (WHEN left fd is satisfied THEN value)* ELSE other END`, where the when part of the statement appears for each possible updated value for the functional dependency *fd* and *other* contains a further phrase. The definition of `var_case_expr` is as follows:

**Definition** `var_case_expr (map : row * phrase) (or : phrase) (fd : fundep) :=`  
`<< CASE [`  
`{ << WHEN [match_on_expr t left (fd)] THEN [v] >> | (t,v) ∈ map }`  
`] ELSE [or] END >>`

Using the definition for case expressions, the definition `updated_var_expr` is introduced, which takes an update set *upl*, the key of the current table entry *key* and a column *col* for which the expression should be produced and creates an entire case match expression as shown in Figure 3.3. The definition of `updated_var_expr` relies on an additional function `calc_updated_var_expr`, which determines a CASE expression which covers the key of the row until *col* and falls back to the phrase given by *or*. The `singleton` function takes a set with one element and returns its single element. Because the functional dependencies are assumed to be in tree form and we are traversing up the tree, there will always only be one element in the list.

**Definition** `calc_updated_var_expr` (*chl* : **change\_set**) (*key* : **colset**) (*col* : **colset**)  
 (*or* : **phrase**) (*map* : (**row** \* **row**) **list option**) :=  
 let *f, changes* = `singleton`({*f, changes* | (*f, changes*) ∈ *chl*, *col* ⊂ `right`(*f*)})  
 let *map'* = **match** *map* **with**  
   | `None` → *changes*  
   | `Some map` → {(*k, v*) | (*k, k'*) ∈ *changes*, (*k', v*) ∈ *map*}  
 if *key* ⊂ `left`(*f*) **then**  
   `var_case_expr` *map'* *or f*  
**else**  
   `calc_update_var_expr` *chl* *key* `left`(*f*) (`var_case_expr` *map'* *or f*) (`Some map'`)

**Definition** `updated_var_expr` (*chl* : **change\_set**) (*key* : **colset**) (*col* : **colset**) :=  
`calc_updated_var_expr` *chl* *key* *col* << `[[col]]` >> `None`

Next, we define a function `updated_pred`, which takes a phrase *P* and replaces all occurrences of a variable node within *P* with a case expression calculated by `updated_var_expr`. We assume the function `phrase_map` exists, which takes a phrase and function *f* : **phrase** → **phrase** which maps each phrase node *n* onto the phrase node *f*(*n*). The `phrase_map` returns a value of type **phrase** and so does the `updated_pred` function.

**Definition** `updated_pred` (*ΔR* : **change\_set**) (*l* : **lens**) (*P* : **phrase**) :=  
 let *chl* := `get_updateset` *ΔR* *fd*(*l*)  
`phrase_map` *P* (**fun** *node* →  
   **match** *node* **with**  
     `Var` *n* → `updated_var_expr` *chl* *key*(*l*) *n*  
     \_ → *node* )

The required rows can then be queried using the following definition for `query_deleted_rows`. We assume there is a function `query`, which takes a lens *l* and then queries the database using SQL.

**Definition** `query_deleted_rows`  $(\Delta R : \text{change\_set}) (l : \text{lens}) (P : \text{phrase}) :=$   
 $\text{query}(\sigma_{\ll \ll \neg P \rrbracket \wedge \ll \text{updated\_pred } \Delta R \ S \ P \rrbracket \gg}(l))$

Finally, we define the function `delta_put_select`, which takes a lens  $l$  and a predicate  $P$  as a **phrase** and a change set  $\Delta R$  and calculates the change set for the underlying lens. It takes all changes in  $\Delta R$  and then adds all rows which need to be deleted.

**Definition** `delta_put_select`  $(l : \text{lens}) (P : \text{phrase}) (\Delta R : \text{change\_set}) :=$   
 $\Delta R \cup \{(t, -1) \mid t \in \text{query\_deleted\_rows } \Delta R \ l \ P\}$

### 3.7 Projection Lens

The projection lens takes a table  $S$  and removes a column  $A$ . In order to determine this  $A$  column's value in the put direction, the user needs to specify which column defines its value as  $X$ . It is required that the functional dependency  $X \rightarrow A$  exists. When performing the **put**, the value for  $A$  is attempted to be found by looking for a row with an identical  $X$  value in the underlying lens. If no such row is found, then the column receives the default value  $a$ , which must also be specified by the user in the lens definition. The output table  $R$  contains all columns of  $S$  except for the column  $A$ .

One efficient way of determining the correct output is to query the database for a lookup table that describes the value of all occurring  $X \rightarrow A$  pairs. This can be done by selecting all distinct values of  $X$  and  $A$  for the underlying view where  $X$  is equal to any of the values occurring in the new view. An example of how to lookup the values is shown in Figure 3.4.

We first define a function `any_match_expr` to generate a phrase, which finds all rows with an  $X$  value matching any  $X$  in the change set. This function can be defined as follows:



$S$	$B$	$C$	$D$
	1	1	5
	2	2	6
	3	2	7

$\Delta(\pi_{B,C}(S))$	$B$	$C$		$L$	$B$	$D$		$\Delta S$	$B$	$C$	$D$
0	1	1	$\Rightarrow$		1	5	$\Rightarrow$	0	1	1	5
0	2	2			2	6		0	2	2	6
-1	3	2						-1	3	2	7
+1	4	3						+1	4	3	4

with  $L := \sigma_{B=1 \vee B=2 \vee B=4}(\pi_{B,D}(S))$

Figure 3.4: An example of calculating the lookup table for the change set  $\Delta(\pi_{B,C}(S))$  with default value 4 and the defining column set to  $B$ .

**Definition** `any_match_expr` ( $\Delta R : \text{change\_set}$ ) ( $X : \text{colset}$ ) :=  
 $\ll \bigvee \{ \text{match\_on\_expr } t \ X \mid (t, m) \in \Delta R \} \gg$

We then define a function `query_lookup_table` which generates the lookup table using the expression from `any_match_expr`. We assume the function called `query_distinct` exists, which explicitly does not return a multi set and removes any duplicates. This is achieved by adding the keyword **DISTINCT** to a SQL query. Removing duplicates ensures that the lookup table cannot be larger than the size of the change set, since it can only be as large as the distinct number of  $X$  values occurring in the change set. For the example shown in Figure 3.4, this would be achieved using the following query:

**SELECT DISTINCT  $B, D$  FROM  $S$  WHERE  $B = 1$  OR  $B = 2$  OR  $B = 4$**

The function `query_lookup_table` can formally be defined as follows:

**Definition** `query_lookup_table`  $(\Delta R : \text{change\_set}) (l : \text{lens}) (X : \text{colset})$   
 $(A : \text{colset}) :=$

$$\text{query\_distinct} \left( \sigma_{\text{any\_match\_expr}} \Delta R \ X \left( \pi_{X,A}(l) \right) \right)$$

Next we define a helper function `lookup_col`, which takes a row  $t$  and finds the correct value for  $A$  using the lookup table  $L$ . It does so by trying to find an entry in  $L$  with a matching  $X$  value, and if it does find one it uses the  $A$  value from that entry. If no such entry is found, the default value  $a$  is taken. The function `lookup_col` is defined as follows:

**Definition** `lookup_col`  $(t : \text{row}) (A : \text{colset}) (X : \text{colset})$

$$\begin{aligned} & (a : \tau_A) (L : \text{row set}) := \\ & \text{let } C = \{t' \mid t' \in L, \text{match\_on } t \ t' \ X\} \\ & \text{match } C \text{ with} \\ & \quad | [] \rightarrow t \cup A = a \\ & \quad | t' :: T \rightarrow t \cup t'[A] \end{aligned}$$

We combine all the previous helper functions to define `delta_put_project`. The lookup table  $L$  is determined using `query_lookup_table` and then each row  $t$  in  $\Delta R$  is extended with the value for  $A$  which is determined by `lookup_col`. The function is defined as follows:

**Definition** `delta_put_project`  $(l : \text{lens}) (A : \text{colset}) (X : \text{colset}) (a : \tau_A)$

$$\begin{aligned} & (\Delta R : \text{change\_set}) := \\ & \text{let } L = \text{query\_lookup\_table } \Delta R \ l \ X \ A \\ & \{(lookup\_col \ t \ A \ X \ a \ L, m) \mid (t, m) \in \Delta R\} \end{aligned}$$

### 3.8 Join Lens

The join lens takes two tables  $S$  and  $T$ , e.g.  $S : (\underline{A}, B, C)$  and  $T : (\underline{C}, D, E)$ , and performs a join in order to produce the table  $R$ , in the given example  $R :$

$(\underline{A}, B, C, D, E)$ , where each row is a combination of a row in  $S$  and a row in  $T$  with the same value for join columns, which in the example this would be  $C$ . In the following we use  $P_l$  to be the set of columns which transitively define the left table. We define  $J$  as the join key. We always assume that the right table is fully defined by join column of the left table (hence  $J \rightarrow \text{dom}(R)$ ), and thus  $P_l \rightarrow \text{dom}(R)$ , since  $P_l \rightarrow J$ .

In the get direction it is easy to determine the output, since all that is necessary is to perform the cross product on the two tables  $R$  and  $S$ , and filter those with identical  $J$  values.

The put direction of join lenses is not uniquely defined and Bohannon et al. define a join template which specifies two predicates  $P_d$  and  $Q_d$ . The two predicates define if the lens should attempt to delete from the left table or the right table in an ambiguous case, and require that  $P_d \vee Q_d = \text{true}$ . Bohannon et al. define three example join lenses using this template:

<b>join_dl</b>	Attempt to delete from the left table first.	$P_d = \text{true}$ and $Q_d = \text{false}$
<b>join_dr</b>	Attempt to delete from the right table.	$P_d = \text{false}$ and $Q_d = \text{true}$
<b>join_both</b>	Attempt to delete from both tables.	$P_d = \text{true}$ and $Q_d = \text{true}$

Similarly, we define a template which takes two predicates  $P_d$  and  $Q_d$  which works for all three examples and additionally any further more complicated predicates that the user may have.

We first introduce some helper definitions required by the join lens in Section 3.8.2.1 and following sections. The join lens is defined as a collection of different cases, where we try to reason about each case individually and then ensure that all cases are covered. The different cases are covered in the sections following Section 3.8.3.

### 3.8.1 Notation and Assumptions

Given a lens or view  $T$ , the domain  $\text{dom}(T)$  is defined as all columns of  $T$ .

We use the example table  $R = S \bowtie T : (\underline{A}, B, C, D, E)$  which is defined as the join

of tables  $S : (\underline{A}, B, C)$  and  $T : (\underline{C}, D, E)$  joined on  $J = \{C\}$ . The defining columns of  $S$  are  $P_S = \{A\}$ , while the defining columns of  $T$  are  $P_T = \{C\}$ .

The projection of a row  $t$  to the columns of left is defined as  $\pi_U(t)$  while a projection onto the columns of the right table is defined as  $\pi_V(t)$ .

## 3.8.2 Helper Definitions

### 3.8.2.1 Row Neutrality

In order to determine how a change entry should be handled by the join function, we need to define some helping definitions.

The first helper function determines if a row is neutral. Neutral rows have a multiplicity of  $m = 0$  and indicate that the row has not been modified in any way.

An example of a neutral entry is shown in (3.7).

$$\begin{array}{c|ccccc} \Delta R & A & B & C & D & E \\ \hline 0 & 2 & 1 & 1 & 2 & 1 \end{array} \quad (3.7)$$

The condition can be defined as:

**Definition** `is_neutral`  $((t, m) : \text{change\_entry}) := m = 0$

### 3.8.2.2 Similar Neutral Entries

The next helper function determines if there is a change entry which contains identical values for the right side, which is also neutral.

Assuming the left table is defined as  $S = (A, B, C)$  and the right table is  $T = (C, D, E)$ , an example of where `ntrl_exists_right = true` for the change entry  $(\{A = 2, B = 2, C = 1, D = 3, E = 2\}, -1)$  is shown in (3.8).

$\Delta R$	$A$	$B$	$C$	$D$	$E$
-1	1	2	1	3	2
0	2	2	1	3	2

(3.8)

There is no complementary `ntrl_exists_left`, as this could never be true, since the right part of a row must be fully defined by the left. The definition for `ntrl_exists_right` is defined in such a way that it is only valid for non neutral elements:

**Definition** `ntrl_exists_right`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set}) (l_2 : \text{lens}) :=$   
 $\neg \text{is\_neutral } (t, m) \wedge \exists (t', m') \in \Delta R. m' = 0 \wedge \text{match\_on } t \ t' \ \text{key}(l_2)$

### 3.8.2.3 Complementary Entries

Similar to determining if there is a neutral change entry, there is also a method to determine if there is a complementary update row. The existence of a complementary entry tells us that there are either updates, or if the complementary entries are identical, that it has not been updated and is neutral. An example of complementary rows is shown in (3.9).

$\Delta R$	$A$	$B$	$C$	$D$	$E$
-1	1	2	1	3	5
+1	1	2	1	3	2

(3.9)

The complementary entries are divided into definitions for the left and the right table. The only rows that can have complements are non neutral rows. The property `compl_exists_left` is true if there exists a row with a negated  $m$  value with the same values for the left tables key, while the property `compl_exists_right` is true if there exists a row with a negated  $m$  value and the same values for the right tables key. For our example both `compl_exists_left` and `compl_exists_right` are true. We define the two properties as follows:

**Definition** `compl_exists_left`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set}) (l_1 : \text{lens}) :=$   
 $\neg \text{is\_neutral } (t, m) \wedge \exists (t', m') \in \Delta R. m' = -m \wedge \text{match\_on } t \ t' \ \text{key}(l_1)$

**Definition** `compl_exists_right`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set}) (l_2 : \text{lens}) :=$   
 $\neg \text{is\_neutral } (t, m) \wedge \exists (t', m') \in \Delta R. m' = -m \wedge \text{match\_on } t \ t' \ \text{key}(l_2)$

### 3.8.2.4 Update Entries

We now define a property that tells us if a row which satisfies `compl_exists_left` is also an update row for either the left table  $S$  or right table  $T$ . Looking at the previous example (3.9) it can be seen that these rows are an update entry for the right table  $T$  but not for the left table  $S$ . In order to determine if an entry contains an update or is a complement without updates, we can compare all values of  $\text{dom}(S)$  for the left table, or  $\text{dom}(T)$  for the right table and determine if they are identical or not.

**Definition** `upd_left`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set}) (l_1 : \text{lens}) :=$   
 $\text{compl\_exists\_left } (t, m) \ \Delta R \ l_1 \wedge \exists (t', m') \in \Delta R. \text{match\_on } t \ t' \ \text{key}(l_1) \wedge$   
 $\neg \text{match\_on } t \ t' \ \text{dom}(l_1)$

**Definition** `upd_right`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set}) (l_2 : \text{lens}) :=$   
 $\text{compl\_exists\_right } (t, m) \ \Delta R \ l_2 \wedge \exists (t', m') \in \Delta R. \text{match\_on } t \ t' \ \text{key}(l_2) \wedge$   
 $\neg \text{match\_on } t \ t' \ \text{dom}(l_2)$

Additionally, we defined to determine when a row has a complementary row, but where the complementary row does not contain any differences. These properties are defined as follows:

**Definition** `non_upd_left`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set}) (l_1 : \text{lens}) :=$   
 $\text{compl\_exists\_left } (t, m) \ \Delta R \ l_1 \wedge \neg \text{upd\_left } (t, m) \ \Delta R \ l_1$

**Definition** `non_upd_right`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set}) (l_2 : \text{lens}) :=$   
 $\text{compl\_exists\_right } (t, m) \ \Delta R \ l_2 \wedge \neg \text{upd\_right } (t, m) \ \Delta R \ l_2$

### 3.8.2.5 Find existing Entries

In some cases it is necessary to find out if there are any existing entries in the view of the parent lens. This process is not always required, and when it is necessary is explained in the next section. Existing entries have the same key and are queries that are performed on the lenses of the underlying view  $l_2$ . The property `found_any_right` determines if there is a record in the right view  $l_2$  with the exact same join columns  $J$ . There is no corresponding version for the left hand side, since this is not necessary, the reason for this is explained in the next section.

**Definition** `found_any_right`  $((t, m) : \text{change\_entry}) (l_2 : \text{lens}) :=$

$$\text{query\_exists}(\sigma_{\text{match\_on\_expr } t \text{ key}(l_2)}(l_2))$$

A similar property called `found_same_right` is defined which tries to find an record in  $l_2$  with the exact same values for  $V = \text{dom}(l_2)$ . This additional property is defined so that it is possible to determine if the value has changed and needs to be updated, or if it can simply be assumed to be a neutral entry.

**Definition** `found_same_right`  $((t, m) : \text{change\_entry}) (l_2 : \text{lens}) :=$

$$\text{query\_exists}(\sigma_{\text{match\_on\_expr } t \text{ dom}(l_2)}(l_2))$$

Finally, we define a property that indicates that it is an update to the underlying table. This is the case if a record with the same key but which is not identical is found.

**Definition** `found_upd_right`  $((t, m) : \text{change\_entry}) (l_2 : \text{lens}) :=$

$$\text{found\_any\_right } (t, m) \ l_2 \wedge \neg \text{found\_same\_right } (t, m) \ l_2$$

### 3.8.2.6 Removed Entries

We define another property is defined which determines if part of a row has been removed from the output completely. Note that this does not necessarily mean

that the row has to be deleted, as a join removes entries without a matching entry on the right table. Instead it merely states that either the left or the right part would not appear in the output of the lens and thus may be deleted.

In the case of a change to the left table, we assume that any complementary or neutral entries would be included in the table, since the key of the left row uniquely identifies the row. For the right underlying table, there could be entries that refer to the specific row which are not mentioned in the change set at all. In order to determine if such rows exist, the database needs to be queried for any rows with the same join key value. This query is not entirely trivial, since we need to also consider the changes in our change set, which requires us to ignore certain rows in the underlying database.

An example of a change set is shown in Figure 3.5. The table  $T$  is assumed to have the functional dependencies  $A \rightarrow B$ ,  $B \rightarrow C$  and  $C \rightarrow D$  and the last join operation was performed with the two tables  $R: (\underline{A}, B, C)$  and  $S: (\underline{C}, D)$ , making the join key  $C$ .

					$A \rightarrow B:$
					$3 \rightarrow 2$
					$4 \rightarrow 2$
					$5 \rightarrow 3$
$\Delta T$	$A$	$B$	$C$	$D$	
-1	2	1	1	5	
+1	3	2	2	6	
+1	4	2	2	6	$\Rightarrow B \rightarrow C:$
-1	4	1	1	5	$2 \rightarrow 2$
-1	5	3	1	5	$3 \rightarrow 2$
+1	5	3	2	6	
					$C \rightarrow D:$
					$2 \rightarrow 6$

Figure 3.5: An example of a change set and its update set.

The underlying table  $T$  could have an entry  $(A = 6, B = 1, C = 1, D = 5)$  which would prevent the deletion of the right entry  $(C = 1, D = 5)$ . There can not be any change entry with  $m = +1$  and  $C = 1$  since it would show up as a complementary row in that case. This means that if there is any change to some column which defines  $C$ , then it must be ignored since all changes cannot result in a row with



$C = 1$ . For the given example in Figure 3.5 any rows with  $(A = 4)$  must be ignored, since the functional dependency update says that any row with  $A = 4$  will have  $B = 2, C = 2$  and will thus not be relevant. This also applies for  $(A = 3)$  and  $(A = 5)$ . In addition any row with  $(B = 2)$  or  $(B = 3)$  can be ignored since they will either have changed their  $B$  value to one in the update set and therefore must have  $C \neq 1$  or they will only have a new  $C$  value and have  $C \neq 1$ .

The query can be generated by determining all functional dependencies which define the join key  $J$  as  $F$ . We then calculate the update set using the change set, and create an expression which ignores any record which matches the key of any update of a functional dependency in  $F$ . Next we generate an expression for all removed rows which are being deleted and not updated, which are rows that do not have a complementary row. We then take the two expressions and combine them into one expression, along with an additional expression that finds rows with the join key matching the row to be removed.

For the given example, the query would be:

$$C = 1 \wedge \neg(A = 2) \wedge \neg(A = 3) \wedge \neg(A = 4) \wedge \neg(A = 5) \wedge \neg(B = 2) \wedge \neg(B = 3). \quad (3.10)$$

The underlying SQL server could then be queried to determine if any such row exists, which prevents having to load potentially very many rows into memory. The above query could be turned into the following SQL query, which returns a single scalar value indicating if such a row exists or not:

```

SELECT EXISTS (
  SELECT * FROM R, S
  WHERE R.C = S.C AND C = 1
    AND NOT (A = 2)
    AND NOT (A = 3)
    AND NOT (A = 4)
    AND NOT (A = 5)
    AND NOT (B = 2)
    AND NOT (B = 3)
) AS t

```

We first define a function `removed_row_expr` which creates a phrase that finds all rows which have been deleted. The function takes all change entries for which no complementary rows exist and which have the multiplicity  $m = -1$ , and creates an expression which compares the values of the key set. These expressions are combined into a single expression using a conjunction.

**Definition** `removed_row_expr`  $(\Delta R : \text{change\_set}) (key : \text{colset}) :=$

$$\ll \bigvee \llbracket \{\text{match\_on\_expr } t \mid (t, m) \in \Delta R, m = -1, \text{compl\_rows } (t, m) \Delta R \text{ key} = \emptyset\} \rrbracket \gg$$

Next we define the function `find_changed_rows`, which finds all pairs of rows  $(t, t')$ , where  $t$  has been removed and  $t'$  is a complementary row in  $\Delta R$ . We use the previously defined `compl_rows` that finds all complementary entries in  $\Delta R$  for a given change entry  $(t, m)$ .

**Definition** `find_changed_rows`  $(\Delta R : \text{change\_set}) (key : \text{colset}) :=$

$$\left\{ (t, t') \mid (t, m) \in \Delta R, (t', m') \in \text{compl\_rows } (t, m) \Delta R \text{ key}, m = -1 \right\}$$

We also define a function `match_changes_expr`, which takes a complementary row pair  $(t, t')$  and a set of functional dependencies  $fds$  and determines if any of the functional dependencies match on the left but do not match on the right.

**Definition** `match_changes_expr`  $((t, t') : \mathbf{row} * \mathbf{row}) (fds : \mathbf{fundepset}) :=$   

$$\ll \bigvee \llbracket \{ \text{match\_on\_expr } t \text{ left}(fd) \mid fd \in fds, \text{match\_on } t \ t' \text{ left}(fd), \\ \neg \text{match\_on } t \ t' \text{ right}(fd) \} \rrbracket \gg$$

The function `defining_fds` takes a set of columns *key* and a set of functional dependencies and finds all functional dependencies that define the given key. This can be done by taking all functional dependencies for which the transitive closure contains *key*. We assume the function `closure`(*fd*, *fds*) is defined, which calculates the closure of the functional dependency *fd* given the functional dependencies *fds* as a set of columns.

**Definition** `defining_fds`  $(key : \mathbf{colset}) (fds : \mathbf{fundepset}) :=$   

$$\{ fd \mid fd \in fds, key \subset \text{closure}(fd, fds) \}$$

We define a function `changes_expr`, which takes a change set  $\Delta R$ , a set of functional dependencies *F*, and a set of columns *J* and then creates an expression which matches all records which transitively define *J* of any entry in the change set of  $\Delta R$ . The `*` refers to the product type. We make use of the helper function `fd_changes_expr`, which takes a set of changes as  $\mathbf{row} * \mathbf{row}$  tuples, where the columns in the left row define the columns in the right row and a functional dependency *f*. The helper function creates an expression matching any row which has the same values for the columns in `left(f)`. The columns in `left(f)` correspond to the columns in the first entry of each pair tuple in the update set.

**Definition** `fd_changes_expr`  $(changes : (\mathbf{row} * \mathbf{row}) \text{ set}) (f : \mathbf{fundep}) :=$   

$$\ll \bigvee \llbracket \{ \text{match\_on\_expr } chl \text{ left}(f) \mid (chl, chr) \in rows \} \rrbracket \gg$$

**Definition** `changes_expr`  $(\Delta R : \mathbf{change\_set}) (F : \mathbf{fundepset}) (J : \mathbf{colset}) :=$   

$$\ll \bigvee \llbracket \{ \text{fd\_changes\_expr } chl \ f \mid f \in \text{defining\_fds } J \ F, \\ (f, changes) \in \text{get\_updateset } \Delta R \ F \} \rrbracket \gg$$

We combine these helper functions to make the function `create_query_expr`, which creates an expression that matches all entries that are not removed. This is done by creating a conjunction out of three terms. The first term checks for rows that match the column  $J$  for the given record. The second term removes rows which have been marked for deletion in the change set and is constructed using `removed_rows_expr`. The third term removes rows that have been changed in such a way that would affect the column  $J$ , since these rows can safely be ignored.

**Definition** `create_query_expr` ( $t : \text{row}$ ) ( $\Delta R : \text{change\_set}$ ) ( $P_d : \text{colset}$ )

$$\begin{aligned} & (J : \text{colset}) (F : \text{fundepset}) := \\ & \ll \llbracket \text{match\_on\_expr } t \ J \rrbracket \wedge \neg \llbracket \text{removed\_row\_expr } \Delta R \ P_d \rrbracket \\ & \wedge \neg \llbracket \text{changes\_expr } \Delta R \ F \ J \rrbracket \gg \end{aligned}$$

Finally, we can define the properties `remove_entry_left` and `remove_entry_right`. In the case of determining if an entry was removed from the left, we only need to check if the entry is a deletion ( $m = -1$ ) and that there are no complementary rows using `compl_exists_left`. For the right table, we also check for neutral rows since this is an indication we do not need to query the database server and we then query the database server to determine if any rows with the same join key would exist after removal of the row. We assume the function `query_exists( $l$ )` is defined, which converts  $l$  into a SQL query and returns a boolean indicating if  $l$  contains any records. We use `create_query_expr` to generate the expression which we use to filter rows when we query the database for rows that would exist after the join with the same join key  $J$  using `query_exists`. The  $\bowtie_J$  operator joins the two lenses  $l_1$  and  $l_2$  for use with the `query_exists` function.

**Definition** `remove_entry_left`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set})$

$$(l_1 : \text{lens}) :=$$

$$m = -1 \wedge \neg \text{compl\_exists\_left } (t, m) \Delta R l_1$$

**Definition** `remove_entry_right`  $((t, m) : \text{change\_entry}) (\Delta R : \text{change\_set})$

$$(l_1 : \text{lens}) (l_2 : \text{lens}) (J : \text{colset}) (fds : \text{fundepset}) :=$$

$$m = -1 \wedge \neg \text{compl\_exists\_right } (t, m) \Delta R l_2 \wedge \neg \text{ntrl\_exists\_right } (t, m) \Delta R$$

$$\wedge \neg \text{query\_exists}(\sigma_{\text{create\_query\_expr } t \Delta R \text{key}(l_1) J \text{fds}}(l_1 \bowtie_J l_2))$$

### 3.8.3 Case Analysis

The complication in processing the change set  $\Delta R$  is determining how a change in the change set affects each of the individual tables. We define two inductive properties `join_left_row` and `join_right_row` in the following sections and assume that if we call `join_left_row` on all records and flatten the result it will produce a change set that can be used to perform `delta_put` on the left table. Similarly all the outputs of `join_right_row` can be used to produce a change set to be used for the right table. We can define `delta_put_join` as follows:

**Definition** `delta_put_join`  $(l_1 : \text{lens}) (l_2 : \text{lens}) (J : \text{colset}) (P_d : \text{phrase})$

$$(Q_d : \text{phrase}) (F : \text{fundepset}) (\Delta R : \text{change\_set}) :=$$

$$\left( \bigcup_{(t, m) \in \Delta R} \text{join\_left\_row } (t, m) \Delta R l_1 l_2 J F P_d, \right. \\ \left. \bigcup_{(t, m) \in \Delta R} \text{join\_right\_row } (t, m) \Delta R l_1 l_2 J F Q_d \right)$$

Instead of trying to define `join_left_row` and `join_right_row` as a single function, we try to break it down into individual cases in which we can easily reason about the correct behaviour. These cases are defined as individual inductive properties, where assuming certain preconditions hold, the output can be calculated in a specific way. These cases are then combined into a coverage tree as shown in Section 3.8.4 in order to ensure that all inputs are covered.

In all the examples rows are marked with the symbols  $*$ ,  $\checkmark$  and **err**. The example always only shows the output produced by the row marked with a  $*$ , not of any other rows. Rows with an **err** are rows where one can reason that the row cannot exist, and can thus be ignored. Rows with an  $\checkmark$  are assumed to also be in the change set, but their output is not shown in the result of the example.

In all cases  $\pi_U$  is the projection onto the domain of the left table and  $\pi_V$  is the projection onto the domain of the right table.

### 3.8.3.1 Neutral

The first case we consider is the case of a neutral entry as shown in (3.11). In this example the row marked with a  $*$  is considered. Because the  $m = 0$  multiplicity implies that the row may not be changed in any way, there cannot be any other rows with  $m = +1$  or  $m = -1$  with conflicting definitions for the left or right column. There could however be a row with a completely different row on the left, yet an identical row on the right. Such a potential row is shown and marked with a  $\checkmark$ . The resulting entries which should be included by the entry marked with  $*$  is shown in the outputs  $\Delta S$  and  $\Delta T$ .

$$\begin{array}{c|ccccc|c} \Delta R & A & B & C & D & E & \\ \hline 0 & 2 & 1 & 1 & 2 & 1 & * \\ +1 & 2 & 2 & 1 & 2 & 1 & \text{err} \\ +1 & 2 & 1 & 1 & 2 & 2 & \text{err} \\ -1 & 2 & 2 & 3 & 2 & 1 & \text{err} \\ +1 & 3 & 2 & 1 & 2 & 1 & \checkmark \end{array} \xrightarrow[\text{join}]{\text{put}} \begin{array}{c|ccc} \Delta S & A & B & C \\ \hline 0 & 2 & 1 & 1 \\ \Delta T & C & D & E \\ 0 & 1 & 2 & 1 \\ \dots & \dots & \dots & \dots \end{array} \quad (3.11)$$

Thus we can derive the rules:

$$\begin{array}{c} \frac{\text{is\_neutral}(t, m)}{\text{join\_left\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ P_d = \{(\pi_U(t), 0)\}} \\ \frac{\text{is\_neutral}(t, m)}{\text{join\_right\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ Q_d = \{(\pi_V(t), 0)\}} \\ \frac{\text{ntrl\_exists\_right}(t, m) \ \Delta R \ l_2}{\text{join\_right\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ Q_d = \{\}} \end{array}$$

### 3.8.3.2 Updates

The next possible case is that there is an update for the current entry. Update entries are a pair of entries with the same key ( $P_l$  for the left table or  $J$  for the right table) but with non-zero negated multiplicities and differing values for any of the non-key attributes. An example of an update to the right table is (3.12). A complementary example of the left table is (3.13). Note that the output produced is only defined by the entry marked by a \*, the other row would then produce the complementary entry required.

$$\begin{array}{c|ccccc} \Delta R & A & B & C & D & E \\ \hline -1 & 2 & 1 & 1 & 2 & 1 & * \\ +1 & 2 & 1 & 1 & 2 & 2 & \checkmark \end{array} \xrightarrow[\text{join}]{\text{put}} \begin{array}{c|ccc} \Delta T & C & D & E \\ \hline -1 & 1 & 2 & 1 \\ \dots & \dots & & \end{array} \quad (3.12)$$

$$\begin{array}{c|ccccc} \Delta R & A & B & C & D & E \\ \hline +1 & 2 & 2 & 1 & 2 & 1 & * \\ -1 & 2 & 1 & 1 & 2 & 1 & \checkmark \end{array} \xrightarrow[\text{join}]{\text{put}} \begin{array}{c|ccc} \Delta S & A & B & C \\ \hline +1 & 2 & 2 & 1 \\ \dots & \dots & & \end{array} \quad (3.13)$$

This suggests the following rules:

$$\frac{\text{upd\_left}(t, m) \ \Delta R \ l_1}{\text{join\_left\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ P_d = \{(\pi_U(t), m)\}}$$

$$\frac{\text{upd\_right}(t, m) \ \Delta R \ l_2}{\text{join\_right\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ Q_d = \{(\pi_V(t), m)\}}$$

### 3.8.3.3 Non-updates

As can be seen in the example (3.14) it is also possible that a pair of complementary rows do not result in an update. This implies that there cannot be a further entry which contradicts this and it is the equivalent of a neutral row in the output. In order to prevent both the row and complementing row from producing the same output, only the row with a negative multiplicity causes the inclusion of a neutral row in the output. Note that this could still lead to the same neutral entry appearing multiple times in the output, and the additional entries have to be filtered out.

$$\begin{array}{c|ccccc} \Delta R & A & B & C & D & E \\ \hline -1 & 2 & 1 & 1 & 2 & 1 \\ +1 & 2 & 1 & 1 & 2 & 2 \end{array} \xrightarrow[\text{join}]{\text{put}} \begin{array}{c|ccc} \Delta S & C & D & E \\ \hline 0 & 2 & 1 & 1 \\ \dots & \dots & & \end{array} \quad (3.14)$$

We can define the following further rules:

$$\frac{\text{non\_upd\_left}(t, m) \ \Delta R \ l_1 \quad m = -1}{\text{join\_left\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ P_d = \{(\pi_U(t), 0)\}}$$

$$\frac{\text{non\_upd\_left}(t, m) \ \Delta R \ l_1 \quad m = +1}{\text{join\_left\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ P_d = \{\}}$$

$$\frac{\text{non\_upd\_right}(t, m) \ \Delta R \ l_2 \quad m = -1}{\text{join\_right\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ Q_d = \{(\pi_V(t), 0)\}}$$

$$\frac{\text{non\_upd\_right}(t, m) \ \Delta R \ l_2 \quad m = +1}{\text{join\_right\_row}(t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ Q_d = \{\}}$$

### 3.8.3.4 Deletions

Deletions, as defined by Bohannon et al., perform the join and then remove all rows which are there but should not have been. An important consideration is that the right table has to be fully defined by the join key. Given two tables with cars and their owners, the owners would have to be the right table, as the join column would uniquely identify an owner while an owner could have many cars.

In our case we first determine if the entry needs to be removed in either the left or the right part. This is done using the `remove_entry_left` or `remove_entry_right` properties. If an entry has to be removed from the left but not from the right, it would mean that in the given example a car has been removed from the owner, while the owner should not be deleted. As such, the only way to remove the joined row is to remove the entry in the left table completely (which would be the car). An example of this is shown in (3.15).

If on the other hand it has been removed from both the left and the right, then it could be removed from either the left, the right or both tables. In this case, the



decision is deferred to the programmer who specifies two predicates  $P_d$  and  $Q_d$ , where for each row either the one or the other entry has to evaluate to true. If  $P_d$  is true, it is deleted from the left table, while  $Q_d$  causes it to be deleted from the right.

Additionally, there is a slightly more special case if it is removed from the right but not from the left. Consider a table of cars and a table of its owners being joined, where the cars table is the left table and the owners table is the right table. It would be the equivalent of a car not being removed while the owner is being removed. This implies that the owner of the car has been changed, and the previous car owner is not listed in any other entries in the database. In this case the owner is never deleted as this is how the existing relational lenses are defined. Deleting the owner would hypothetically still produce the same output however.

$$\begin{array}{c}
 \begin{array}{c|ccccc} \Delta R & A & B & C & D & E \\ \hline -1 & 2 & 1 & 1 & 2 & 1 & * \\ 0 & 3 & 2 & 1 & 2 & 1 & \checkmark \\ +1 & 4 & 2 & 1 & 2 & 1 & \checkmark \end{array} \xrightarrow[\text{join}]{\text{put}} \begin{array}{c|ccc} \Delta S & A & B & C \\ \hline -1 & 2 & 1 & 1 \\ \cdots & \cdots & & \\ \Delta T & C & D & E \\ \hline 0 & 1 & 2 & 1 \\ \cdots & \cdots & & \end{array} \\
 \end{array} \quad (3.15)$$
  

$$\begin{array}{c}
 \begin{array}{c|ccccc} \Delta R & A & B & C & D & E \\ \hline -1 & 2 & 1 & 1 & 2 & 1 & * \\ +1 & 2 & 1 & 2 & 2 & 1 & \checkmark \end{array} \xrightarrow[\text{join}]{\text{put}} \begin{array}{c|ccc} \Delta S & A & B & C \\ \hline -1 & 2 & 1 & 1 \\ \cdots & \cdots & & \\ \Delta T & C & D & E \\ \hline 0 & 1 & 2 & 1 \\ \cdots & \cdots & & \end{array} \\
 \end{array} \quad (3.16)$$

The rules can formally be defined as follows:

$$\begin{array}{c}
 \text{remove\_entry\_left } (t, m) \Delta R l_1 \quad \text{remove\_entry\_right } (t, m) \Delta R l_1 l_2 J fds \\
 P_d(t, m) \\
 \hline
 \text{join\_left\_row } (t, m) \Delta R l_1 l_2 J fds P_d = \{(\pi_U(t), -1)\} \\
 \\
 \text{remove\_entry\_left } (t, m) \Delta R l_1 \quad \text{remove\_entry\_right } (t, m) \Delta R l_1 l_2 J fds \\
 \neg P_d(t, m) \\
 \hline
 \text{join\_left\_row } (t, m) \Delta R l_1 l_2 J fds P_d = \{\}
 \end{array}$$

$$\begin{array}{c}
\text{remove\_entry\_left } (t,m) \Delta R l_1 \quad \neg \text{remove\_entry\_right } (t,m) \Delta R l_1 l_2 J fds \\
\hline
\text{join\_left\_row } (t,m) \Delta R l_1 l_2 J fds \quad P_d = \{(\pi_U(t), -1)\}
\end{array}$$

$$\begin{array}{c}
\text{remove\_entry\_right } (t,m) \Delta R l_1 l_2 J fds \quad \text{remove\_entry\_left } (t,m) \Delta R l_1 \\
Q_d(t,m) \\
\hline
\text{join\_right\_row } (t,m) \Delta R l_1 l_2 J fds \quad Q_d = \{(\pi_V(t), -1)\}
\end{array}$$

$$\begin{array}{c}
\text{remove\_entry\_right } (t,m) \Delta R l_1 l_2 J fds \quad \text{remove\_entry\_left } (t,m) \Delta R \\
\neg Q_d(t,m) \\
\hline
\text{join\_right\_row } (t,m) \Delta R l_1 l_2 J fds \quad Q_d = \{\}
\end{array}$$

$$\begin{array}{c}
\text{remove\_entry\_right } (t,m) \Delta R l_1 l_2 J fds \quad \neg \text{remove\_entry\_left } (t,m) \Delta R l_1 \\
\hline
\text{join\_right\_row } (t,m) \Delta R l_1 l_2 J fds \quad Q_d = \{\}
\end{array}$$

### 3.8.3.5 Neutral Add

If the user inserts a record, and there no other entries in the change set that refer to the same record, then it is necessary to query the database in order to determine if such a record exists in the unchanged view. The first case that can occur is that a record is added which already exists in the underlying tables. This can be checked by using the `found.same.right` property. If the row is found unchanged in the underlying database, then a neutral entry is added to the change set. An example of this is shown in (3.17).

This can only be the case for a row in the right table, as if there was an entry in the left table, then it would have to be in the output, since it has a foreign key constraint. As such additions without any complements for the right side are always included.

$$\begin{array}{c}
\begin{array}{c|ccc} T & A & B & C \\ \hline & 1 & 2 & 1 \\ & \dots & & \end{array} \\
\begin{array}{c|ccccc} \Delta R & A & B & C & D & E \\ \hline +1 & 2 & 1 & 1 & 2 & 1 & * \end{array}
\end{array}
\begin{array}{c}
\begin{array}{c|ccc} \Delta T & A & B & C \\ \hline 0 & 2 & 1 & 1 \\ \dots & \dots & & \end{array}
\end{array}
\begin{array}{c}
\begin{array}{c} \xrightarrow{\text{put}} \\ \xrightarrow{\text{join}} \end{array}
\end{array}
\quad (3.17)$$

This can be formally expressed using the following rules:

$$\frac{\neg \text{compl\_exists\_left } (t, m) \Delta R l_1 \quad m = +1}{\text{join\_left\_row } (t, m) \Delta R l_1 l_2 J fds P_d = \{(\pi_U(t), +1)\}}$$

$$\frac{\neg \text{compl\_exists\_right } (t, m) \Delta R l_2 \quad m = +1 \quad \text{found\_same\_right } (t, m) l_2}{\text{join\_right\_row } (t, m) \Delta R l_1 l_2 J fds Q_d = \{(\pi_V(t), 0)\}}$$

### 3.8.3.6 New Add

The next possibility is that an entry is added when there is no existing record with a similar key. In this case the record should simply be added to the update list. An example is shown in (3.18).

$$\begin{array}{c|ccc} T & A & B & C \\ \hline \neg & 1 & 2 & 1 \\ & \dots & & \end{array} \xrightarrow[\text{join}]{\text{put}} \begin{array}{c|ccc} \Delta T & A & B & C \\ \hline +1 & 1 & 2 & 1 \\ \dots & \dots & & \end{array} \quad (3.18)$$

$$\begin{array}{c|ccccc} \Delta R & A & B & C & D & E \\ \hline +1 & 2 & 1 & 1 & 2 & 1 & * \end{array}$$

The following rule covers the case of inserting a record if there are no existing underlying records:

$$\frac{\neg \text{compl\_exists\_right } (t, m) \Delta R l_2 \quad m = +1 \quad \neg \text{found\_any\_right } (t, m) l_2}{\text{join\_right\_row } (t, m) \Delta R l_1 l_2 J fds Q_d = \{(\pi_V(t), +1)\}}$$

### 3.8.3.7 Update Add

The final case to consider is where an existing record exists in the underlying table, which has the same key but is not identical. This implies that the row needs to be updated in some form or another. Assuming that the record  $t'$  exists in the table  $T$ , (3.19) shows what the valid output would be for  $\Delta T$ .

$$\begin{array}{c}
\begin{array}{c|ccc}
T & A & B & C \\
\hline
& 1 & 2 & 2 \\
& \dots & & 
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c|ccc}
\Delta T & A & B & C \\
\hline
-1 & 1 & 2 & 2 \\
+1 & 1 & 2 & 1 \\
\dots & \dots & & 
\end{array}
\end{array}
\quad (3.19)$$

$\xrightarrow[\text{join}]{\text{put}}$

$$\begin{array}{c|ccccc}
\Delta R & A & B & C & D & E \\
\hline
+1 & 2 & 1 & 1 & 2 & 2 \quad *
\end{array}$$

$$\frac{\neg \text{compl\_exists\_right } (t, m) \ \Delta R \ l_2 \quad m = +1 \quad \text{found\_upd\_right } (t, m) \ l_2 \quad t' = \text{singleton}(\text{query\_distinct}(\sigma_{\text{match\_on\_expr } t\text{key}(l_2)}(l_2)))}{\text{join\_right\_row } (t, m) \ \Delta R \ l_1 \ l_2 \ J \ fds \ Q_d = \{(\pi_V(t), +1); (t', -1)\}}$$

### 3.8.4 Coverage

One method to give us more confidence in its correctness is to show coverage of all cases. This ensures that if all cases were correct, there are none that are missing. In order to show coverage we depict the path as a tree, where green nodes are termination nodes. For the entire tree to be covered, all leaf nodes need to be termination nodes.

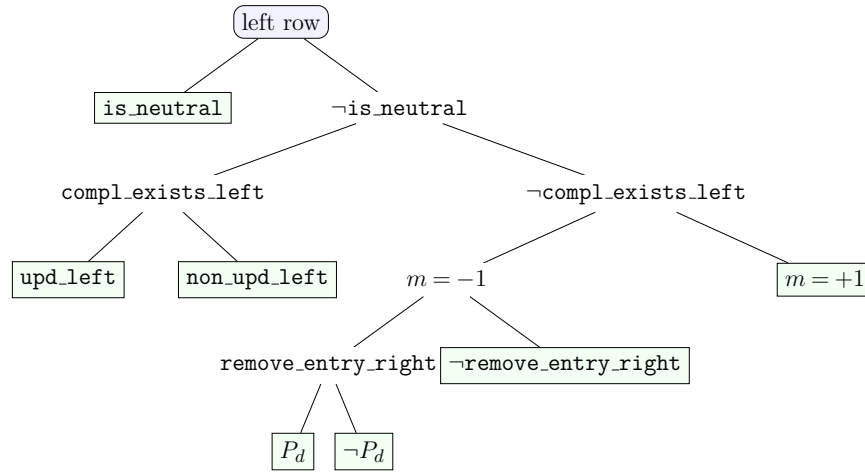


Figure 3.6: Left row coverage tree.

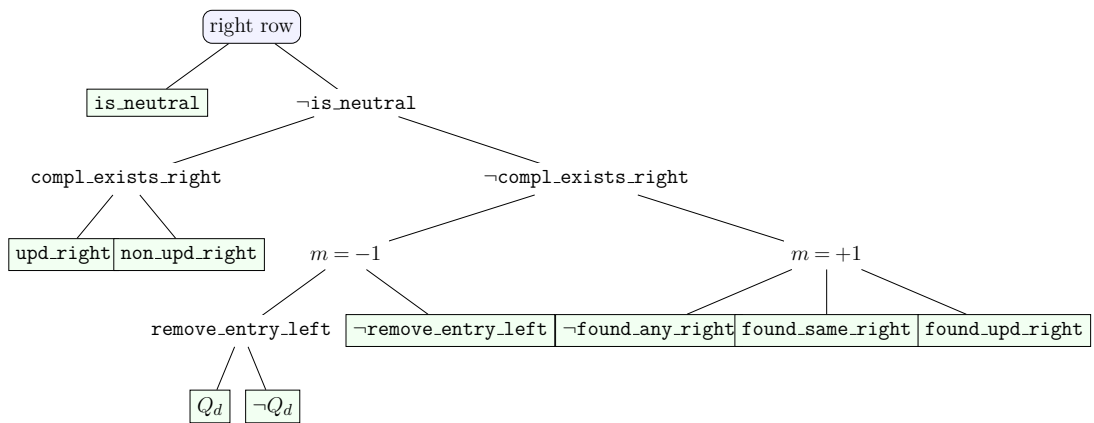


Figure 3.7: Right row coverage tree.

## 3.9 Put Function

### 3.9.1 Table Put

In order to commit changes back to the database it is necessary to convert the change set into a set of SQL commands which can be executed on the database. We make the assumption that the table only contains a single functional dependency in the form  $K \rightarrow V$ , where  $K$  is the set of key attributes and  $V$  is the set of all other attributes.

We first want to define a function in order to create SQL insert statements. An example of an insert statement would be:

```
INSERT INTO myTable (A,B,C)
VALUES (1,2,3), (2,4,5), (3,6,7)
```

We first define some helper functions in order to create some of the phrase elements required. The `comma_separated` function takes a list of phrases and flattens them into a single phrase, where each phrase is separated by commas, e.g.  $[\ll 1 \gg, \ll 2 \gg, \ll 3 \gg]$  gets converted into  $\ll 1, 2, 3 \gg$ .

**Definition** `comma_separated` ( $P : \text{phrase list}$ ) :=

$$\ll [\text{hd}(P)] \{ \ll, p \gg \mid p \in \text{tl}(P) \} \gg$$

The `insert_tuple_expr` takes a row  $r$  and a list of columns  $D$  and converts them into a bracketed tuple as required by an insert statement. For example it converts the row  $(A = 1, B = 2, C = 3)$  with  $D = [B, C]$  into  $\ll (B = 2, C = 3) \gg$ .

**Definition** `insert_tuple_expr` ( $D : \text{col list}$ ) ( $r : \text{row}$ ) :=

$$\ll ([\text{comma\_separated } \{ \ll [r[d]] \gg \mid d \in D \}]) \gg$$

Given these two helper functions it is possible to define the function `insert_expr`, which takes a lens  $l$  and a set of rows  $R$  and creates a single INSERT statement that inserts multiple values.

**Definition** `insert_expr` ( $l : \text{lens}$ ) ( $R : \text{row set}$ )

```
let D = list(dom(R))
<< INSERT INTO  $\llbracket l \rrbracket$  ( $\llbracket \text{comma\_separated } \{\ll C \gg \mid C \in D\} \rrbracket$ )
  VALUES  $\llbracket \text{comma\_separated } \{\text{insert\_tuple\_expr } D \ t \mid t \in R\} \rrbracket$  >>
```

Next we define a function in order to create SQL update statements. An example of an update statement we would like to generate is:

```
UPDATE my_table
SET B = 2, C = 3
WHERE A = 1
```

We first define a helper function which allows us to generate the comma separated key value expressions as used after the SET keyword. The function `key_value_expr` takes a row  $r$  and a column  $D$  and creates the required expression. An example would be the row  $(A = 5, B = 6, C = 7)$  along with the column set  $[B, C]$  which would produce the phrase  $\ll B = 6, C = 7 \gg$ .

**Definition** `key_value_expr` ( $r : \text{row}$ ) ( $D : \text{colset}$ ) :=

```
<<  $\llbracket \text{comma\_separated } \{\ll \llbracket d \rrbracket = \llbracket r[d] \rrbracket \gg \mid d \in D\} \rrbracket$  >>
```

We can use the previously defined `match_on_expr` function in order to produce the predicate after the WHERE keyword along with the `key_value_expr` function to define the `update_expr` as follows:

**Definition** `update_expr` ( $l : \text{lens}$ ) ( $t : \text{row}$ ) :=

```
<< UPDATE  $\llbracket l \rrbracket$  SET  $\llbracket \text{key\_value\_expr } t \text{ (dom}(l) - \text{key}(l)) \rrbracket$ 
  WHERE  $\llbracket \text{match\_on\_expr } t \text{ key}(l) \rrbracket$  >>
```

Next we define an expression for producing delete statements, which can be done using the `match_on_expr`. We define the `delete_expr` function which takes a set

of rows  $R$  and a lens  $l$  and produces a statement that deletes multiple rows at once, as follows:

**Definition** `delete_expr` ( $l : \text{lens}$ ) ( $R : \text{row set}$ ) :=  
 $\ll \text{DELETE FROM } \ll l \gg \text{ WHERE } \bigvee \ll \{\text{match\_on\_expr } t \text{ key}(l) \mid t \in R\} \gg \gg$

Finally we define the function `table_put`, which takes a change set  $\Delta R$  and a lens  $l$  and generates and executes the required SQL statements. All rows with a positive multiplicity are used to produce a SQL update statement, while all rows without a complementary row are used to produce an insert or a delete statement, depending on the multiplicity. We assume the function `execute` exists, which executes a SQL command on the target database server.

**Definition** `table_put` ( $l : \text{lens}$ ) ( $\Delta R : \text{change\_set}$ ) :=  
`execute (delete_expr l {t | (t, -1) ∈ ΔR, compl_rows (t, -1) ΔR key(l) = ∅});`  
`execute (insert_expr l {t | (t, +1) ∈ ΔR, compl_rows (t, +1) ΔR key(l) = ∅});`  
`{execute (update_expr l t) | (t, +1) ∈ ΔR, compl_rows (t, +1) ΔR key(l) ≠ ∅}`

### 3.9.2 Combined Put

With all our lens delta put definitions, as well as the table put definition, it is possible to combine them in order to define a single `delta_put` function, taking a lens  $l$  and a change set  $\Delta R$ . The function determines what type of lens  $l$  is, and calls the corresponding delta put function. If  $l$  is a primitive lens, then it calls the `table_put` function, giving it a change set and committing it to the underlying SQL database.



**Definition** `delta_put`  $(l : \text{lens}) (\Delta R : \text{change\_set}) :=$

```

  match  $l$  with
  | lens  $l \rightarrow \text{table\_put } l \Delta R$ 
  | lensdrop  $l A X a \rightarrow \text{delta\_put } l (\text{delta\_put\_project } l A X a \Delta R)$ 
  | lensjoin  $l_1 l_2 J P_d Q_d \rightarrow$ 
    let  $\Delta R', \Delta R'' = \text{delta\_put\_join } l_1 l_2 J P_d Q_d fd(l) \Delta R$ 
     $\text{delta\_put } l_1 \Delta R'; \text{delta\_put } l_2 \Delta R''$ 
  | lensselect  $l P \rightarrow \text{delta\_put } l (\text{delta\_put\_select } l P \Delta R)$ 

```

### 3.10 Discussion on Correctness

While Section 3.8.4 ensures coverage for the join lens, we don't formally prove correctness. Proving correctness would require a few additional steps that are out of the scope for the work presented here.

The usual way to prove that lenses are correct involves proving that the lenses are well-behaved and satisfy `GetPut` and `PutGet` for all possible inputs. By incrementalizing the lenses this process becomes more complicated because there are three steps involved. The put operation first generates a change set, then applies all lenses to the change set and finally submits the change set to the database. The proof could be slightly simplified by trying to make use of the invariant properties we mention in Section 3.3.

If we know that the  $\xleftarrow{\Delta U}_F$  operation is sufficient to reconstruct tables, it should be possible to prove that the property in Section 3.3.2 applies. Knowing that this invariant property holds may make it possible to prove `GetPut` and `PutOp`. Alternatively it may be possible to prove equivalence with the relational lenses by Bohannon et al. [11], as they have correctness proofs.

We leave this as future work.



# Chapter 4

## Language Integration

Section 3 explains how to incrementalize the existing work on relational lenses. In addition to implementing delta lenses, this work also extends the Links programming language with incremental relational lenses as a language feature. The incremental relational lenses are implemented with similar syntax to the proposal by Bohannon et al.

### 4.1 Types

We assume that the programming language already has built in syntax and semantics for database access, including a **table** type. In addition to the table type, we introduce a **lens** type, which contains the information returned by the **sort** function which was defined in Section 3.5.2. We also assume that there is a polymorphic type **set** and a type **row** which takes a domain. This includes which columns the lens returns, which constraints the lens has, and which functional dependencies the data must adhere to.

$$\begin{aligned} \tau ::= & \dots \mid \textit{domain row} \\ & \mid \textit{type set} \\ & \mid \textit{table } \dots \\ & \mid \textit{lens sort} \end{aligned}$$

## 4.2 Values

In order to simplify the type system, we define a primitive lens wrapper called `lens` which takes an underlying table value. We then define additional values for each of the different lenses.

The projection lens `lensdrop` takes a lens to project onto, a column which it should drop, the key which determines the column, and an additional default value if no other value can be determined. The join lens `lensjoin` takes two lenses  $l_1$  and  $l_2$ , the join key and two predicates  $e_1$  and  $e_2$  which determine if it should try to delete from the left or right table. Finally, the select lens `lensselect` takes an underlying lens and a predicate it should use to filter rows.

$$\begin{aligned}
 v ::= & \dots \mid \text{`lens` } v \\
 & \mid \text{`lensdrop` } v_1 \ x_1 \ x_2 \ v_2 \\
 & \mid \text{`lensjoin` } v_1 \ v_2 \ J \ e_1 \ e_2 \\
 & \mid \text{`lensselect` } v \ e
 \end{aligned}$$

## 4.3 Syntax Rules

This section contains definitions for the syntax rules, which define lens related expressions for the target language. There are two rules corresponding to the `get` and `put` operations. They both take a lens value and perform the corresponding operation on the respective lens. The `put` rule also takes an updated view which should be applied to the database. Then there are expressions to define each of the different lenses.

$$\begin{aligned}
 e ::= & \dots \mid \text{`get` } e \\
 & \mid \text{`put` } e_1 \ \text{to } e_2 \\
 & \mid \text{`lensdrop` } x_1 \ \text{determined by } x_2 \ \text{default } e_2 \ \text{from } e_1 \\
 & \mid \text{`lensselect` from } e \ \text{where } e \\
 & \mid \text{`lensjoin` } e_1 \ \text{with } e_2 \ \text{on } e_3 \ \text{left } e_4 \ \text{right } e_5
 \end{aligned}$$

## 4.4 Context

We assume there are two contexts which are used to model side effects in the program.

The first is the database state, which is denoted by the symbol  $\Theta$ . This is needed because the `lens_put` operation alters the database state into a  $\Theta'$ .

The second is the program context  $\Theta$ , which may contain variables in the current scope. This is needed to evaluate unquasiquoted expressions within a phrase which may refer to other variables or functions.

We assume that an expression  $e$  evaluates to a value  $v$  under the contexts  $\Theta$  and  $\Gamma$  which is written  $\Theta, \Gamma, e \Downarrow \Theta', \Gamma', v_1$ , where  $\Theta'$  and  $\Gamma'$  could be updated contexts. If we don't care about the new states an expression evaluates to within a judgement we also write  $\Theta, \Gamma, e \Downarrow v$ .

## 4.5 Evaluation

We first define three rules, which take lens expressions and convert them into lens values. In all cases it is necessary to evaluate the child lens expressions into a lens value first, and to then construct the new lens value using the child lens values.

The lens expression takes an expression  $e$  which evaluates to a table  $v$  and constructs a lens with  $v$ . While it would potentially be possible to directly use tables, introducing a lens primitive simplifies the type system, as we are able to have a single type for all lenses.

$$\frac{\Gamma, \Theta, e \Downarrow v}{\Gamma, \Theta, \text{**lens** } e \Downarrow \Gamma, \Theta, \text{**lens** } v}$$

The lens drop statement takes two identifiers  $x_1$  and  $x_2$  and a default value  $e_2$ . The identifiers are unchanged, but the expression  $e_1$  is evaluated to determine a constant which is used to construct the lens.

$$\frac{\Gamma, \Theta, e_1 \Downarrow v_1 \quad \Gamma, \Theta, e_2 \Downarrow v_2}{\Gamma, \Theta, \text{**lensdrop** } x_1 \text{ **determined by** } x_2 \text{ **default** } e_2 \text{ **from** } e_1 \Downarrow \Gamma, \Theta, \text{**lensdrop** } v_1 \ x_1 \ x_2 \ v_2}$$

The select lens takes two expressions for the lens and as a filter predicate. The lens expression is evaluated to the lens value. The filter expression is convertible to SQL and contains references to variables which correspond to column values. We use the  $\llbracket \cdot \rrbracket_\Gamma$  notation to allow unquasiquoted expressions in  $\llbracket \cdot \rrbracket$  brackets within a phrase to be evaluated using the context  $\Gamma$  to constant values and then placed in the expression.

$$\frac{\Gamma, \Theta, e_1 \Downarrow v_1}{\Gamma, \Theta, \text{lensselect from } e_1 \text{ where } e_2 \Downarrow \Gamma, \Theta, \text{lensselect } v_1 \llbracket e'_2 \rrbracket_\Gamma}$$

The join lens takes two expressions for child lenses which are both evaluated to values. Then it takes an expression  $e_3$ , which must be in the form of  $(x_1, \dots, x_n)$ , which is converted into a set of columns, corresponding to the join key  $J$ . The function `colset` parses the expression  $e_3$  into a set of columns and is assumed to exist. Finally, the two predicates  $e_4$  and  $e_5$ , which determine where to delete the ambiguous rows, are evaluated using the  $\Downarrow_\llbracket \cdot \rrbracket$  as described above.

$$\frac{\Gamma, \Theta, e_1 \Downarrow v_1 \quad \Gamma, \Theta, e_2 \Downarrow v_2 \quad J = \text{colset}(e_3)}{\Gamma, \Theta, \text{lensjoin } e_1 \text{ with } e_2 \text{ on } e_3 \text{ left } e_4 \text{ right } e_5 \Downarrow \Gamma, \Theta, \text{lensjoin } v_1 \ v_2 \ J \llbracket e_4 \rrbracket_\Gamma \llbracket e_5 \rrbracket_\Gamma}$$

Finally, we define two rules for the `get` and `put` operations. The `get` operation evaluates an expression  $e_1$  as  $v_1$  for a lens and evaluates `lens_get` with  $v_1$ . We assume the function `lens_get` exists, which uses the database state  $\Theta$  to calculate the output.

$$\frac{\Gamma, \Theta, e \Downarrow v_1 \quad v_2 = \text{lens\_get } v_1 \ \Theta}{\Gamma, \Theta, \text{get } e \Downarrow \Gamma, \Theta, v_2}$$

The `put` operation evaluates the expression  $e_1$  as  $v_1$  for the lens and  $e_2$  as  $v_2$  as the changed view. It then executes `lens_put`, which we assume is a computation with global state side effects. The result of the computation is unit, but denoted with an  $*$  to indicate the side effects from the `lens_put` operation.

$$\frac{\Gamma, \Theta, e_1 \Downarrow v_1 \quad \Gamma, \Theta, e_2 \Downarrow v_2 \quad \Theta' = \text{lens\_put } v_1 \ v_2 \ \Theta}{\Gamma, \Theta, \text{put } e_1 \text{ with } e_2 \Downarrow \Gamma, \Theta', ()}$$

## 4.6 Typing Rules

The **get** function takes a lens and returns a set of rows with the columns as specified by  $U$  in the sort of the lens.

$$\frac{\Gamma \vdash e_1 : \mathbf{lens} (U, P, F)}{\Gamma \vdash \mathbf{get} \ e : \mathbf{U row set}}$$

Performing a **put** requires a lens and a set of records. The set of records needs to match the sort type of the lens. The original definition by Bohannon et al. also specifies that the records need to be satisfied by the predicate, but we defer this to runtime evaluation. It may be possible to make use of this in combination with refinement types or similar.

$$\frac{\Gamma \vdash e_1 : \mathbf{lens} (U, P, F) \quad \Gamma \vdash e_2 : U \ \mathbf{row set}}{\Gamma \vdash \mathbf{put} \ e_1 \ \mathbf{with} \ e_2 : ()}$$

A lens expression takes a table and returns something of type lens. The sort for the lens is determined by taking the columns  $U$  and a key from the input table  $K$ . The sort columns are  $U$  and the table is assumed to have the single functional dependency of the key  $K$  determining all other columns  $U \setminus K$ . Bohannon et al. use the top set  $\top$  as the predicate since their work is in set notation. The  $\top$  is replaced with **true** as we use the boolean expression notation.

Due to how the functional dependencies are defined, we assume that all lens expressions can only result in a valid tree form and therefore this is not explicitly checked in each expression as is the case in the original relational lenses paper. This is because we make the assumption that base tables contain data with a single dependency  $K \rightarrow V$ , where  $K$  is the key and  $V$  all other non prime attributes. The only changes to functional dependencies can happen if we join tables, where the right table must be fully defined by the left table, and a projection where a functional dependency is removed. Since each table can only appear once, we assume that functional dependencies always satisfy tree form, as long as the typing rules are upheld.

$$\frac{\Gamma \vdash e : \mathbf{table} (U, K)}{\Gamma \vdash \mathbf{lens} \ e : \mathbf{lens} (U, \mathbf{true}, \{K \rightarrow U \setminus K\})}$$

The lens drop expression takes a lens and returns something of type lens again. The input lens has a set of columns  $U$  and an additional column  $x_1$  of type  $\tau$ , which should be dropped. In order for it to be valid to drop the column  $x_1$ , it needs to be defined by another column  $x_2$ , which is specified by the user. We check this by trying to split the functional dependencies  $F$  into a set  $F'$  and the functional dependency  $x_1 \Rightarrow x_2$ .  $F'$  is then the new set of functional dependencies for the lens sort value. The user additionally defines a default fallback value as the expression  $e_2$ , which should evaluate to type  $\tau$  to match the column.

The original definition defines the predicates in a way that  $P[U - A] \bowtie P[A] = P$ , where  $A = a \in P[A]$  and  $P[U - A]$  is the new sort type. While the exact behaviour has not been studied in depth, it implies that there may not be any interdependency between the two columns, since we can split the predicate  $P$  into a conjunction of two predicates with the two different columns. Since this has not been studied yet, we assume  $P$  does not refer to  $x_1$  and we do not change its definition.

$$\frac{\Gamma \vdash e_1 : \mathbf{lens} (U \cup \{x_1 : \tau\}, P, F) \quad F \equiv F' \cup \{x_2 \rightarrow x_1\} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{lensdrop} \ x_1 \ \mathbf{determined\ by} \ x_2 \ \mathbf{default} \ e_2 \ \mathbf{from} \ e_1 : \mathbf{lens} (U, P, F')}$$

The select lens expression takes a lens  $e_1$  and a predicate  $e_2$ . The select lens only filters out lenses, and therefore does not make any changes to the functional dependencies  $F$  or the columns  $U$ . The expression  $e_2$  is checked under the context of  $\Gamma$  combined with the columns  $U$  and must type check to a boolean value.

The current type checking rules do not assume unquasiquoted expressions, and this is left as future work. In this case the predicate would have to be determined at runtime, where the unquasiquoted parts of the expression can be evaluated in order to produce the final expression. The predicate of the output views sort is set to be the conjunction of the old predicate  $P$  and the expression  $e_2$ .

$$\frac{\Gamma \vdash e_1 : \mathbf{lens} (U, P, F) \quad \Gamma, U \vdash e_2 : \mathbf{bool}}{\Gamma \vdash \mathbf{lensselect} \ \mathbf{from} \ e_1 \ \mathbf{where} \ e_2 : \mathbf{lens} (U, P \wedge e_2, F)}$$

The join lens takes two lens expressions  $e_1$  and  $e_2$ , a join key  $e_3$  expression and two expressions  $e_4$  and  $e_5$  to determine where to delete rows from. The expression  $e_3$  is expected to be a tuple of column names, which can be converted into the



column set  $J$ . The expressions  $e_4$  and  $e_5$  are treated like the predicate of a select lens and must evaluate to a boolean value under the contexts  $\Gamma$  and  $U$ . The relational lenses require that the left lens defines the right lens in practice we swap the two underlying lenses if they are given in the incorrect order.

$$\begin{array}{c}
\Gamma \vdash e_1 : \mathbf{lens} (U, P_1, F_1) \quad \Gamma \vdash e_2 : \mathbf{lens} (V, P_2, F_2) \\
\Gamma, U \vdash e_4 : \mathbf{bool} \quad \Gamma, U \vdash e_5 : \mathbf{bool} \\
J = \mathit{colset}(e_3) \quad F_2 \models J \rightarrow V \\
\hline
\Gamma \vdash \mathbf{lensjoin} \ e_1 \ \mathbf{with} \ e_2 \ \mathbf{on} \ e_3 \ \mathbf{left} \ e_4 \ \mathbf{right} \ e_5 : \mathbf{lens} (U \cup V, (P_1 \wedge P_2), F)
\end{array}$$

## 4.7 Links Example

In this section we show some examples of our actually implemented incremental relational lenses code.

We make use of the existing methods for defining database and table handles. In our running example we work with two tables. The first table is called *categories* and the second table is called *products*. Figure 4.1 shows how a database connection and table definition can be expressed in existing syntax. These tables are modified from the dellstore example database <sup>1</sup>.

In Figure 4.2 we define lenses on our tables. We first define two lens wrappers for the products and categories tables. Note that the implementation requires the specification of table keys. This is because Links allows the specification of multiple table keys in its native syntax. We then define a lens called *productsLens*, which joins the two tables on the common column *category*. This lens is by default a right delete lens, though there is syntax which allows the specification of deletion predicates. After defining the join lens, we define a selection lens filters all rows from the join lens with the predicate *actor* = “CHEVY FOSTER”. We call this selection lens *filterLens*.

With the select lens *filterLens* we can now query the database using the **get** as shown in Figure 4.3. The data can then be changed by the user using any methods and then the **put** can be used to push those changes back to the database.

---

<sup>1</sup>[https://wiki.postgresql.org/wiki/Sample\\_Databases](https://wiki.postgresql.org/wiki/Sample_Databases)

```

var db = database "database" "user" "host:port:user:pass";
var categoriesTable = table "categories"
  with (category : Int, categoryname : String)
  tablekeys [["category"]]
  from db;
var productsTable =
  table "products"
  with (prod_id : Int, category : Int, title : String, actor : String,
        price : Float, special : Int, common_prod_id : Int)
  tablekeys [["prod_id"]]
  from db;

```

Figure 4.1: Defining a database connection and table handles in Links.

```

var productsLens =
  lens productsTable tablekeys (prod_id);
var categoriesLens =
  lens categoriesTable tablekeys (category);
var prodCategoriesLens =
  lensjoin productsLens with categoriesLens on category;
var filterLens =
  lensselect from prodCategoriesLens where actor == "CHEVY FOSTER";

```

Figure 4.2: Defining lenses using our incremental relational lenses extension.

```
var data = get filterLens;  
# modify data to newData  
put filterLens with newData
```

Figure 4.3: Defining lenses using our incremental relational lenses extension.



# Chapter 5

## Evaluation

Existing work by Bohannon et al. show that it is possible to define relational lenses which allow updates to a view to be translated into updates to the underlying relational database. These relational lenses suffer from being inefficient on larger databases in traditional web server configurations, because the entire database needs to be pulled and processed from the database server. The proposed solution is to incrementalize relational lenses, so that changes to a view can be tracked as a change set, which is translated into an easily applicable change set to the underlying database. The previous chapter describes how incremental lenses can be implemented. In this section, we evaluate the execution performance of the defined lenses in order to determine if they can be efficiently used.

All experiments were run on an Intel Core i7-4600U with 8GB RAM. The remote database server has 16GB RAM and an Intel Core i5-6500 CPU.

### 5.1 Number of Input Rows

#### 5.1.1 Setup

We compare our implementation of incremental relational lenses to a naive implementation of relational lenses. In order to measure the performance difference, we create two tables and populate them with random data, the first containing  $n$  rows and the second containing  $n/15$  rows. We then perform a change by taking a column and incrementing its value in the output view, and measure the time

it takes to calculate the updated tables in the case of the naive version, and the change set for the underlying tables in the incremental version. The performance is measured in two values. The first is the total execution time, which is the difference in timestamps before and after calling the put operation. The second is the total query execution time, which is the sum of differences in timestamps before and after each query execution on the database server. The naive version queries each intermediate lens value in order to calculate the output, while the incremental version has to query certain information depending on the lens and the change set. The experiments were performed 5 times and the median of all execution times was chosen. These measurements were performed on a database server which was located on the same machine.

### 5.1.2 Analysis

Figure 5.1 shows the total execution performance depending on the number of rows for both the naive implementation as well as the incremental version. The y-axis is logarithmically scaled, as the naive version is an order of magnitude larger than the incremental version. While the naive version takes over 8 seconds for 10000 rows, the incremental version only needs 26 ms.

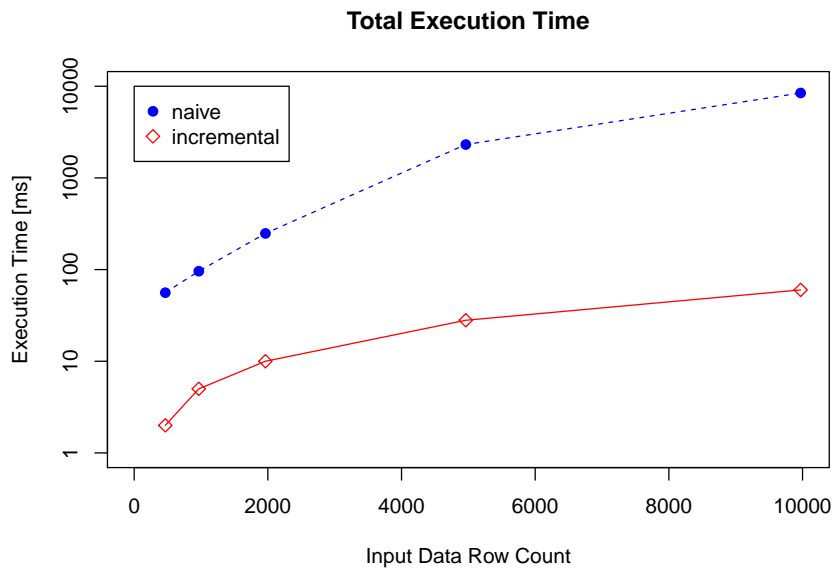


Figure 5.1: The total time it takes for the put operation to complete depending on the number of rows. The y-axis is logarithmically scaled.

Figure 5.2 shows the query execution performance depending on the number of rows. It can be seen that while the naive version takes up to 28ms for querying the database, while the incremental version only spends 2ms querying the database server. This shows that most of the total execution time of the naive version is spent performing local computations. The reason the naive version is so inefficient, is that most operations are implemented as an algorithm with complexity  $O(n^2)$ . While it is possible to implement these more efficiently, part of the problem with the naive version is that one cannot reuse the already efficient database server implementation. In addition the query execution can only be improved by making the database server more efficient, and the query time can be viewed as a lower bound for the performance value. Not only is the incremental version's query execution time lower than the naive query execution time, but the incremental total execution time is also lower than the query execution time of the naive implementation. This shows that incrementalized relational lenses will be more efficient than a heavily optimized naive version.

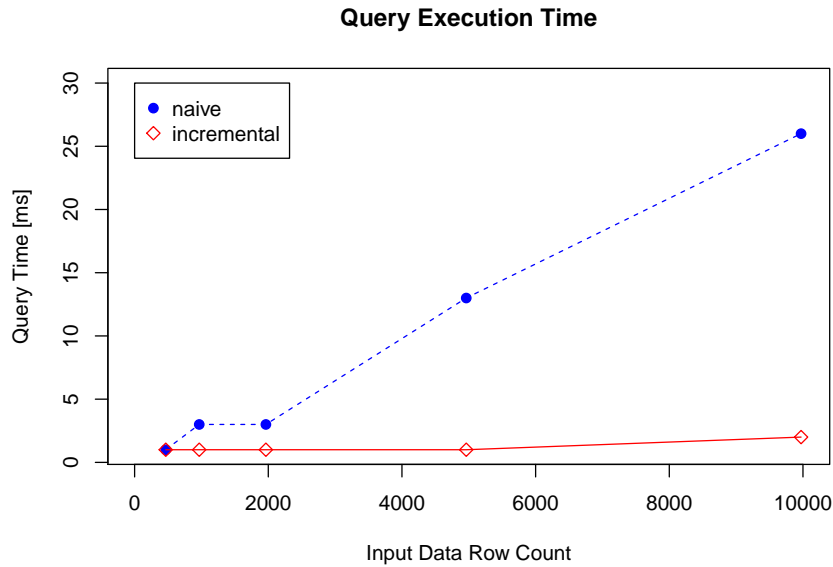


Figure 5.2: The total time spent querying the database to determine the result of put depending on the number of rows.

## 5.2 Number of Tables

### 5.2.1 Setup

In addition to determining how the number of rows affects the output of the lens we also test how the number of input tables affects the performance. We automatically generate  $n$  tables, each with 10000 rows. All  $n$  tables contain three columns,  $p_n$  as the primary key,  $p_{(n+1)}$  as the foreign key to the next table and a column  $c_n$  which contains a random value. All foreign keys are initialized to a value in the range of  $1 \leq i \leq \frac{10000}{15}$ . After performing a natural join on all the lenses, the output is filtered using a select lens, to filter out for  $p_2 = 10$ . Then for each setup we measure both the total execution time as well as the query execution time for removing the first entry as well as for adding a new entry with  $p_1 = 10001$ . Note that for  $n$  tables we have  $n - 1$  joins. All benchmarks were performed using a database server running on a remote machine.

### 5.2.2 Analysis

Figure 5.3 shows the total execution time depending on the number of lenses for both the addition and the removal of an entry. In the case of the addition, we always have  $n + 1$  queries which are executed on the database server. The put operation for an added row takes 6ms for a single table, and 40ms for the same operation with 10 tables. It shows that performance is roughly linear for both operations, with some fluctuation most likely related to background computations. This shows that the performance is acceptable even with a large number of tables.

The removal of a row only requires 2 queries in this example, regardless of the number of underlying tables. As such it can be executed slightly faster than the addition of a row, but it is still only roughly 25% faster, despite using 2 queries instead of 11 queries in the case of 10 tables.

Figure 5.4 shows the query execution time of the same operations. It can be seen that roughly 70% of the total execution time is spent querying the database server. The query time ranges from 5ms to 30ms for the range of 1 to 10 tables. The efficiency is mainly dependent on the database server instead of the local



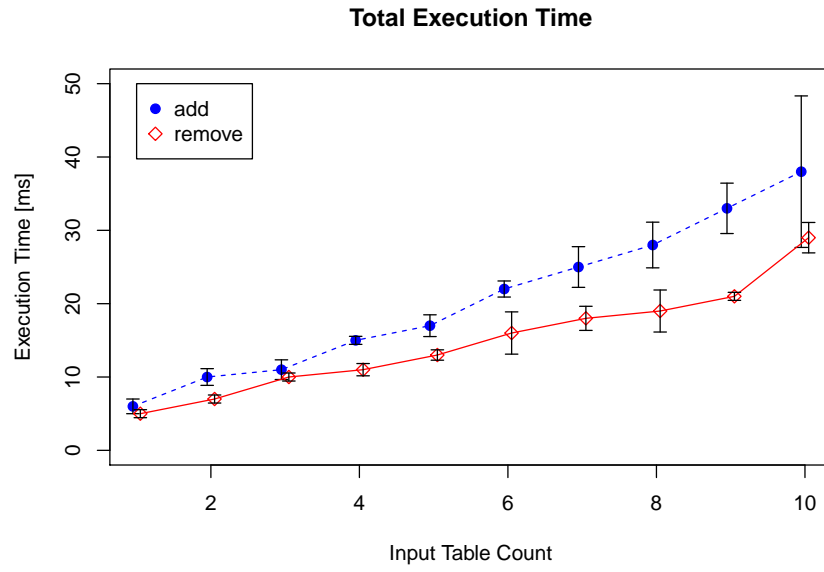


Figure 5.3: The total execution time of a put operation depending on the number of input tables.

machine, and improving the efficiency of the local code will only slightly help the overall performance.

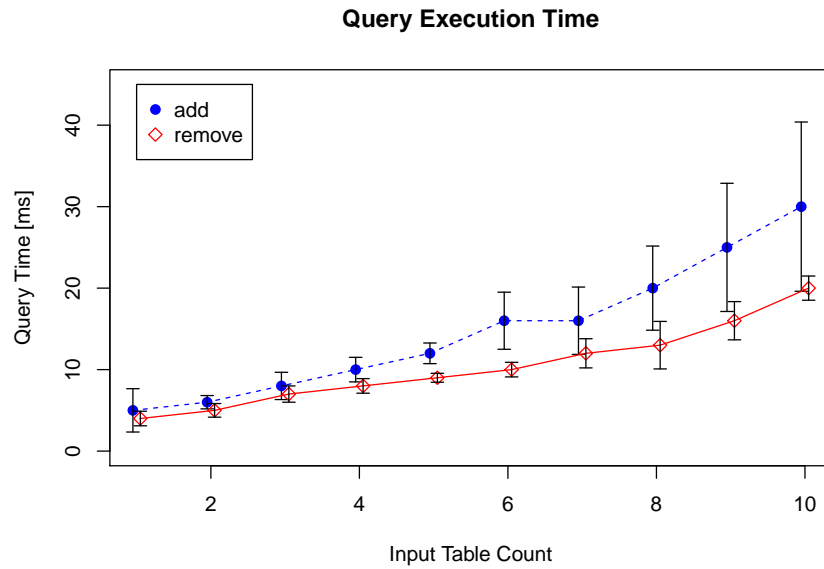


Figure 5.4: The query execution time of a put operation depending on the number of input tables.

This experiment was also run with the naive implementation. The total execution

time for the naive implementation ranged from roughly 35ms to 5min for the different table counts, and the total query time ranged from roughly 14ms to 414ms. This shows that the incremental lenses are more efficient than it would most likely be possible to improve the naive versions efficiency.

# Chapter 6

## Related Work

### 6.1 Edit Lenses

The work presented here tries to solve a problem similar to the one solved by edit lenses [12]. Lenses are typically bidirectional transformations that contain two functions for each direction. The `get` function maps a source data structure  $x \in X$  to a target data structure  $y \in Y$ . In the other direction, the `put` function maps a  $y \in Y$  together with the  $x \in X$  in order to produce an updated  $x' \in X$ .

Edit lenses differ from regular lenses by mapping changes in a data structure  $x \in X$  to changes in a data structure  $y \in Y$ , instead of directly mapping the data structures to each other. Edit lenses try to solve the issue of alignment [13], where lenses can have difficulties matching up information between the two data structures. Edit lenses can also be symmetric bidirectional transformations, where each of the data structures may contain information that the other does not include and so neither of the data structures can be used to fully compute the other one.

The assumption is that edit lenses receive changes in the form of insertions, modifications, deletions and potentially further changes such as reorders and similar. Since these changes are primitive it is easier to map them to changes in a corresponding data structure, since one assumes that between each command the two are kept in sync. An example is a list, where inserting a new record immediately results in the insertion of a record in the right list at the same position.

The incremental lenses as described here also try to keep track of changes, except that the changes are computed by comparing the two views, instead of primitive commands which are applied to the data structure. This means that the changes are much more primitive and essentially limited to insertions and deletions, but this is not necessarily problematic as we assume we are working with unordered relational data. Relational data assumes functional dependencies which have to be kept track of and ensured. Specifically this means that changes to one row can cause changes to another row, and additional steps are necessary to ensure that PutGet and GetPut are not violated.

## 6.2 Incremental View Maintenance

The priority set in the work presented here is to make it feasible to propagate updates in the put direction. The application for this is when the user makes changes to a view and wishes to have his changes applied to the database. This problem also exists in the get direction, where it may be desirable to update materialized views when the underlying database has been changed. Maintaining views is generally desirable for performance reasons, since recalculating an entire view may be expensive, while only changing a few rows may be more efficient.

Like the view update problem incrementally updating views has been a long studied problem. There have been many different formalizations for updating views [14, 15]. A nice modern approach to this problem, has been to view databases from a mathematical perspective, and to consider views as a function for which a delta can be computed [16]. In this approach a view can be described as a function  $f$  which takes a set from  $A$  which is to be changed, and a further set of sets from  $\Theta$  which are also used to compute the view. Given the function  $f$  it is possible to determine the delta function  $\Delta f$ , which takes a set from  $A$ , a set of further sets from  $\Theta$  and a set of changes  $U$ .

$$\begin{aligned} f &: A \times \Theta \rightarrow A' \\ \Delta f &: A \times \Theta \times U \rightarrow A' \end{aligned}$$

We assume the operator  $+$  exists, which takes a set of changes  $u \in U$  and applies

it to the set  $a \in A$ . Similar to a derivative, it then allows us to calculate an updated view using the previous view result as follows:

$$f(x+u, \theta) = f(x, \theta) + (\Delta f)(x, \theta, u)$$

This is similar to how, given the definition of a regular derivative  $f'(x)$ , it is possible to calculate  $f(x+h)$  as follows:

$$f'(x) = \lim_{\mu \rightarrow 0, \mu > 0} \frac{f(x+\mu) - f(x)}{\mu}$$

$$f(x+h) \approx f(x) + h \cdot f'(x)$$

This method can be extended in order to calculate higher order deltas. In addition it can be shown that with each further gradient, the expression becomes simpler and less expensive to compute. This technique makes it possible to show that it is less complicated to update the view using incremental view maintenance than it is to recalculate the entire view.

## 6.3 Language Integrated Query

One important concept related to the practical use of databases is language integrated query [17]. There are different forms of language integrated query, but in general language integrated query extends a programming language with the ability to express queries in the host language, which can be converted into an expression for a target language, e.g. SQL for relational databases, so that the expression can be executed efficiently on the target system. This is usually accompanied by the ability to execute an expression on the local system with the same syntax, meaning that by learning the host language the user already learns how to interact with other systems. An example of a LINQ statement is shown in Figure 6.1.

There are different implementations of language integrated query, the most notable and widespread of which is Microsoft LINQ, which supports querying databases using LINQ to SQL as well as further extensions for parallel computations and

```
var res = from x in l
      where x.Name = 'bob'
      select x.Age
```

Figure 6.1: An example query in the syntax of Microsoft LINQ.

XML views [18]. Kleisli is another example of a collection programming language for biological databases using language integrated query [19]. Furthermore Links itself already supports a form of language integrated query [7].

One important aspect of language integrated query is meta-programming [20]. Meta-programming includes features like quasiquotation which allow language expressions to be parsed as syntax trees which can then be read and modified during execution. Furthermore, these expressions can be compiled back into executable code if required. This is used to take the expression which is used to query the database and to convert it to the target language (e.g. SQL).

Most existing work on language integrated query is more focused on the forward direction and generally only supports a primitive put back system. Microsoft makes use of the Microsoft Entity Framework, which allows table columns to be mapped to class objects [21]. The objects then keep track of any changes made to them, so that when all changes are committed, the changes can be converted to SQL code which is sent to the database server. Newly created objects or deleted objects can be marked for insertion or for deletion.

Links has been extended with syntax similar to SQL for insert, update and delete expressions. Figure 6.2 shows the syntax of how to perform modifications to the database as taken from the developer manual. While both of these approaches are useful, both of them require some understanding of how the changes need to be applied.

**insert**  $t$  **values**  $(f_1, \dots, f_k)$   $rs$

**update** (**var**  $r \leftarrow t$ )

**where**  $condition$

**set**  $(f_1 = v_1, \dots, f_n = v_n)$

**delete** (**var**  $r \leftarrow t$ )

**where**  $condition$

Figure 6.2: The language syntax for insertions, deletions and updates for database entries.





# Chapter 7

## Conclusion

### 7.1 Problem Summary

Existing work by Bohannon et al. introduce the concept of relational lenses, which define bidirectional transformations for relational data [11]. Relational lenses define a set of operations which allow a view to be calculated, and give a correct and well-behaved method of applying an updated view to the underlying data source.

These relational lenses are defined as relational algebra operations however, and cannot easily and efficiently be applied to traditional web server setups, which require communication via SQL. Instead they would require the client to download entire tables and perform costly operations, which quickly become inefficient on larger datasets.

A common method for making interfacing databases easier and safer is LINQ. LINQ which allows database queries to be formulated in native language constructs. The native language constructs can then be converted into efficiently executable SQL [7, 17, 19, 20]. These approaches mainly focus on the aspect of querying databases, and either require the user to formulate changes similar to queries [7] or track the underlying tables as entities and do not allow changes to views [21].

## 7.2 Proposed Solution

The proposed solution is extending the existing work on relational lenses by Bohannon et al. and incrementalizing it so that it keeps track of changes instead of working on the entire set of data, similar to edit lenses [12] or existing work on incremental view maintenance [16]. This allows changes to be handled much more efficiently, since it is only necessary to process smaller amounts of data and any further required data can be queried from the database server. This solution also allows the efficiency of the database server to be reused, without requiring highly optimized relational operations to be implemented by the local application.

These operations are also implemented as language integrated features, which allow the user to define lenses which are typechecked and prevent SQL injection attacks. They are also easier to learn, because the programmer doesn't have to be familiar with SQL and instead only has to learn the host language syntax. In addition, the user can write an application that manipulates the view, without having to worry about how the changes to the view are propagated back to the database. This makes the lenses intuitive and easy to use.

## 7.3 Results

In addition to formalizing incremental relational lenses, they were also implemented as a Links language extension. Using the implementation performance experiments were run and they were compared to naive lenses which queried the entire database from the server and then recalculated the entire underlying table.

The performance experiments show that the naive implementation quickly becomes inefficient, and not only does querying entire views quickly become expensive, but performing the local calculations with naive implementations can quickly require execution times in the magnitude of minutes instead of milliseconds. The incremental version on the other hand, is able to perform updates within tens of milliseconds even in complicated cases of up to 10 joins, requiring only little computation on the local machine.

## 7.4 Future Work

The work presented here is not formally verified and one important step required for practical use would be to ensure that the implementation described here conforms to the existing work on relational lenses. This includes the language integration side, for which progress and preservation could be proven. It would also be good to analyze further complications such as conflicts between updates and how performance could be improved. Additionally a more precise performance model of the current implementation would be helpful to determine how feasible it would be in practical environments. It would also be good to determine how the performance of the queries run on the database behaves.

The work presented by Bohannon also allows different merge operators for the join template, and suggests an additional squash operator which allows for more generic tables, since they do not have the requirement of being in tree form anymore. This operator has not been considered in the work here and has been left as future work.

The language integration syntax is still modelled on the version by Bohannon et al., which is relational lens specific and requires the user to learn the syntax. Links' language integrated query currently makes use of list comprehensions and using similar syntax for defining relational lenses would be more intuitive and less work to learn. Links models joins as nested comprehensions and supports a more powerful syntax, which may not easily be translatable to individual lenses. It would be interesting to combine relational lenses with existing syntax for manipulating tables, where instead of performing the insert / update / delete operation on a table it is performed on the lens. This could be done by converting the changes to a change set and then propagating it through the lens.



# Bibliography

- [1] U. Dayal and P. A. Bernstein, “On the correct translation of update operations on relational views,” ACM Transactions on Database Systems (TODS), vol. 7, no. 3, pp. 381–416, 1982.
- [2] F. Bancilhon and N. Spyratos, “Update semantics of relational views,” ACM Transactions on Database Systems (TODS), vol. 6, no. 4, pp. 557–575, 1981.
- [3] J. N. Foster, B. C. Pierce, and S. Zdancewic, “Updatable security views,” in Computer Security Foundations Symposium, 2009. CSF’09. 22nd IEEE. IEEE, 2009, pp. 60–74.
- [4] P. Stevens, “A landscape of bidirectional model transformations.” GTTSE, vol. 5235, pp. 408–424, 2007.
- [5] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt, “Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem,” ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 29, no. 3, p. 17, 2007.
- [6] M. Hofmann, B. Pierce, and D. Wagner, “Symmetric lenses,” in ACM SIGPLAN Notices, vol. 46, no. 1. ACM, 2011, pp. 371–384.
- [7] E. Cooper, S. Lindley, P. Wadler, and J. Yallop, “Links: Web programming without tiers,” in Formal Methods for Components and Objects. Springer, 2007, pp. 266–296.
- [8] J. Cheney, S. Lindley, G. Radanne, and P. Wadler, “Effective quotation,” 2014.
- [9] J. Cheney, S. Lindley, and P. Wadler, “Query shredding: efficient relational evaluation of queries over nested multisets,” in Proceedings of the 2014 ACM

- SIGMOD international conference on Management of data. ACM, 2014, pp. 1027–1038.
- [10] S. Abiteboul, R. Hull, and V. Vianu, Foundations of databases: the logical level. Addison-Wesley Longman Publishing Co., Inc., 1995.
- [11] A. Bohannon, B. C. Pierce, and J. A. Vaughan, “Relational lenses: a language for updatable views,” in Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2006, pp. 338–347.
- [12] M. Hofmann, B. Pierce, and D. Wagner, “Edit lenses,” in ACM SIGPLAN Notices, vol. 47, no. 1. ACM, 2012, pp. 495–508.
- [13] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce, “Matching lenses: alignment and view update,” in ACM Sigplan Notices, vol. 45, no. 9. ACM, 2010, pp. 193–204.
- [14] X. Qian and G. Wiederhold, “Incremental recomputation of active relational expressions,” IEEE transactions on knowledge and data engineering, vol. 3, no. 3, pp. 337–341, 1991.
- [15] T. Griffin, L. Libkin, and H. Trickey, “An improved algorithm for the incremental recomputation of active relational expressions,” IEEE Transactions on Knowledge and Data Engineering, vol. 9, no. 3, pp. 508–511, 1997.
- [16] C. Koch, “Incremental query evaluation in a ring of databases,” in Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. ACM, 2010, pp. 87–98.
- [17] J. Cheney, S. Lindley, and P. Wadler, “A practical theory of language-integrated query,” ACM SIGPLAN Notices, vol. 48, no. 9, pp. 403–416, 2013.
- [18] D. Kulkarni, L. Bolognese, M. Warren, A. Hejlsberg, and K. George, “Linq to sql: .net language-integrated query for relational data,” MSDN .NET Framework Developer Center (<http://msdn.microsoft.com/enus/library/bb425822.aspx>), 2007.
- [19] L. Wong, “Kleisli, a functional query system,” Journal of Functional Programming, vol. 10, no. 1, pp. 19–56, 2000.

- [20] D. Syme, “Leveraging .net meta-programming components from f#: integrated queries and interoperable heterogeneous execution,” in Proceedings of the 2006 workshop on ML. ACM, 2006, pp. 43–54.
- [21] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar, “Anatomy of the ado. net entity framework,” in Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007, pp. 877–888.