Perm: Efficient Provenance Support for Relational Databases

DISSERTATION

FOR THE DEGREE OF A
DOCTOR OF INFORMATICS

AT THE FACULTY OF ECONOMICS
BUSINESS ADMINISTRATION AND
INFORMATION TECHNOLOGY
OF THE
UNIVERSITY OF ZURICH

by
BORIS GLAVIC
from
Germany

Accepted on the recommendation of
PROF. DR. MICHAEL BÖHLEN
PROF. DR. GUSTAVO ALONSO

2010

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zürich, April 14. 2010[1]

The Vice Dean of the Academic Program in Informatics: Prof. Dr. Harald Gall

[1] Date of the graduation

Abstract

In many application areas like scientific computing, data-warehousing, and data integration detailed information about the origin of data is required. This kind of information is often referred to as *data provenance*. The provenance of a piece of data, a so-called *data item*, includes information about the source data from which it is derived and the transformations that lead to its creation and current representation. In the context of relational databases, provenance has been studied both from a theoretical and algorithmic perspective. Yet, in spite of the advances made, there are very few practical systems available that support generating, querying and storing provenance information (We refer to such systems as *provenance management systems* or *PMS*). These systems support only a subset of SQL, a severe limitation in practice since most of the application domains that benefit from provenance information use complex queries. Such queries typically involve nested sub-queries, aggregation and/or user defined functions. Without support for these constructs, a provenance management system is of limited use. Furthermore, existing approaches use different data models to represent provenance and the data for which provenance is computed (*normal data*). This has the intrinsic disadvantage that a new query language has to be developed for querying provenance information. Naturally, such a query language is not as powerful and mature as, e.g., SQL.

In this thesis we present *Perm*, a novel relational provenance management system that addresses the shortcoming of existing approaches discussed above. The underlying idea of *Perm* is to represent provenance information as standard relations and to generate and query it using standard SQL queries; ”Use SQL to compute and query the provenance of SQL queries”. *Perm* is implemented on top of *PostgreSQL* extending its *SQL* dialect with provenance features that are implemented as query rewrites. This approach enables the system to take full benefit from the advanced query optimizer of *PostgreSQL* and provide full SQL query support for provenance information.

Several important steps were necessary to realize our vision of a ”purely relational” provenance management system that is capable of generating provenance information for complex SQL queries. We developed new notions of provenance that handle SQL constructs not covered by the standard definitions of provenance. Based on these provenance definitions rewrite rules for relational algebra expressions are defined for transforming an algebra expression q into an algebra expression that computes the provenance of q (These rewrites rules are proven to produce correct and complete results). The implementation of *Perm*, based on this solid theoretical foundation, applies a variety of novel optimization techniques that reduce the cost of some intrinsically expensive provenance operations. By applying the *Perm* system to schema mapping debugging - a prominent use case for provenance - and extensive performance measurements we confirm the feasibility of our approach and the superiority of *Perm* over alternative approaches.

Zusammenfassung

Viele Anwendungsgebiete, wie zum Beispiel Wissenschaftliche Berechnungen, Data-Warehousing und Datenintegration, benötigen detaillierte Informationen über die Herkunft von Daten. Solche Informationen werden oft als *Data Provenance* bezeichnet. Die Herkunft eines so genannten Datenelements, beinhaltet Informationen über die Eingabedaten von denen das Datenelement abgeleitet wurde und die Transformationen die zu seiner Entstehung und aktuellen Darstellung beigetragen haben. Provenance für relationale Datenbanken ist sowohl theoretisch als auch in algorithmischer Hinsicht untersucht worden. Trotz der Fortschritte auf diesem Gebiet, muss ein Mangel an praktisch einsetzbaren Systemen konstatiert werden, die die Erzeugung und Speicherung von Provenance und Anfragen über solche Informationen unterstützen (Im Folgenden bezeichnen wir solche Systeme als *Provenance Management Systeme* oder kurz *PMS*). Herkömmliche Systeme unterstützen nur Teilmengen der Sprachkonstrukte von SQL, was eine erhebliche Einschränkung der Praxis-Tauglichkeit dieser Systeme darstellt, da die meisten Anwendungsgebiete, die von Provenance Funktionalität profitieren, komplexe Anfragefunktionalität benötigen. Dazu gehören zum Beispiel geschachtelte Unter-Anfragen, Aggregation und vom Benutzer definierte Funktionen. Ein *PMS*, das diese Sprachkonstrukte nicht unterstützt, ist nur von sehr eingeschränktem Nutzen. Die meisten Ansätze benutzen unterschiedliche Datenmodelle zur Repräsentation von Provenance Informationen und der Daten für die Provenance berechnet wurde (*normale Daten*). Dies hat den unvermeidbaren Nachteil, das eine neue Anfragesprache entwickelt werden muss, um Provenance Daten abfragen zu können. Es ist nicht verwunderlich, das die Mächtigkeit und Reife solcher Sprachen nicht an die einer langzeitig entwickelten Anfragesprache wie SQL heranreicht.

In dieser Dissertation stellen wir das innovative *PMS Perm* vor, das die oben genannten Nachteile von herkömmlichen Systemen behebt. Die dem Ansatz zugrundeliegende Idee ist es, Provenance Informationen als normale Relationen darzustellen, die mit Hilfe von Standard SQL Anfragen generiert und angefragt werden; "Benutze SQL um die Provenance von SQL Anfragen zu berechnen und anzufragen". *Perm* wurde basierend auf *PostgreSQL* umgesetzt und erweitert den SQL Dialekt dieses Systems mit neue Sprachkonstrukten für die Berechnung von Provenance. Diese Sprachkonstrukte sind intern als Anfragetransformationen (*query rewrites*) realisiert. Dieser Ansatz ermöglicht es *Perm* von dem fortschrittlichen Anfrageoptimierer von *PostgreSQL* zu profitieren und ermöglicht den Einsatz von SQL als Anfragesprache für Provenance Informationen.

Die Umsetzung unserer Vision eines "rein relationalen" *PMS* erfolgte in mehreren Schritten. Zunächst war die Entwicklung von neuen Provenance Definitionen notwendig, um SQL Konstrukte zu unterstützen, die von den Standard Provenance Definitionen nicht behandelt werden. Basierend auf diesen Definitionen haben wir Anfragetransformationen entwickelt, die eine Anfrage q in eine Anfrage überführen, die die Provenance von q berechnet. Diese Anfragetransformationen sind beweisbar korrekt und vollständig. Die Implementierung von *Perm* baut auf diesem soliden theoretischen Hintergrund auf. In der Implementierung werden neuartige Optimierungstechniken eingesetzt, um die Effizienz der inhärent aufwändigen Provenance Berechnung zu steigern. Der erfolgreiche Einsatz von *Perm* zur Fehlerdiagnose für Datenintegration - einem weit verbreiteten Einsatzgebiet von Provenance - und umfangreichen Experimente zur Analyse der Effizienz der Provenance Berechnung in *Perm* belegen die Vorteile unseres Systems gegenüber alternativen Ansätzen.

Acknowledgments

First I like to thank my advisors Gustavo Alonso, Michael Böhlen, and Klaus R. Dittrich for supporting me in writing this thesis and conducting the research reported in this thesis. First and foremost, Gustavo, thanks for spending so much time on discussions about my work, pushing me to fully implement my ideas, being patient with my slow process of writing, and opening up opportunities to work with interesting people. Michael, thanks for being willing to step in as a adviser at such late stage of my dissertation and spending the time and effort to get accomplished with my work.

I also would like to thank the guys at the University Zürich Database Technology Research Group and ETH Systems Group for many helpful discussions and funny social activities like hiking, skiing, drinking, and bowling to name just a few.

I thank my family and friends. Especially my parents for contributing to my existence.

Last but not least, I thank my co-authors: Gustavo Alonso, Klaus R. Dittrich, Renée Miller, Laura Haas, Nesime Tatbul, Kyumars Sheykh Esmaili, Peter Fischer, Ira Assent, Ralph Krieger, and Thomas Seidl.

I would like to dedicate this thesis to Klaus who passed away in late 2007. Klaus advised me in the preliminary research that led to the work reported in this thesis and is the person who got me interested in provenance in the first place.

Contents

List of Figures

List of Theorems

List of Definitions

Chapter 1

Introduction

In many application domains like scientific computing, data-warehousing, data integration, curated databases, grid-computing, and workflow management detailed information about the origin of data is required. A large portion of data generated and stored by these application domains is not entered manually by a user, but is derived from existing data using complex transformations. Understanding the semantics of such data and estimating its quality is not possible without having extensive knowledge about the data's origin and the transformations that were used to create it. This kind of information is often referred to as *data provenance* or data lineage. Data provenance is information that describes how a given piece of data, called *data item*, was produced. The provenance includes source and intermediate data as well as the transformations involved in producing the concrete data item. In the context of a relational databases, the source and intermediate data items are relations, tuples and attribute values. The transformations are SQL queries and/or functions on the relational data items. For the aforementioned application domains and, in general, for every application domain where data is heavily transformed, data provenance is of essential importance. Provenance information can be used to estimate the quality of data and its trustworthiness, to gain additional insights about it or to trace errors in transformed data back to its origins.

This thesis presents a novel provenance management system called *Perm* (Provenance Extension of the Relational Model) and its formal foundations. *Perm* is capable of computing, storing and querying provenance for relational databases. The system is implemented on top of *PostgreSQL* extending its *SQL* dialect with provenance features that are implemented as query rewrites.

1.1 Motivation

Efficiently generating, querying and storing relational provenance information is important in a broad range of application domains and has been studied in a variety of contexts. Most of the approaches for provenance define it as information that answers the following question: Given a query q and a tuple t from the result of q, which are the tuples from the relations accessed by q that caused t to appear in the result of q. Thus, provenance describes a relationship between the input and output data of a query q (or other type of transformation in the general case) [1]. Currently, a user who wants to retrieve provenance information has to create it manually, which is a labour-intensive and error-prone approach. Several research prototypes have been developed that support a user in this process by generating provenance automatically for SQL queries. These systems support only a subset of SQL, a severe limitation in practice since most of the application domains that benefit from provenance information use complex queries. Such queries typically involve nested sub-queries, aggregation and/or user defined functions. Without support for these constructs, a *provenance management system* is of limited use.

In general, the provenance of a given tuple t includes all the source tuples used as input for the query that created t. However, to consider the complete input of a query as the provenance of its output is often misleading and contra-intuitive. As a general rule, a certain part of the input influences only part of the output. Given that provenance information tends to be quite large, it is important to narrow down exactly

[1] Some approaches define provenance in a different way. This is discussed in chapter 2.

which input data *contributes* to which output data. Different interpretations of what *contributes* actually means have been presented. We refer to a notion that defines which subset of the input of a transformation belongs to the provenance of a given part of the output as *contribution semantics*. For instance, widely adopted *contribution semantics* are *Why*-provenance [BKT01], *Where*-provenance [BKT01] and *Lineage* [CWW00]. These *contribution semantics* are only applicable for small subsets of SQL or are not applicable at all for SQL, because they cannot handle features like bag semantics.

Mere storage and generation of provenance information is not very useful, if no query facilities for provenance are provided. Existing approaches capture provenance information by extending the underlying data model. This has the intrinsic disadvantage that the provenance must be stored and accessed using a different data model than the actual data. In consequence, provenance query support is very limited, because a new query language for provenance has to be developed and implemented. Even more important, it is not possible to use provenance and normal data in the same query to, e.g., explore their relationship to each other.

1.2 The *Perm* Approach

In the motivation we have outlined the shortcomings of existing provenance management systems. Our approach to overcome this shortcomings presented in this thesis can be summarized in one sentence as ”Use SQL to compute and query the provenance of SQL queries”. What do we mean by that? We envision a system that represents the results of a query alongside with its provenance in a single relation using a representation of this information that is directly interpretable by the user. Even more, provenance should not only be represented relationally, also the generation of this kind of information should be implemented as SQL queries. These goals were not chosen randomly, but are based on an analysis of the causes for the shortcomings of existing systems. As mentioned before these systems tend to use a different data model for provenance and *normal information* (The data stored in the database or generated by queries). This has the intrinsic disadvantage that it is not possible to query provenance and normal data using the same query language. Thus, the association between information and its provenance cannot be explored declaratively. Furthermore, a new query language has to be developed for provenance which naturally is not as sophisticated as a language like SQL that has been in use for several decades and constantly evolved over time. These problems can be avoided by representing provenance and normal data in the same data model and explicitly modeling the associations between normal and provenance data. The motivation for our goal to compute provenance as SQL queries is based on a similar argument. Using the same computational method for querying data and computing provenance allows for a tighter integration of provenance functionality into SQL. For instance, this allows to express a query over the result of a provenance computation as a single SQL statement by using the provenance computation as a sub-query.

To realize our vision of a ”purely relational” provenance management system with *Perm* several obstacles had to be overcome.

1.2.1 Contribution Semantics for Complex Queries

The quality of provenance information generated by a provenance management system directly depends on the *contribution semantics* that is used to derive provenance information. As mentioned before, a *contribution semantics* defines which tuples from a query's input belong to the provenance of an output tuple of the query. Cui el al. presented an intuitive *contribution semantics* for relational algebra expressions in [CWW00] (often called *Lineage*). Unlike other *contribution semantics*, *Lineage* is also defined for set operations and aggregation and, therefore, is a good candidate to become the *contribution semantics* of *Perm*. However, *Lineage* does not extend well to negation and nested sub-queries. Furthermore, the representation of provenance information applied by this *contribution semantics* is not suited for our approach. Therefore, we developed *Perm-Influence contributions semantics* (*PI-CS*), a new type of *contribution semantics* based on *Lineage*, that uses a different representation of provenance and solves the problems of *Lineage* concerning negation and nested sub-queries. Based on *PI-CS* we present the novel notion of *transformation* provenance that models which parts of a query contributed to a result tuple.

1.2.2 Relational Representation of Provenance Information

In *Perm*, if the provenance of a query q is requested by a user, a single relation is returned that contains both the original query results of q and its provenance. Such a relation is constructed by taking the original result tuples and extending them with contributing tuples from the base relations accessed by the query. Thus, a single tuple in this representation associates an original result tuple of the query with tuples in its provenance. This approach enables functionality that is not found in other systems. For instance, the single relation approach for results and provenance allows users to query provenance information using SQL to drill down and identify concrete dependencies, something not supported by most systems.

1.2.3 Provenance Computation Through Query Rewrite

In *Perm*, the provenance of a relational algebra expressions q according to *PI-CS* is computed by using algebraic rewrite rules to transform q into an algebra expression q^+ that propagates the provenance alongside with the original query results. The result of q^+ is the relational representation of provenance discussed above. Note that q^+ is expressed in the same algebra as q. Besides the ”Use SQL to compute and query provenance of SQL queries” advantage, representing provenance as algebraic rewrite rules has the additional advantage that it is possible to formally prove the correctness and completeness of the provenance computation. I. e., the rewrite rules are guaranteed to generated provenance according to our *contribution semantics* definition.

1.2.4 DBMS Integration

Perm has been fully implemented as an extension of *PostgreSQL*. The SQL dialect of *PostgreSQL* is enriched with new language constructs for triggering and controlling provenance computations. We refer to the resulting language as *SQL-PLE* (*Provenance Language Extension*). To be able to apply the algebraic rewrites to SQL queries the rewrite rules have been translated into SQL. The provenance rewrites are applied to the internal query tree representation of *PostgreSQL* generated by the *Parser* of this system. A rewritten query, being represented in the same query tree representation, is executed by the unmodified execution engine of the underlying database system. Hence, *Perm* benefits from the advanced query optimization techniques applied by *PostgreSQL*. Furthermore, this approach enables a user to store provenance information in a relation or view and use it as a sub-query.

Perm is the first *PMS* capable of computing the provenance of the (almost) complete set of SQL query constructs. For instance, *Perm* can be used to compute the provenance of all queries from the *TPC-H* decision support benchmark. Additional features found in Perm include the ability to perform partial provenance computation (e.g., to know the contributing tuples from a view rather than from base relations, a feature also found in Trio [Wid05] and [CW00a]) and support for provenance generated manually by the user or by another provenance management system (*external provenance*).

Of particular interest is the fact that *Perm* supports all these features rather than only a subset of them like existing systems. We integrated a variety of novel optimization techniques that reduce the cost of some intrinsically expensive provenance operations into the system.

In summary, the major contributions of this thesis are:

- We present *PI-CS*, a new type of *contribution semantics* with a relational representation of provenance information that extends traditional *contribution semantics* with support for complex SQL constructs like nested sub-queries. Based on this *contribution semantics* we present the novel notion of *transformation* provenance.
- We demonstrate how provenance can be computed according to this *contribution semantics* using algebraic rewrites and prove the completeness and correctness of these rules.
- We present the implementation of the *Perm* system that integrates the algebraic rewrites into *PostgreSQL* and provides full SQL support for querying provenance.

Some of the material presented in this dissertation has been published previously. A survey on data provenance was presented in [GD07]. The underlying idea of the *Perm* approach and algebraic rewrite rules have been published in [GA09a]. [GA09b] extended this work with extensions of the *contribution semantics* and rewrite rules for nested sub-queries. A demonstration of the *Perm* system has been given at the *SIGMOD* 2009 demonstration track [GA09c].

1.3 Thesis Outline

The outline of this thesis is as follows. In chapter 2 we give an overview and comparison of research approaches in the area of data provenance, introduce a consistent terminology for provenance concepts, and formulate requirements for a fully fledged provenance management system. We present two comparisons in this chapter. Firstly, *contribution semantics* definitions are compared according to their expressiveness. Secondly, existing provenance management systems are compared regarding the functionality they provide. The theoretical foundation of *Perm* is presented in chapters 3 and 4. Chapter 3 introduces the *contribution semantics* applied by *Perm*. This *contribution semantics* translates the widely used *Lineage contribution semantics* to a relational representation and solves the problems of this definition with nested sub-queries and negation. The query rewrite rules that generate queries for computing provenance with respect to this *contribution semantics* are presented in chapter 4. In addition we prove the completeness and correctness of the rewrite rules in this chapter. Having established the formal background of *Perm*, chapter 5 discusses the implementation of the system. The performance of *Perm* is evaluated by extensive performance measurements in chapter 6. Chapter 7 analyzes the practical impact of the system by means of applying it to schema mapping debugging, a standard use case for provenance. We conclude and present avenues for future work in chapter 8.

Chapter 2

Related Work and Terminology

In this section we review related work in the area of data provenance, discuss *contribution semantics* for provenance in database systems, and analyze and compare the functionality provided by current *provenance management systems* (PMS). We introduce our terminology for provenance concepts. This terminology differs from terminology presented in the literature if we deem that there is confusion about the name of a concept or the common terminology is not precise enough. *Data provenance*, also called *lineage* or *pedigree*, is information about the origin and creation process of data. Provenance information is used to understand the semantics of processed data, tracing errors from the result of a transformation back to its input data, and estimating the quality of derived data.

Data provenance is of essential importance in many application domains, especially for application domains where data is heavily transformed. While the focus of this work is not to discuss every possible application domain in detail, we still provide a short overview of the most important ones. Scientists in the fields of biology, chemistry and astronomy use so-called *curated* databases. Curated databases contain data that is not the direct result of measurements from real world experiments, but has been manually modified, annotated and transformed. Examples for curated databases are *GenBank* [BB88] and *Swiss-Prot* [BA97]. For a scientist using a *curated database* it is hard to estimate the quality of such heavily processed data. Provenance information is needed to understand from which source data a data item is derived and which transformations were applied to the source data to produce the current version of the data item. The importance of provenance for *geographical information systems* (*GIS*) has been recognized early on (e.g., see [Lan89]). Several meta-data standards that include provenance have been developed in this field. Workflow management systems and grid-computing are gaining interest from scientific communities, because they enable scientists without computer science background to compose data processing workflows from transformations developed for their field of research. Workflow systems provide support for data provenance to allow for the re-execution of workflows, document the origin of data generated by a workflow, and provide debugging facilities for workflows. Data-warehouses integrate data from different sources and with different data representations. Views are defined over the integrated data to simplify the retrieval of information. Provenance is needed to trace errors in the result of a view result back to erroneous source data and to understand errors in the process of cleaning and integrating data (*Extract-Transform-Load* or *ETL*). In data integration and data exchange schema mappings, declarative descriptions of the relationships between schemas, are used to transform data from one schema into another and to rewrite queries over one schema into queries over another schema. The generation and execution of such mappings is a semi-automatic and error-prone process. Providing provenance can ease the debugging of data integration processes and aid a user in understanding complex schema mappings. Provenance can be used for different purposes in the presented application domains:

- **Error Tracing**: Errors in the result data of a transformation are caused either by errors in the input data and/or errors in the transformation. Provenance enables a user to trace errors in derived data back to erroneous source data and in some cases to distinguish between errors caused by an incorrect transformation and errors caused by faulty source data (if (s)he can determine if the source data is correct).

- **Quality Estimation**: Estimating the quality of heavily transformed data is a non-trivial task. The quality of data produced by a transformation depends on the quality of the input data and the properties of the applied transformation. Provenance information can ease the quality estimation, if the quality of the input data is known to the user.

- **Understanding Derived Data**: Provenance can be used to gain a better understanding of the semantics of data generated by multiple transformations steps. In complex settings the semantics of derived data are often fuzzy and incomprehensible without examining the origin of the processed data and transformations that were applied to derive it.

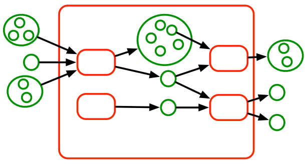


Figure 2.1: Abstract View of Data Manipulations

2.1 Terminology

In an abstract view arbitrary data manipulations can be modeled as a set of *transformations* that access *input* data and produce *output* data. E.g., a *transformation* could be an SQL query, a program written in C, or a workflow in a workflow management system. Typically, the inputs and outputs of a transformation exhibit a certain structure. We use the term *data item* to refer to a *structural unit* of data. *Data items* are often hierarchically structured in the sense that a more complex data item is composed of smaller, less complex data items. We use the term *granularity* to specify a certain level in such hierarchies. For example, in the relational data model a *relation* is a data item that contains several *tuple-granularity* data items. Not only data, but also transformations can be hierarchically structured (e.g., a relational algebra statement is composed by combining relational algebra operators). Therefore, the notion of *granularity* is also applicable to transformation. Figure 2.1 presents an abstract example of these concepts. Data items (dots) of various granularity are processed by a transformation (rectangles) that is composed of several simpler transformation parts. As mentioned above, *Data Provenance* (or short) *provenance* is information about the origins of a data item and the *transformations* that were used to derive it. Therefore, two types of information are generally considered to form the provenance of a result data item d:

- Information about the data items from which d is derived.
- Information about the transformations that generated d.

In [Tan04] these types of provenance were called *provenance of data* and *provenance of a data product*. We use the notions *data provenance* and *transformation provenance*.

2.1.1 Data Provenance

Data provenance is information about the input data items that were used to create (*contributed* to) an output data item d. *Data provenance* approaches can be classified according to which input data items are considered to belong to the provenance of d, which we refer to as *contribution semantics* (*CS*). Several definitions of *contribution semantics* have been proposed in the literature. Here we present a higher level classification of *CS* types and postpone the discussion of concrete definitions to section 2.3. We consider a *CS* definition to belong to the class of *input-CS* (*IN-CS*) if it considers all input data items of a transformation to belong to the provenance of an output of this transformation. Instances of this class of *CS* are easy to define and compute, because no knowledge about the inner workings of a transformation is required. *IN-CS* is used by provenance-aware workflow systems and in grid-computing approaches where transformations are modeled as black boxes. For most use cases *IN-CS* does not provide enough detail to be useful in practice, but for some application scenarios it is the only type of *CS* for which automatic provenance computation can be provided. Under *copy-CS* (*C-CS*) a data item is considered to belong to the provenance of an output data item, if it has been copied literally from the input to the output (we do not specify if complete or just partial copying is required). Thus, the provenance of a data item d includes the data items that contributed values to d. *C-CS* class contribution semantics are generally used with fine-granular data items and allow to trace back the origin of values in the result of a transformation. This class of *CS* is not well suited for transformations that generate output data items without copying input values (e.g., aggregation of values). An extension to *C-CS* is *influence-CS* (*I-CS*). Contribution semantics belonging to this class include all data items in the provenance of an output data item d, that had some influence on the creation of d. For instance, data items that were used in a filter condition in the transformation that produced d. In

CS Category	Abbreviation	Description
Input-CS	IN-CS	The provenance of a data item d includes all input data items of the transformation(s) that produced d.
Copy-CS	C-CS	The provenance of a data item d includes data items which have been copied (partially or completely) to d.
Influence-CS	I-CS	The provenance of a data item d includes all data items that have influenced the creation of d in some way.

Figure 2.2: Data Provenance Contribution Semantics Categories

contrast to *IN-CS*, both *I-CS* and *C-CS* require knowledge about the inner working of the transformation for which provenance is computed. In spite of this disadvantage, these *CS* classes are in general preferred to *IN-CS*, because they provide more precise information about the relation between the input and output data items of a transformation. Figure 2.2 summarizes the *CS* classes discussed above.

Example 2.1. *As an example for the contribution semantics classes consider a transformation expressed as the following SQL query over the database presented below:*

```
SELECT name FROM person p, country c
WHERE p.countryId = c.id AND c.name = 'Germany';
```

person

name	*countryId*
Heinz Meier	*1*
Yu Chen	*2*

country

id	*name*
1	*Germany*
2	*Canada*
3	*China*

Query Result

name
Heinz Meier

If we assume the granularity to be tuples, then an IN-CS type would include all tuples from relations person and country into the provenance of the result tuple of the query presented above (the complete input of the query). An I-CS type would include only the tuples (Heinz Meier, 1) and (1,Germany), because only these tuples were used to produce the output tuple. A C-CS type would only include tuple (Heinz Meier, 1), because the value Heinz Meier in the result tuple was copied from this input tuple. None of the values from tuple (1,Germany) are copied to the result. Therefore, this tuple is not included in the provenance according to a C-CS type.

2.1.2 Transformation Provenance

In contrast to *data* provenance, *transformation* provenance is information about the transformations that generated a result data item d. For instance, the execution time of the transformation, run-time parameters, or the parts of the transformation that were active during the execution for hierarchically structured transformations. As mentioned above, the concept of granularity is applicable for transformation provenance too. For instance, the provenance of a workflow result could be modeled as properties of the workflow as a whole or as properties of each individual task of the workflow. Also the concept of contribution semantics translates to transformation provenance. E.g., all parts of a transformation that were executed could be considered to belong to the provenance, or just the parts that generated a non empty result. Most of the approaches that use *transformation* provenance are from the area of workflow systems and grid-computing.

Example 2.2. *Consider the following query over the database from the last example:*

```
SELECT id FROM country WHERE name = 'Germany'
UNION
SELECT id FROM country WHERE name = 'France';
```

We assume transformation provenance is defined as the parts of the transformation that produces result data. For the example query only SELECT id FROM country WHERE name = ’Germany’ *belongs to the provenance, because the other input of the set union does not produce any results.*

2.1.3 Provenance Computation

Provenance management systems should not rely on a user to manually provide provenance information, but they should be able to record or compute fine-grained provenance automatically. Surprisingly, quite a few approaches in provenance research have ignored this fundamental requirement. Provenance can be either recorded eagerly during the execution of a transformation (*eager* computation) or computed on-demand at the time it is requested by a user (*lazy* computation). *Eager* computation has access to all run-time properties of a transformation, and, thus, has the advantage that it can be applied to a wider range of transformation types. Computing provenance *eagerly* generates run-time overhead for the transformation and requires additional storage space to store the produced provenance information. In contrast, *lazy* computation mostly does not result in additional storage space and run-time overhead, but is not applicable for all types of transformations and can slow down provenance retrieval. In many settings *lazy* and *eager* aspects can be combined to benefit from the advantages of both methods, by computing provenance on-demand by re-executing the transformation and recording provenance like in the *eager* approach. This combination of *eager* and *lazy* computation is not applicable without storing additional information about a transformation. For example, access to the input data of the transformation is needed. If a transformation is non-deterministic, in the sense that it may produce different results for two executions over the same input data, this approach is not applicable[1].

Independent of the choice of *eager* or *lazy* computation there are two possible ways to generate provenance information. In the *propagation* approach information about source data items is propagated to the result of a transformation. Each data item in the result of a transformation will carry information about the source data items it is derived from (assuming that the propagation scheme reflects the semantics of the transformation). This is mostly achieved by instrumenting the original transformation to propagate provenance information. Most *propagation* approaches use *eager* computation. The *inverse* approach uses information about the structure of the transformation to compute an inverse of the transformation. For most practical applications the applied transformation functions have no inverse in the strict mathematical sense. Thus, additional information like, for example, the inputs of the transformation, is needed to be able to compute the inverse. Alternatively, provenance can be approximated by computing a pseudo inverse. For many interesting types of transformations the inverse approach cannot be applied because run-time information is needed to understand which output data item is derived from which input data item. Most *inverse* approaches use *lazy* computation. In *Perm*, we apply an lazy propagation approach, to avoid the cost of computing provenance if it is not needed.

2.1.4 Classification Scheme for Provenance Management Systems

To be able to discuss the differences and commonalities of provenance management systems, we present a classification scheme for comparing the critical properties of such systems and notations of provenance. Seven aspects of provenance that determine the functionality of a *PMS* are presented below:

- **Type of Provenance**: Does the *PMS* support *data* provenance, *transformation* provenance, or both provenance types?
- **Data and Transformation Model**: Which kinds of *data items* and *transformation* is the provenance definition applicable to?
- **Contribution Semantics**: Which type of contribution semantics is used?
- **Granularity**: Which granularity is supported by the provenance definition?
- **Representation**: How is provenance information represented?
- **Computation**: Does the system support automatic generation of provenance information and if so, how is provenance computed?

[1] In praxis many non-deterministic transformations can be transformed into deterministic ones by recording extra information. For instance, if a transformation is non-deterministic because it accesses the current system time, recording the value of this parameter at the time of the execution removes this source of non-determinism.

- **Querying**: Which types of queries over provenance information are supported? Is it possible to query both the data and the provenance using the same query language?

Answers to the questions stated above determine to a great extend the usefulness of a provenance management system. For example, a *PMS* is not very useful, if it does not support automatic generation of provenance and querying of this kind of information. In addition, these aspects enable an objective comparison of provenance management systems. Therefore, we use this classification scheme in the comparison of relational provenance management system presented in section 2.4.

2.2 An Overview of Provenance Research

In this section we provide an overview of research on data provenance. Several surveys on data provenance have been published in the last few years ([CCT09, FKSS08, GD07, IW09, SPG05a, SPG05b, Tan04, Tan07, DCBE$^+$07]). A theoretical treatment on the relationships between different *CS* types is given by Cheney et al. [CCT09]. The surveys presented in [DCBE$^+$07] and [FKSS08] both review provenance approaches from the workflow and grid-computing communities. Simmhan et al. [SPG05a, SPG05b] present a taxonomy of systems with provenance support. Glavic et al. [GD07] introduces a classification scheme of the functionality provided by *PMS* and analyzes several systems according to this scheme. Tan et al. [Tan07, Tan04] identify open problems in the area of provenance in databases.

2.2.1 Provenance in Databases

The topic of provenance for relational databases was first discussed in the context of visualization [WS97]. Transformations are represented as functions from one attribute domain to another. Provenance is traced by using inversions of these functions. Another approach for provenance management in a visualization environment is presented in [Gro04]. Groth et al. record user actions in an interactive visualization environment and present the whole user interaction as a DAG (directed acyclic graph). The user can navigate in this graph and jump back to previous states of the system.

Buneman et al. [BCCV06, BCC06] represent data from various data sources in a tree data model. In this framework, updates, insertions, and deletes are copy, paste, and insertion operations on these trees. The authors present a query language for this data model and define a *CS* type for copy, paste and insert operations. This data model was also applied for archiving [BKTT04]. This use case requires unique keys for every data object. These keys can be used to identify different versions of a data item in different versions of a database (or data repository).

Several systems integrate support for annotations into a relational database system. If a system provides support for propagating annotations during query execution, this can be used to compute provenance. For instance, the source data items may be annotated with unique identifiers. Hence, the annotations of a query result data item *d* represent the association between *d* and the source data items in its provenance. In *DBNotes* [BCTV04] each attribute value in a database can be annotated with a set of text annotations. The *Mondrian* [GKM05] system provides support for annotations that span multiple attributes of a tuple. *BDBMS* [EAE$^+$09, EOA$^+$08, EOA07] allows for annotations over rectangular regions spanning multiple attributes and tuples (assuming a fixed order on the tuples in the relation). *MMS* [SV07a, SV07b, SV07c] is an annotation system that models the associations between data and its annotations as queries. The result of such a query identifies the data items that carry a specific annotation. Annotation systems vary widely in their support for querying and propagating annotations, which determines if they are well-suited for managing provenance. We discuss these systems in detail in section 2.4.

Cui et al. present a system that *lazily* computes tuple granularity provenance of SQL queries using a type of *I-CS* [LZW$^+$97, CW00c, CW00b, CWW00, CW00a, Cui02]. Provenance is generated on demand by rewriting the original query into one or more reverse queries (*inverse* approach). *Trio* is a system that extends relational databases with support for provenance and uncertainty (called uncertainty-lineage-database or short *ULDB*) [Wid05, ABS$^+$06a, ABS$^+$06b, BSHW06, SBHW06, AW07, MTdK$^+$07, SUW07, WM07, WSU07, DSANW08, DSTW08, Wid08, WTS08, AW09]. Newer versions of *Trio* represent provenance as boolean formulas over base relation tuple identifiers. These formulas are used to compute the probability of a query result tuple based on the probabilities of the base relation tuples in its provenance. *Trio* applies a type of *I-CS*. Combining the ideas from *DBNotes* and *WHIPS* Zhang et al. [KP07, ZZZP07a] present a database system that uses query rewrite to propagate tuple granularity provenance with a type of *I-CS*. The original query result schema is extended with an additional attribute that stores provenance as a set of base relation tuple identifiers. Vansummeren et al. [VC07] present an approach to compute the provenance of a restricted set of SQL updates.

Provenance information has been used to translate deletions on a view into deletions on relations stored in the database. Similar results on this topic have been established by Cui et al. [CW01b] and by Buneman et al. [BKT02].

Early publications on data provenance (e.g., see [CWW00]) suggested that not only the provenance of existing results, but also for non-existing results could be of interest. Recent approaches address this problem. *Why-not* provenance [CJ09] helps a user in understanding why an input data item of a transformation did not contribute to the output of the transformation. In this work, this functionality was actually described as why a certain tuple is not in the output, but the system is only able to answer this kind of question if the query does not change attribute values. For a user provided pattern over the input data the *Why-not* provenance contains all parts of the transformation (called *picky*) where *i*, one of the input tuples that conform with the pattern, "got lost". "Got lost" is defined on top of *data* provenance. If the input of an operator in the transformation contains a tuple that is derived from *i* and the output does not contain such a tuple, then this operator is *picky*. To compute *Why-not* provenance the *data* provenance of several parts of the query have to be computed. Trio is used to generate the *data* provenance information.

Another approach that addresses the provenance of non-answers is [HCDN08]. In this approach potential modifications (inserts and updates) to a given database instance are studied that would result in a tuple *t* appearing in the result of a query *q*. The treatment is limited to select-project-join queries (*SPJ*) queries with conjunctive equality predicates. SQL is used to compute the potential modifications by rewriting the original query. In the result of the rewritten query modifications are represented as strings of the form *old value* → *new value*. This representation is not well-suited for being queried, because the modifications are encoded as strings. The approach presented in this work cannot automatically determine if the predicates used in a query are unsatisfiable. Some kind of constraint solving would be required to solve this problem (similar to the approach of *reverse query processing* presented in [Bin08]).

The Artemis system [HHT09] computes potential inserts to a given database instance that would cause a set of tuples that match a set of user specified patterns E to occur in a list of views. The system effectively computes the provenance of the tuple patterns in E. The provenance of these patterns is modeled as so-called *generic witnesses* that contain labeled nulls. Based on this set of generic witnesses *c-table* [2] representations of the relations in the database are generated that represent the original database instance and the witnesses computed from the user provided tuple patterns. The view definitions for the views for which patterns were specified are executed over these c-tables using the query execution semantics presented by Tomasz et al. [ILJ84] to generate the original views extended with tuples that match the patterns and potential side-effects of the modifications. In the resulting views original tuples will carry the condition *true* and pattern matches and side-effects are annotated with the conditions that have to be fulfilled for them to appear in the view. In the current version *Artemis* supports USPJ views (union-select-project-join). The system could be extended to support aggregation using the approach for aggregation over conditional tables presented by Lechtenbörger et al. [LSV02], but will inherit the problems of this approach (for example, large annotations on the result tuples of an aggregation).

Reverse Query Processing [Bin08, BKL06] bears some similarities with the provenance for non-answers approaches. Given a query q and a desired result relation R for this query, reverse query processing generates a database instance D on which the result of executing q would be R. The difference between this approach and the non-answers provenance approaches is that reverse query processing has a more complete handling of conditional expressions and supports a broader range of algebra operators, but is not able to modify an existing database instance to generate the desired query result.

Provenance and annotation management for relational databases has been studied in-depth from a theoretical point of view. Several types of *CS* have been proposed: *Why*-provenance [BKT01], *Where*-provenance [BKT01], and *Lineage* [CW00c]. *CS* are of considerable importance in provenance research, because they define which parts of a transformation input belongs to the provenance. Therefore, we discuss them in detail in section 2.3. In [GKT07a], Green et al. present a provenance representation (*How*-provenance) that annotates tuples with elements from a semi-ring. Containment of queries under annotation propagation has been studied Green [Gre09] and Tan [Tan03]. Geerts et al. [GVdB07] studied the relational completeness of the *color* algebra, a relational algebra with support for annotation propagation and querying. Several theoretical approaches define provenance as data-flow analysis ([CAA07, Che07]). Cheney et al. [CAA08] introduce provenance traces that generalize several *CS* types for the nested relational calculus (*NRC*) and present an operational semantics of *NRC* that in addition to the normal result builds a trace of the computation. Traces capture both *data* and *transformation* provenance, but are extremely

[2] *C-tables* are relations where each tuple is annotated with a condition. See [ILJ84].

verbose and, therefore, not applicable to real-world problems.

2.2.2 Non-Database Provenance Systems

2.2.2.1 Distributed Systems and Grid-Computing

Provenance plays an important role in *GryPhyN* [AZV+02, ZWF06, FVWZ02, ZWF+04, AF00, Fos03, CFV+08], a research project developing techniques for processing and managing large distributed data sets in data grids. A system called *Chimera* is introduced that manages the provenance of performed computations. Chimera offers a Virtual Data Catalog to store provenance information. A user registers transformations types, data objects and derivations (an execution of a transformation) in the Chimera Virtual Data Catalog (*VDC*). The VDC is implemented as a relational database. *VDL* (Virtual Data Language) provides query and data definition facilities for the Chimera system. While the first prototype is limited to file system data objects and executable program transformations, the system is to be extended to support relational or object-oriented databases and SQL-like transformations.

Groth et al. [CTX+05, GLM04a, GJM+06b, GMTM05, GJM+06a, GLM04b, MCG+05, GMM05, KVVS+06, GGS+03, WMF+05a, SM03, GMF+05, TGM+06, MI06, KTL+03, MGM+08, MGBM07, WMF+05b] have developed a protocol (PReP: P-assertion Recording Protocol) for recoding the provenance of interactions between services in a service-oriented architecture (SOA). This protocol requires the communicating services to send so called p-assertions, which are recorded by a provenance store web service. P-assertions represent either inputs and outputs sent by the services using *IN-CS* or metadata provided by the services about their internal state. Depending on the internal state data provided by the services it could be possible to record fine-grained provenance with this approach. The p-assertions transmitted by the services are stored by a provenance store service. To be able to correlate the multiple p-assertions sent by the communicating services, an identifier is attached to each p-assertion. PReServ [GMTM05] is a web service implementation of the PReP protocol. The provenance store service provided by this implementation uses a modular architecture. Thus, enabling multiple storage models for provenance data storage. Provenance information can be queried through the query API included in PReServ.

ESSW [FB01, Bos02, Bos04], the Earth System Science Workbench, is a scientific computing environment with a central server that records metadata and provenance about computations executed by scientists working at client workstations. The metadata collection is realized by client-side perl scripts that act as wrappers for the transformations applied by the users and send the collected meta-data as XML documents to the central server. At the server side these XML documents are stored in a relational database.

2.2.2.2 Workflow Management Systems

Several workflow management systems have been extended with provenance recoding functionality. The *Taverna* workflow-management-system [ZGG+03b, ZGG+03a, MPL+06, SRG03, GGS+03, ZGSB04, ZWG+04] was developed by the *myGrid* project. The goal of *myGrid* is to apply semantic web technologies to simplify data analysis for life science research. *Taverna* represents the provenance of workflows as *RDF* data. In *Karma* [SPG08a, CPS+09, SPG08b] workflows are composed of services. The system provides an interface through which each service participating in a workflow can publish its provenance. Thus, provenance generation is offloaded to the actors in the workflow. *Karma* collects the published provenance information and stores it in a relational database. *Pegasus* [KDG+08] and *Swift* [ZHC+07, ZDF+05] are workflow management systems based on the ideas developed in *Chimera* (see discussion above). *REDUX* is an extension of the *Windows Workflow Foundation* that captures the provenance of workflow executions. Provenance is stored in a relational database and queried using SQL. *REDUX* supports the generation of executable workflow definitions from the result of a provenance query.

Kepler [ABJF06, BML+06, ABML09a, ABML09b, LPA+08, BML08, MBZL08, LAB+06] is a workflow management system that supports multiple execution paradigms and data models. A generic provenance support was introduced that can handle provenance for different *Directors* implemented in the system. With the *COMAD* (Collection-oriented modeling and design) paradigm [LAB+06] developed in Kepler each actor in a workflow updates an XML data stream. Fine-grained provenance support and an XPath based query language for *COMAD* were introduced by Anand et al. [ABML09a, ABML09b]. The major advantage of this approach over other workflow based approaches is the support for recording fine-grained *data* provenance.

VisTrails [HLB+08, SFC07, DCBE+07, SVK+08, SKS+07, EKA+08, CFS+06] is a workflow management and visualization system which is designed to support rapidly evolving workflows. The system traces the history and provenance of a workflow definition and its executions while the definition is modified by a user. Provenance is stored either as XML or in a relational database. Independent of the data model used for storage, provenance is queried using the *vtPQL* query language developed for *VisTrails* that supports keyword-based and navigational search (e.g., like ”is a previous version of”) with selection predicates.

2.2.2.3 Storage Schemes for Provenance

The size of provenance data can easily exceed the combined size of base data and result data. Several approaches tried to address this problem by introducing compression techniques for provenance. Compression usually increases the cost of queries over the compressed provenance which lead to the development of storage schemes that aim at reducing the cost of retrieval instead of minimizing the storage overhead of provenance.

Chapman et al. [CJR08] present an compression algorithm for a tree model of data and its provenance. Common subtrees are factored out, thus, stored only once, and references are used to refer to the single instance of such subtrees. If data items inherit the provenance of the parent data items that include them, then only the association between provenance and the parent data item is stored. The default behavior in this case is to propagate provenance from a parent data item d to all children of d that do not have provenance information attached themselves. Furthermore, if many data items have the same provenance p, the association between data and provenance information can be described by predicates that evaluate to true if applied to the data items which are associated to p.

Heinis et al. [HA08] developed a representation for the provenance of workflow executions presented as graphs that increases the performance of provenance retrieval. This is achieved by a graph encoding that enables the efficient retrieval of paths in this graph. The provenance of workflow executions is modeled as a dependency graph that stores the tasks executed by the workflow and the data items used as inputs or generated as output by these tasks. The authors argue that most systems store such graphs as parent-child relationships which implies the use of recursive queries to retrieve the provenance of an output data item because the paths between inputs and outputs of the complete workflow have to be reconstructed from the parent-child relationship representation. This problem is addressed by using interval tree encoding [Tro05] to represent provenance graphs. While compression and storage layouts for efficient retrieval are important issues in data provenance research, it remains to be seen how the proposed schemes perform if included into a fully-fledged *PMS*.

Anand et al. [ABML09a] present an compression scheme for provenance of workflow traces generated as XML by the *Kepler* workflow management system discussed above. XML traces are stored in a relational database. Various compression strategies are discussed and their influence on query and update performance is analyzed. For instance, common subsequences can be factored out similar to the approach of Chapman et al. [CJR08]. Transitive provenance associations (e.g., A was created from B which in turn was generated from C, thus C belongs to the provenance of A) can be either stored explicitly or generated at query time.

2.2.2.4 GIS

In the GIS research area, the importance of provenance has been recognized early on. Most publications from this area focused on the development of metadata standards (e.g., [HE97, HQGW93, Lan89, Lan93, SCL01, SCN+93, SCL99]) which model provenance information. A well-designed metadata standard could provide a basis for a provenance management system, but the proposed standards are limited to the GIS-domain and cannot be easily generalized. More important the metadata defined by these standards is meant to be provided by a user and may not be appropriate for automatic generation of provenance.

2.2.2.5 Provenance for Schema Mappings

Schema mappings, logical specifications of the relationships between schemas, are used in data integration and data exchange to rewrite queries over a global schema into queries over local schemas or to translate data from a source to a target schema. Schema mappings typically do not map all elements of the source and target schema. In data exchange executable transformations (*implementing transformations*) are generated from schema mappings that generate an instance of a target schema from a source schema instance and produce new values for target schema elements that are not specified by the mappings. Mapping data from one schema to another is a complex semi-automatic process involving multiple correlated steps: Data cleaning, identification of correspondences between source and target schema elements, generating schema mappings, and generating implementing transformations from the mappings. Hence, debugging such a process tends to be quite complex. Recent approaches to schema mapping debugging facilitate provenance information to simplify the debugging process.

SPIDER [CT06] defines provenance as so-called *routes* computed for a subset of a target instance. Each route is a possible way of producing the tuples of interest by sequentially applying schema mappings to tuples (*route-steps*) in the source instance (and the tuples generated by previous mapping applications in the route). A route combines *data* provenance with *transformation* provenance for mapping transformations. Because routes only consider the logical specification of a mapping, they have the advantage of being independent of the concrete implementation of a mapping in the form of a transformation query or program. On the other hand this independence can also be problematic if an error is caused by an incorrect transformation. Furthermore, no query facilities for routes are provided, they can only be explored using the visual interface of *SPIDER*.

ORCHESTRA [GKIT07] is a collaborative data sharing system (CDSS) that uses schema mappings to exchange updates in a peer-to-peer network. *How*-provenance is used to describe the origin of mapped updates. The mappings that were used to translate an update are represented as functions in the semi-ring provenance model. We discuss this model in detail in section 2.3.

MXQL [VMM05] generates provenance information for data exchange settings. Provenance information is generated during the execution of a transformation that implements a mapping by generating a target instance from a source instance. The generated target instance is enriched with relations that store *mapping* provenance information and provenance that relates source to target schema elements. *MXQL* provides full query language support for provenance and mapping information (SQL).

2.2.2.6 Other Approaches

The Open Provenance model [MFM$^+$07], a graph based representation of *data* and *transformation* provenance, was proposed as an attempt to standardize the representation of provenance information. The goal of this work is to provide a model of provenance that can be used to exchange information between *PMS*, allow the representation of different types of provenance in one model, to describe a basic set of structural constraints that define sound provenance graphs, and introduce inference rules on these graphs (e.g., transitivity).

Marathe et al. [MS97, Mar01] study the provenance of the array-manipulation language (*AML*), an algebra for querying multi-dimensional arrays. A user can select a part of the output (using an projection-like operator of the algebra) of an *AML* expression for which (s)he would like to compute provenance. Provenance computation is realized by rewriting the algebra expression to generate the provenance.

Seltzer et al. [BGH$^+$06, LNH$^+$05, SMRH$^+$05] developed *PASS* (*Provenance-Aware Storage System*), a prototype of a storage system that integrates provenance information. An extension of a Linux kernel and file system is used to manage provenance data. The provenance of a file is stored in a new type of file node. Unlike normal files, provenance information is never deleted and exists as long as the file system itself. The Lineage Filesystem [SC05] is another approach that extends a file-system with provenance functionality. The linux kernel is extended to log process and file-related system calls. These logs are read by a daemon that periodically processes this log to generate provenance information and store this information in a *MySQL* database.

ES3 (*Earth System Science Server*) [FMS08, FS08a, FS08b, FSP07] tracks the execution of processes and their system calls, enhances this information with unique identifiers, and reconstructs provenance from these traces that are stored as XML data. The traces are generated by a logger that can be extended with plugins to allow for different methods of extracting the trace during program execution. Two plugins where presented in [FMS08]. One that intercepts and logs system calls using the *strace* facilities of Unix operating systems. The other one is a plugin for the *IDL* analysis environment [htt09b] that instruments *IDL* scripts to realize the logging.

Following the vision of using program analysis methods to generate provenance outlined Cheney et al. [Che00, Che07, CAA07], *Valgrind* [Net09] is used by [ZZZP07b, KP07] to generate provenance information for arbitrary executable programs. The executable of the program is instrumented to propagate provenance information alongside with the original computation based on the data-dependencies of the program. This is similar to the forward computation of a dynamic slice of a program (see e.g., [KR98] and [KY94]). Control-dependencies are ignored by this approach because they can degrade the quality of provenance information by inclusion of false positives in the provenance. It would be interesting to see wether this approach can be extended to include relevant control-dependencies into the provenance computation.

In many application domains, especially in distributed systems, provenance computation requires the unique identification of data items used and generated by a transformation. [CJ08] presents and evaluates several strategies for uniquely identifying data items.

2.3 Comparison of Contribution Semantics

We now discuss the *CS* definitions introduced in the literature. The discussion is limited to approaches that provide formal specifications of *CS* for the relational data model. Here we are only interested in the properties of *CS* types and not their formal specification. The interested reader is refereed to [CCT09] for formal *CS* definitions and proofs for some of the properties of *CS* types we state in this section.

2.3.1 Why-CS

Granularity	tuple
Classification	I-CS
Representation	Set of Set of Tuples
Invariant Under Query rewrite	Yes / No
Transformation Scope	USPJ

Figure 2.3: Why-CS

Buneman et al. [BKT01] introduced two types of contribution semantics called *Why-provenance* and *Where-provenance*. In this work a hierarchical data model was used, but these *CS* types were later adapted for the relational model in [BKT02] and for the nested relational calculus (NRC) in [BCV08]. We base our discussion on the relational version presented in [BKT02] and the extended discussion of these *CS* types presented in [CCT09]. *Why-provenance*, which we refer to as *Why-CS*, is a tuple granularity *I-CS* type. Two types of *Why-CS* were defined by the authors; one that is sensitive to query rewrite (*Why-CS*) and one that is invariant under query rewrite (*IWhy-CS*). We call a contribution semantics invariant under query rewrite, iff it produces the same provenance for equivalent queries (two queries are called equivalent, if they produce the same result on every possible database instance). Under *Why-CS* the provenance of a result tuple t of a query q executed over a database instance I is represented as a set $Why(Q,I,t)$ of *witnesses* (usually we omit the instance I if it is clear from the context). A *witness* w is a set of tuples from the instance I with $t \in Q(w)$. That means, the result of executing query q on witness w contains t. A witness for a tuple t may contain tuples that do not contribute to t. For example, the complete instance I is a trivial witness for every query and result tuple. The set $Why(Q,I,t)$, called the *witness-basis*, contains only a subset of all witnesses of t. The *witness-basis* is defined as a set of construction rules that operate on the algebra representation of a query. We do not present these rules here, the interested reader is referred to [CCT09]. Figure 2.5 shows an example database instance, queries defined over the schema of the example database, the results of executing these queries, and the provenance of result tuples according to different *CS* types. The database models shops, items and a stock relation that represents the relationship between shops and items in their stock. For convenience all base relations and result tuples are labeled with identifiers (e.g., i_2). As an example of *Why-CS* consider the provenance of result tuple t_1 from query q_1, a simple selection on the *price* attribute of the *item* relation with a projection on the item name. The set $Why(q_1,t_1)$ contains a single witness $\{i_1\}$ that contains the tuple i_1 from which t_1 is derived. Recall that *Why-CS* is sensitive to query rewrite. For instance, queries q_2 and q_3 from Figure 2.5 are equivalent, but, e.g., $Why(q_2,t_4) \neq Why(q_3,t_4)$. Intuitively, *Why-CS* provenance contains all the tuples from instance I that had some influence on the creation of t. E.g., $Why(q_4,t_2)$ includes the three tuples that were joined by q_4 to produce result tuple t_2 in spite of the fact that only data from tuple s_2 is copied to t_2. The version of *Why-CS* that is invariant under query rewrite (*IWhy-CS*) is defined as the set $IWhy(Q,t)$ of minimal witnesses contained in $Why(Q,t)$. A witness w is minimal, iff no subset of w is also an element of $Why(Q,I,t)$. In [CCT09] this set is called the *minimal witness basis*. The invariance under query rewrite of *IWhy-CS* is proven by showing the equivalence of *IWhy-CS* with an alternative definition of this provenance type that is invariant under query rewrite by definition (the condition that decides if a witness belongs to the provenance does not reference the syntactical structure of a query). As an example of *IWhy-CS* consider queries q_2 and q_3 from the example. These queries are equivalent and, therefore, have the same *IWhy-CS* provenance: The single minimal witness i_4. Both variants of *Why-CS* are defined for unions of conjunctive queries (or union of select-project-join queries with only equality selection and join predicates: *E-USPJ*), but can be extended for aggregation and other types of set operations. Figure 2.3 summarizes the properties of *Why-CS*.

2.3.2 Where-CS

Granularity	attribute value
Classification	C-CS
Representation	Sets of attribute value identifiers
Invariant Under Query rewrite	Yes / No
Query Language Scope	E-USPJ

Figure 2.4: Where-CS

The second type of contribution semantics introduced in [BKT01] is *Where-provenance* (*Where-CS*). *Where-CS* is a type of *C-CS* with attribute-value granularity. Under this *CS* type provenance describes from which input attribute values a given output attribute value is derived from. Like *Why-CS*, two versions of this *CS* type are defined. One that is sensitive to query rewrite, because it is defined over the syntactical structure of a query, and, one that is invariant under query rewrite. We refer to these types as *Where-CS* and *IWhere-CS*. *Where-CS* is defined as a set of annotation propagation rules that propagate annotations from the input of a query to its output. One rule is defined for each supported algebra operator that defines the propagation behavior of this operator. These propagation rules can be used to compute the provenance of a query, if every attribute value in the database instance is annotated with a unique identifier. Under this assumption the annotations that are generated for an attribute value in the result of a query represent the provenance of this value. In contrast to *Why-CS* under *Where-CS* only values that are copied from the input to the output of a query belong to the provenance of an output. For this type of provenance selection conditions that enforce the equality of two attributes are also considered as a form of copying. As an example for *Where-CS*, reconsider query q_4 from the example shown in Figure 2.5. The provenance of the attribute value t_{11} contains only s_{12}, the name of the shop that is copied literally from the input of the query (here t_{1x} denotes the attribute value at the x-th attribute of tuple t_1). Note the difference to *Why-CS* which includes tuples from all relations joined in this query, because all of these tuples contributed to the result by their participation in the join operation. As an example for the sensitivity of *Where-CS* to query rewrite consider queries q_5 and q_6 from the example. These queries are equivalent, but their provenance is different. The rewrite invariant version of this contribution semantics (*IWhere-CS*) is defined over the rewrite sensitive version. The annotations placed on an output attribute value of a query q are defined as the union of all annotations that are generated by computing the *Where-CS* annotations for every query q' that is equivalent to q. Even though there are infinitely many queries q' that are equivalent to q, it was proven in [BCTV05] that the number of equivalent queries with different annotation propagation behavior is finite. This is due to the fact that there are only finitely many annotations in the database instance. Therefore, the number of different result annotations based on this set of annotations has to be finite too. Note that while for *Why-CS* the rewrite invariant version of this *CS* type is contained in the rewrite sensitive version, for *Where-CS* the exact opposite holds. For instance, the *IWhere-CS* provenance of queries q_5 and q_6 for result attribute value t_{11} is the set $\{o_{11}, o_{21}, o_{31}\}$ which contains both the *Where-CS* provenance of query q_5 and query q_6. Like *Why-CS*, *Where-CS* is defined for *E-USPJ*-queries. An extension for aggregation seems possible (at least for the version that is sensitive to query rewrite), but is not very useful because aggregated values are computed by the transformation and not copied from the input. Figure 2.4 summarizes the properties of *Where-CS*.

2.3.3 Lineage-CS

Granularity	tuple
Classification	I-CS
Representation	List of Sets of Tuples
Invariant Under Query rewrite	No
Query Language Scope	Set-ASPJ

Figure 2.6: Lineage-CS

The provenance of views in a datawarehouse was studied in the scope of the *WHIPS* datawarehouse project developed at Stanford University [CW00a]. The first formal specification of contribution semantics was developed in the *WHIPS* project. The *CS* type presented in this work is a type of *I-CS* with tuple level granularity. Adapting the terminology of recent data provenance publications we call this type of contribution semantics *Lineage-CS*. Under *Lineage-CS* the provenance of a result tuple t of an algebra expression q is modeled as a list $Lin(Q, I, t)$ of subsets of the input relations of q that fulfills three conditions. First, the result of evaluating q over the provenance of t should produce exactly t and nothing else. This condition is similar to the definition of a witness in *Why-CS*. Second, every tuple in the provenance should contribute to t. Third, the subsets of the input relations in the provenance are the

item

	itemId	name	price
i_1	1	Lawnmower	130
i_2	2	Fertilizer	15
i_3	3	Shovel	20
i_4	4	Pickaxe	30

stock

	shopId	ItemId
o_1	1	1
o_2	1	2
o_3	1	4
o_4	2	1
o_5	2	3

shop

	shopId	name	location
s_1	1	Do-It-Yourself	New York
s_2	2	Garden-shop	Boston

```
q1 = SELECT i.name FROM item i WHERE i.price > 100 ;
q2 = SELECT DISTINCT i.* FROM item i JOIN item j ON (i.price <= j.price);
q3 = SELECT * FROM item i;
q4 = SELECT DISTINCT s.name
     FROM item i, shop s, stock t
     WHERE i.itemId = t.itemId AND t.shopId = s.shopId AND i.price < 100;
q5 = SELECT * FROM stock;
q6 = SELECT DISTINCT a.* FROM stock a, stock b WHERE a.shopId = b.shopId;
```

Result q₁

	name
t_1	Lawnmower

Result q₅ and q₆

	name
t_1	Do-It-Yourself
t_2	Garden-shop

Result q₂ and q₃

	itemId	name	price
t_1	1	Lawnmower	130
t_2	2	Fertilizer	15
t_3	3	Shovel	20
t_4	4	Pickaxe	30

Result q₄

	name
t_1	Do-It-Yourself
t_2	Garden-shop

Lineage-CS

$$Lin(q_1,t_1) = <\{i_1\}>$$
$$Lin(q_2,t_4) = <\{i_4\},\{i_1,i_4\}>$$
$$Lin(q_3,t_4) = <\{i_4\}>$$
$$Lin(q_4,t_1) = <\{i_2,i_4\},\{s_1\},\{o_2,o_3\}>$$
$$Lin(q_4,t_2) = <\{i_3\},\{s_2\},\{o_5\}>$$

Why-CS

$$Why(q_1,t_1) = IWhy(q_1,t_1) = \{\{i_1\}\}$$
$$Why(q_2,t_4) = IWhy(q_2,t_4) = \{\{i_4\}\}$$
$$Why(q_3,t_4) = \{\{i_4\},\{i_1,i_4\}\}$$
$$IWhy(q_3,t_4) = \{\{i_4\}\}$$
$$Why(q_4,t_1) = IWhy(q_4,t_1) = \{\{i_2,o_2,s_1\},\{i_4,o_3,s_1\}\}$$
$$Why(q_4,t_2) = IWhy(q_4,t_2) = \{\{i_3,s_2,o_5\}\}$$

Where-CS

$$Where(q_1,t_{12}) = IWhere(q_1,t_{12}) = \{\{i_{12}\}\}$$
$$Where(q_4,t_{11}) = \{s_{12}\}$$
$$Where(q_5,t_{11}) = \{o_{11}\}$$
$$Where(q_5,t_{12}) = \{o_{12}\}$$
$$Where(q_6,t_{11}) = \{o_{11},o_{21},o_{31}\}$$
$$Where(q_6,t_{12}) = \{o_{12}\}$$
$$IWhere(q_5,t_{11}) = IWhere(q_6,t_{11}) = \{o_{11},o_{21},o_{31}\}$$
$$IWhere(q_5,t_{11}) = IWhere(q_6,t_{11}) = \{o_{12},o_{41}\}$$

How-CS

$$How(q_1,t_1) = i_1$$
$$How(q_2,t_4) = {i_4}^2 + i_1 \times i_4$$
$$How(q_3,t_4) = i_4$$
$$How(q_4,t_1) = (i_2 \times s_1 \times o_2) + (i_4 \times s_1 \times o_3)$$
$$How(q_4,t_2) = i_3 \times s_2 \times o_5$$

Figure 2.5: Contribution Semantics Examples

maximal sets that fulfill the first two conditions. We discuss the formal definition in detail in section 3.2.1. *Lineage-CS* is similar to *Why-CS*, because like this *CS* type it also includes tuples in the provenance that only have a ”conditional” influence on the output by, e.g., being used in a join condition. The representation of *Lineage-CS* is tightly bound to the syntactical structure of a query, because the provenance is modeled as a list of subsets of the input relations of a query. Hence, this type of *CS* is sensitive to query rewrite. For example, exchanging the left and right input of a join modifies its provenance by changing the order in which the subsets of the input relations appear in the provenance. More examples for *Lineage-CS* are shown in Figure 2.5. E.g., note the difference in representation between $Lin(q_4, t_1)$ and $Why(q_4, t_1)$. *Lineage-CS* is defined for *SPJ*-queries, aggregation, and set operations (*Set-ASPJ*). Figure 2.6 reviews the properties of this *CS* type.

2.3.4 How-CS

Granularity	tuple
Classification	I-CS
Representation	Polynoms over tuple identifier variables
Invariant Under Query rewrite	No
Query Language Scope	USPJ + datalog recursion

Figure 2.7: How-CS

In [GKT07a] Green et al. introduced the concept of *How*-provenance. Relational tuples are annotated with elements from a semi-ring K. The propagation of these annotations throughout a query is defined by mapping the algebra operators to the addition and multiplication operations of the semi-ring. E.g., join is mapped to multiplication. The semi-ring model was originally defined for *USPJ* queries with recursion and later extended for a subset of *XQuery* over unordered XML [FGT08]. The semi-ring model generalizes several important extensions of the relational model (e.g., bag-semantics and probabilistic databases) and *data* provenance contribution semantics (*Where-CS*, *Lineage-CS*, and *Why-CS*). For example, if the semi-ring of natural numbers is used to annotate tuples, then these annotations model bag-semantics. E.g., joining a tuple s with multiplicity 3 with a tuple t with multiplicity 5 results in a tuple with multiplicity 15 ($3 \times 5 = 15$). Note that it was later shown in [CCT09] that for some cases *Where-CS* provenance cannot be modeled in the semi-ring-model. The authors of the original paper [GKT07a] also introduced a new type of *CS* using the semi-ring model (*How-CS*). Tuples are annotated with polynomials over a set of variables where each variable represents one tuple from the database instance. The addition and multiplication operations in such an polynomial represent alternative and conjunctive use of tuples in the transformation that produced t. For instance, if query $q = R \bowtie_{a=b} S$ joins tuples r_1 and s_1 to produce output tuple t, then the provenance of t would be represented as $r_1 \times s_1$. Because of the indication of alternative and conjunctive use of tuples by *How*-provenance, this type of provenance can be considered to carry *transformation* provenance information in addition to *data* provenance. Some examples of *How-CS* are given in Figure 2.5. For instance, $How(q_4, t_4)$ is $i_4{}^2 + i_1 \times i_4$ indicating that the result tuple t_4 of query q_4 has two alternative derviations. One that joins tuple i_4 with itself ($i_4{}^2$) and one that joins tuple i_1 with tuple i_4. *How-CS* is sensitive to query rewrite (e.g., $How(q_2, t_4) \neq How(q_3, t_4)$). Figure 2.7 summarizes the properties of this *CS* type.

Note that using an *CS* type that is invariant under query rewrite may increase the performance of provenance computation, because it enables the application of relational equivalence rules in provenance computation, i.e., standard approaches to query optimization can be applied. On the other hand, the computations of the query rewrite invariant *CS* types are more complex, negating this potential advantage. For instance, *IWhere-CS* is computed by executing several *Where-CS* computations. While extending the relational model in a way that invalidates query equivalence is considered as ”bad practice”, it can be advantageous to define contribution semantics that are sensitive to query rewrite. Users usually have an intention in writing their queries in a certain way. A *CS* type that is sensitive to query rewrites honors the user intentions and, thus, may produce provenance information that better fits user expectations.

2.4 Comparison of Provenance Management Systems

We now discuss several provenance management systems in detail. We only discuss database systems with provenance support and leave out approaches from the workflow and grid-computing communities because they are less related to our approach. Furthermore, we leave out systems that are not reasonably mature or are just mere "proofs of concept " implementations.

2.4.1 WHIPS

In context of the WHIPS datawarehouse prototype the provenance of views is computed *lazily* according to *Lineage-CS* [LZW$^+$97, CW00c, CW00b, CWW00, CW00a, Cui02]. Cui et al. presented algorithms that can be used to trace back the provenance of arbitrary *Set-ASPJ* queries. The algorithms apply the *inverse* approach by generating a tracing query that computes the provenance of each relational operator in a query q, and, thus recursively tracing back the provenance of a result tuple t or set of result tuples one operator at a time. The authors proved that if a query q is transformed into a canonical form, then the trace back queries can be generated for segments of several operators in this query, but in general it is not possible to generate a single trace back query for the complete query q. As mentioned before *Lineage-CS* represents provenance as a list of relations (subsets of the input relations). User defined functions (UDFs) are employed to split the result of a tracing query and produce this representation, because the relational model does not support queries with more than one output relation. The major drawback of this approach is that each provenance computation requires the execution of several queries and UDFs and, thus, does not exploit the whole optimization potential of the underlying DBMS. Another disadvantage of this approach is that the association between the result tuples and their provenance is not preserved in the representation applied in *WHIPS*. The provenance of a set of result tuples is represented in the same way as the provenance of a single result tuple. In this representation it is not clear to which result tuple a tuple in the provenance belongs to. In [CW01a] the WHIPS system was extended to support provenance for non-relational transformation which are applied in ETL-processes that import data into a datawarehouse system. *ETL* processes are classified according to their provenance behavior (e.g., a process that produces one output data item from several input data items is called an *aggregator*). The authors present rules to deduce the behavior of a complex transformation from the behavior of the transformations it is composed of. The classification of a transformation is assumed to be provided by the creator of the transformation.

2.4.2 Trio

Trio is a system that extends relational databases with support for provenance and uncertainty (called uncertainty-lineage-database) [Wid05, ABS$^+$06a, ABS$^+$06b, BSHW06, SBHW06, AW07, MTdK$^+$07] [SUW07, WM07, WSU07, DSANW08, DSTW08, Wid08, WTS08, AW09]. In the first versions of *Trio* provenance was represented as mappings between input tuple identifiers and output tuple identifiers. These mappings were stored in so-called *lineage* tables. Lineage tables are generated *eagerly* during query execution. Recent publications on *Trio* (e.g., [WTS08]) represent provenance as boolean formulas over the probabilities of base relation tuples similar to *How*-provenance. The system implements a query language called *TriQL* that is based on SQL and enhanced with features to handle provenance and uncertainty. The predicate *Lineage(R,S)* can be used to filter out tuples from the result of a query, if the tuple r from relation R is not in the provenance of tuple s from relation S (both relations have to be accessed in the *FROM* clause of the query). *Trio* is implemented on top of *PostgreSQL* as a middleware written in Python with some part of the functionality outsourced into UDF's. The current version of *Trio* supports provenance computation for a restricted version of *Set-ASPJ* (Set operations and *ASPJ* queries). I.e., no sub-queries in the *FROM* clause, only a single set operation, and aggregation cannot be combined with set operations or joins. Because *Trio* is implemented as a middleware solution with a standard relational back-end, relations with uncertainty have to be simulated as normal relations. This and the separation of lineage tables result in a *TriQL* query being translated into several SQL queries and UDF calls, which has a negative effect on the performance of the system. Furthermore, the results of these queries have to be post-processed to generate a result representation that conforms to the *Trio* data model, causing additional run-time overhead. Provenance results in storage overhead in *Trio*, because the lineage relations are stored permanently. Despite of its shortcomings *Trio* is the first system to combine uncertainty and provenance.

2.4.3 DBNotes

Tan et. al. presented *DBNotes*, an database system with support for annotations on attribute values ([BKT01, Tan03, BCTV04, CTV05, Ale05]) . *DBNotes* enables a user to annotate the attribute values in the relations stored in the database. Provenance is modeled by annotating each attribute value with an unique identifier. *psql*, the query language of *DBNotes*, extends *USPJ* queries with selection predicates restricted to conjunctive equality conditions (*E-USPJ*) with language constructs that control propagation of annotations. The two main propagation schemes *default* and *default-all* apply *Where-CS* and *IWhere-CS*. The third propagation scheme enables a user to manually define from which source attributes the annotations of a query result attribute are derived. This propagation scheme makes *DBNotes* the first *PMS* with user defined contribution semantics, though only contribution semantics that are definable on schema level are supported. For example, it is not possible to change the propagation behaviour of annotations based on attribute values. *DBNotes* is implemented as a middleware on top of a standard relational database system. Annotated relations are modeled as standard relations by adding an annotation attribute A_a for each attribute A of the original relation. Annotations are stored as plain text in these annotation attributes. If an attribute value is associated with more than one annotation, the original tuple to which this attribute value belongs has to be duplicated. Therefore, this representation can result in significant storage overhead. *psql* queries are rewritten into SQL queries over this storage scheme. Depending on the propagation scheme that is applied in the *psql* query the result is a wrapper query that unions several modified versions of the original query and sorts the result on the original result attributes. The result of the wrapper query contains duplicates of the original result tuples each associated with a part of the annotations for this tuple. A post-processing step that runs in linear time gathers all annotations of a result tuple and returns the annotated tuple. Like for *Trio*, the performance of the system suffers from the overhead introduced by post-processing, but in contrast to this system *DBNotes* rewrites a *psql* query into a single, though arguably more complex, query. The *psql* language has no constructs to access annotations directly, thus it is not possible to issue queries over generated provenance information.

2.4.4 Mondrian

Mondrian [GKM06, GKM05], like *DBNotes* is an annotation management system for relational databases. *Mondrian* supports annotations on sets of attributes values, where all attributes values from a set belong to the same tuple, subsuming the functionality of *DBNotes* that allows only single attribute value annotations. In the terminology of [GKM05] annotation values are called colors and a set of attributes that carries a color is called a colored block. Queries are expressed in the *color-algebra* that is basically a *E-USPJ* (*USPJ* queries and only equality comparisons are allowed in selection and join predicates) fragment of relational algebra extended with operators that modify blocks. *Mondrian* is implemented as a middleware solution on top of *MySQL*. Colored relations are represented by extending the attributes of the original relation with an attribute that stores colors and an additional attribute per original attribute that indicates whether the attribute belongs to a block. In this representation a tuple represents one original tuple and one of its blocks. Thus, the original tuple will be duplicated according to the number of blocks for this tuple. Queries in the *color-algebra* are translated into SQL queries over this representation. Hence, in terms of performance and storage overhead *Mondrian* suffers from the same problems as *DBNotes*. The color-algebra is well-suited for annotations created by users, but some of the propagation rules are not well-suited for provenance annotations. For instance, the join operator of the color-algebra only propagates annotations to a result tuple that are present in both input tuples. i.e., for provenance annotations this means the provenance is empty for all tuples except the ones that are produced by joining a tuple with itself.

2.4.5 MMS

In [SV07a, SV07c] the *MMS* system is introduced that models annotations as normal data. The association between annotations and the annotated data is described by SQL queries that are stored as attribute values of a new data type *query*. The result of such a query identifies the attribute values that carry a specific annotation. For instance, a query SELECT name FROM employee WHERE age=30; would annotate the name attribute values of all employee tuples with an age equal to 30. To enable querying of the associations between data and annotations a new predicate $Q[A_1,\ldots,A_n] \doteq [v_1,\ldots,v_n]$ is added to SQL that evaluates to true if the result of the query q stored in attribute Q contains a tuple with values $v_1,\ldots,v_n$ in the attributes $A = A_1,\ldots,A_n$. If q does not contain all attributes in A the predicate evaluates to false. The predicate is implemented by query rewrites. First, a query is issued to get all the values of the query data-type attribute Q and check for each query value if its evaluation contains the attributes in A. To be able to perform this check the result attributes of the evaluation of each query value are stored in additional relations. The original query is then rewritten into a union of individual query parts that each represent the execution of the original query and one of the query values from the result of the first step. Thus, the size of the rewritten query is linear in the number of query values that are extracted in the first step. A problem with this approach is that if a relation is annotated with many different annotations, like it would be the case for fine-grained provenance information, the dynamic evaluation of queries will result in a huge performance overhead. In addition, since in this case most queries will annotate only a single tuple, the main advantage of this approach is lost. In contrast to *DBNotes* and *Mondrian*, *MMS* has no support for automatic propagation of annotations throughout a query which would be required to compute the provenance of a query over the base data automatically.

2.4.6 BDBMS

BDMBS [EAE$^+$09, EOA07, EOA$^+$08] is an extension of *PostgreSQL* with support for annotations. Annotations can be placed on rectangular regions in a table or on combinations of tuples of a join result. The system uses so-called annotation tables to store all annotations placed on a data relation. Multiple annotation tables for a single data relation enable to distinguish between annotations of different types. For instance, separate annotation relations could be used for annotations from different users. SQL is extended with new constructs for adding and querying annotations. In the *ADD ANNOTATION* command the user specifies a query to select the rectangular region that should be annotated. During query processing an annotation a from an annotation table is propagated to a result tuple t if the annotation table is mentioned after an annotated relation R in the *FROM*-clause, the result attributes intersect with the rectangle defined by the annotation, and the input tuple of relation R from which t is produced is covered by the rectangle. The annotation propagation is implemented as query rewrites, by left-joining the annotation table to the data relation R. However, it is not specified in detail how annotations are propagated for queries involving aggregation, set-operations or a *DISTINCT*-clause.

2.4.7 ORCHESTRA

ORCHESTRA [IKKC05, GKT$^+$07b, IGK$^+$08, GKT07a, Gre09, GKIT07] is a collaborative data sharing system (CDSS) that uses schema mappings to exchange updates between peers with different schemas. For each peer updates to other peers are translated into local updates by using the mappings between this peer's schema and the schemas of the other peers. Updates are not exchanged instantly but as batch updates on a scheduled basis. *ORCHESTRA* uses tuple-granularity *data* provenance (*How*-provenance) to store where updated data is coming from. *How* provenance is enhanced with *mapping* annotations that record which mapping has combined which tuples to generate an updated data item. These annotations are modeled as functions in the semi-ring provenance model. E.g., if a tuple is derived by mapping m_1 from source tuples t_1, t_2 through a join and by mapping m_2 from source tuple t_3 then the provenance polynomial would be $m_1(t_1 \times t_2) + m_2(t_3)$. Provenance is used in this system mainly for deciding if an update from another peer should be applied to a peer. Peers have policies that are matched against the provenance of an update to determine if an update fulfills the policies. *ORCHESTRA* provides a GUI that can be used to navigate provenance information, but no query language support for provenance data. The provenance polynomial of a tuple is recorded in an additional attribute, but it is not possible for queries to access the contents of this attribute (e.g., is tuple x in the provenance of tuple y).

2.4.8 Forward Tracing Data Lineage

Zhang et al. [KP07, ZZZP07a] compute tuple granularity provenance by rewriting SQL queries to propagate provenance information. This approach uses a variation of *Lineage-CS*. Provenance is represented as sets of tuple identifiers. The original schema of a query is extended with a single attribute that is used to store such a set. Queries are rewritten by recursively rewriting each ASPJ segment in the query to iteratively build the provenance sets. This approach is the first to consider nested sub-queries, but is restricted to uncorrelated queries and their provenance definition produces false positives if applied to nested sub-queries. This is due to the fact that the definition considers the complete provenance of a nested sub-query to belong to the provenance of the query it is used in. For example, consider an SQL sub-query like ... WHERE a IN (SELECT ...);. It is evident that for this query only tuples from the sub-query that are equal to a are necessary to fulfill the *WHERE* clause condition. Therefore, only these tuples should be included in the provenance. The representation as a set of tuple identifiers has two important drawbacks. First, the query language would have to be extended to allow for efficient querying of the provenance information. Second, in the set representation it is not clear how tuples were combined by the query to produce a result tuple.

2.5 Summary and Requirements for a relational PMS

In this section we summarize our discussion of competitive *PMS* and present requirements for a full-fledged relational *PMS*. The comparison presented in the last section reveals that the importance of provenance support has been realized by the database community, but that all of the presented systems have severe disadvantages. Table 2.1 summarizes the comparison presented in the last section. Similar arguments apply for the presented *CS* types. Most of them are defined only for a very limited subset of SQL and may not produce meaningful results when extended to support more language constructs.

A provenance management system should be developed with the goal of being useful for a wide range of application domains with different provenance needs. Neither *data* nor *transformation* provenance contain all the information needed for most provenance use cases. Thus, both provenance types should be supported by a *PMS*. It is our belief that there is no ”one-fits-all” *CS type*, because each of them exposes different information about the origin of a data item and its creation process. Therefore, a user should be able to choose between different *CS* types for both *data* and *transformation* provenance to be able to pick the one that fits his needs best. A *PMS* should be able to generate fine-grained provenance automatically for a preferably complete subset of SQL transformations. For complex queries the *PMS* should provide the functionality to compute the provenance according to an intermediate result instead of tracing back the origin of a data item to the base relations accessed by the query. A *PMS* should provide support for external provenance (provenance created manually or by another *PMS*). Ideally, the system should place no restrictions on the representation used for external provenance and should be able to process it during the automatic generation of provenance information. For instance, if a user wants to retrieve the provenance of a query over data with external provenance, the provenance computation for this query should take the external provenance into account. The mere generation and storage of provenance information is not very useful. *PMS* should provide extensive query language support for provenance data. Ideally, provenance and normal data and the associations between both can be queried using the same query language. An expressive representation of provenance information is needed to fulfill this requirement. Storage of provenance information is necessary to preserve it for later use, but provenance information can easily outgrow the normal data and therefore should only be generated when requested by the user. Below we list the requirements developed in this paragraph:

1. **Automatic generation of provenance information with various CS types**
2. **Extensive query facilities for provenance**
3. **Provenance storage and lazy computation**
4. **Partial provenance computation**
5. **Data and transformation provenance support**
6. **Support for external provenance**

With the *Perm* approach presented in this thesis we address the requirements outlined above. *Perm* generates tuple-level provenance automatically for an almost complete subset of SQL **(1)**. For instance, we support aggregation, set operations and correlated sub-queries. Provenance is computed on-demand by rewriting the query for which provenance should be computed **(3)**. The rewritten query that computes the provenance behaves like a normal SQL query and, thus, its results can be stored using the SQL *INTO* clause **(3)** or used in a sub-query **(2)** to enable SQL queries over provenance data. *Perm* supports *data* provenance with different *CS* types and *transformation* provenance **(5)**. The user can instruct the system to include external provenance and handle it as if it was created by the system itself **(6)**. In *Perm* provenance can be either traced back to the data stored in base relations of the database or to the results of a sub-query or view referenced in the query for which provenance should be computed **(4)**.

In contrast to the *PMS* presented in section 2.4, *Perm* supports a broader set of SQL language constructs and is the only system to support both *data* and *transformation* provenance. The only other system besides *Perm* that supports more than one *CS* type is *DBNotes*. We will demonstrate in chapter 5 that *Perm* is also more efficient than comparable approaches.

Table 2.1: Comparison of the Functionality provided by PMS

System	Supported Transformations	Supported CS	Provenance Representation	Computational Method	Provenance Query Support	Supported Granularity
WHIPS	Set-ASPJ	Lineage-CS	List of sets of tuples	lazy, inverse	SQL	tuple
Trio	Set-ASPJ (TriQL) with limitations	Lineage-CS	Tuple identifier pairs each associating a result tuple with provenance tuple	eager, propagation	Lineage-predicate	tuple
DBNotes	E-USPJ (psql)	Where-CS, IWhere-CS, Custom-CS	Set of attribute value identifiers	eager, propagation	GUI	attribute value
Mondrian	E-USPJ* (color-algebra)	-	-	eager, propagation	E-USPJ (color-algebra)	sets of attribute-values
MMS	No automatic provenance generation	-	-	lazy, propagation	SQL	regions that can be expressed as SQL queries
BDBMS	No automatic provenance generation	-	-	Eager, propagation	Set-ASPJ	rectangular regions over attributes and tuples
ORCHESTRA	USPJ + Recursion	How-CS	Semi-ring-polynomials over tuple identifiers	eager, propagation	GUI	tuple
Forward Tracing Data Lineage	ASPJ + uncorrelated sub-queries	Variation of Lineage-CS	Sets of tuple identifiers	eager, propagation	SQL**	tuple

*: The applied annotation propagation rules are not well-suited for provenance computation

**: The representation of the set of tuple identifiers as a single attribute value does prevent useful application of SQL on this representation

Chapter 3

Contribution Semantics

In this chapter we formally define *Perm-Influence-Contribution-Semantics* (*PI-CS*) the *contribution semantics* developed for the *Perm* system and prove several important properties of the provenance generated by this *CS* type. *PI-CS* is a type of *I-CS* based on *Lineage-CS*. We decided to develop our own type of *CS*, because the representation used by *Lineage-CS* (and also other *CS* types) is not suited for our approach to implement a "purely relational" provenance management system. Furthermore, as we will demonstrate in this chapter, *Lineage-CS*, in contrast to *PI-CS*, does not extend to queries with nested sub-queries (*sublinks*) and queries with negation. We define *C-CS* types and *transformation* provenance *CS* for the use in *Perm* as extensions of *PI-CS*.

First, we introduce an extended relational algebra which allows for a natural algebraic representation of SQL queries. Afterwards, we introduce *PI-CS*, demonstrate its applicability to the operators of the algebra, and study the relationship between this *CS* type and *Lineage-CS*. Finally, we present *C-CS* types and *transformation* provenance. Note that in this chapter we are only discussing the semantics of provenance and do not develop algorithms for generating provenance according to this semantics. Provenance computation is discussed in chapter 4.

3.1 Perm Relational Algebra

In this section we introduce notational preliminaries and the relational algebra that are needed for the theoretical foundation of *Perm*. The algebra is defined in such a way that SQL queries have natural counterpart algebra expressions and it is easy to translate between the SQL and algebra representation of a query. This property is important, because it is not feasible to build a formal framework of provenance based on SQL, but, as we will demonstrate in chapter 5, to be able to integrate provenance computation into a DBMS, the results established for algebra expressions have to be translated to SQL. As usual a relational database D is modeled as a *database schema* S and a *database instance* I. A database schema is a set of relation schemas: $S = \{\mathbf{R_1}, \ldots, \mathbf{R_n}\}$. Each relation schema is a function from a finite set $A \subset \mathscr{A}$ to the set of attribute domains $\mathscr{D}$ (we refer to the elements of this set as *data types*) where $\mathscr{A}$ is the set of possible attribute names. $\mathscr{S}$ is used to denote the set of all possible relation schemas. Every attribute domain is expected to contain the special value null: ε. Each relation schema is associated with a name by a function $Name : \mathscr{S} \rightarrow \mathscr{N}$ that assigns each relation schema in a database schema to an unique name. We assume a total order on the attribute names of a relation schema. I.e., a function $pos_{\mathbf{R}} : A \rightarrow \mathbb{N}$ that assigns each attribute from schema $\mathbf{R}$ to a unique position from the set $1, \ldots, |\, A \,|$. We use $\mathbf{R}(a_1 : d_1, \ldots, a_n : d_n)$ as a notational shortcut for a relation schema with attributes a_1 to a_n, name R, attribute order $a_1 : 1, \ldots, a_n : n$, and domains $d_1, \ldots, d_n$.

A *database instance* $I = \{R_1, \ldots, R_n\}$ of a database schema S is a set of *relations* that contains one relation R for each relational schema $\mathbf{R}$ in S. A relation R for a relation schema $\mathbf{R}$ is a subset of $RI = d_1 \times \ldots \times d_n$ and a function $mult : RI \rightarrow \mathbb{N}$. Each element $t = (v_1, \ldots, v_n)$ in RI is called a *tuple* and the value m assigned by *mult* to t is called the multiplicity of t. This means we are using the so-called *bag*- or *multiset*-semantics where every tuple is allowed to occur more than once in a relation. The elements $v_1, \ldots, v_n$ of a tuple are called attribute values. For convenience we use t^m to denote a tuple t with multiplicity m and t as a shortcut for t^1.

If q is an algebra expression, then $\mathbf{Q}$ denotes the schema of the result relation produced by evaluating q^1. We use $[[q]](I)$ to denote the result of evaluating algebra expression q over the database instance I. The database instance is omitted if it is clear from the context or irrelevant to the discussion. Q is used as a shortcut for $[[q]]$. The *Perm* algebra includes the standard operators of the relational algebra. The evaluations of all algebra operators are presented in Figure 3.1. To simplify the definition of some operators negative or zero multiplicities of tuples indicate that a tuple does not belong to a relation.

Nullary Operators: A relation access is denoted by the name of the accessed relation. We allow for construction of singleton relations containing only a constant tuple t denoted by t.

Unary Operators: **Duplicate removal** $\delta(q_1)$ eliminates duplicates from its input (in other words it sets the multiplicity of every tuple to one). **Selection** $\sigma_C(q_1)$ returns all tuples t from Q_1 that fulfill the selection condition C (written as $t \models C$). A selection condition is an expression build from attributes, constants, comparisons (e.g., equality, less than, ...), function calls, and logical operators ($\neg, \wedge, \vee, \ldots$). C is restricted to return a boolean result. In addition we allow for conditional expressions: $if\ (e_1)\ then\ (e_2)\ else\ (e_3)$ evaluates to e_2 if e_1 evaluates to true. Otherwise it evaluates to e_3. This is similar to the *CASE* construct in SQL. The algebra defines two versions of **Projection**. One duplicate preserving version (Π^B: the superscript B stands for *bag*) and one duplicate removing version (Π^S: the superscript S stands for *set*). The duplicate preserving version $\Pi^B{}_A(q_1)$ returns the results of evaluating all projection expressions from the list $A = (a_1, \ldots, a_m)$ for each tuple in Q_1. Projection expressions are similar to selection conditions except that they are not restricted to return a boolean result and that the outermost construct in a projection expression can be a renaming $e \to a$ that causes the attribute which stores expression e to be named a in the result schema. The duplicate removing version of projection ($\Pi^S{}_A(q_1)$) is defined as the application of the duplicate removal operator to the result of the duplicate preserving projection. Sometimes we use Π to denote the duplicate preserving version of projection. **Aggregation** $\alpha_{G,agg}$ groups its input on a list of group-by attributes and computes the aggregation functions from the list *agg* of aggregation functions for each group. One output tuple is produced for each group that contains the values of the group-by attributes for this group and the results of the aggregation functions [2]. In the definition presented in Figure 3.1 agg_i is one aggregation function from the list *agg* and B_i is the attribute used as input to aggregation function agg_i [3].

Join Operators: The *Perm* algebra includes several join operators. The **Cross product** $q_1 \times q_2$ is defined as in standard relational algebra. In the definition (t_1, t_2) denotes the concatenation of tuples t_1 and t_2. **Inner Join** $q_1 \bowtie_C q_2$ is a shortcut for applying a selection with condition C to the result of the cross product between q_1 and q_2. Three outer join types are defined in the algebra: **Left outer join** (⟕), **Right outer join** (⟖), and **Full outer join** (⟗). The outer join types are based on the inner join, but preserve tuples that are not joined with any other tuple. As the names indicate left outer join preserves only tuples from its left input, right outer join preserves only tuples from its right input, and full outer join preserves tuples from both inputs. $null(\mathbf{Q})$ denotes a tuple with schema $\mathbf{Q}$ and all attributes values set to *null*.

Set Operators: The algebra supports the three standard set operations **union** ($\cup$), **intersection** ($\cap$), and **set difference** ($-$). Like the projection operator, set operations are provided as a duplicate preserving and duplicate removing version (denoted by S and B).

[1] Defining $\mathbf{Q}$ independent of an database instance is valid, because the result schema of an algebra expression only depends on the database schema over which it is defined.

[2] We define the semantics of the standard aggregation functions $sum, avg, count, \ldots$ as in SQL. I. e., applying count to an empty relation returns zero and applying the other aggregation functions to an empty relations returns *null*

[3] Note that allowing only a single attribute as input of an aggregation function and only group-by attributes instead of group-by expressions does not limit the expressive power of the algebra. Expressions like $\alpha_{c*d,sum(a+b)}(q_1)$ can be written as $\alpha_{g_1,sum(agg_1)}(\Pi_{c*d \to g_1, a+b \to agg_1}(q_1))$ in our algebra. For brevity, we will use the first notation when appropriate.

Nullary Operators

$$\begin{aligned} [[t]] &= \{t\} \\ [[R]] &= \{t^n \mid t^n \in R\} \end{aligned}$$

Unary Operators

$$\begin{aligned} [[\delta(q_1)]] &= \{t \mid t^n \in Q_1\} \\ \big[\big[\Pi^B{}_A(q_1)\big]\big] &= \{t' = (v_1, \dots, v_m)^{sum} \mid sum = \sum_{t^n \in Q_1, t.A = t'} (n)\} \text{ for } A = (A_1, \dots, A_m) \\ \big[\big[\Pi^S{}_A(q_1)\big]\big] &= \delta(\Pi^B{}_A(q_1)) = \{a = (a_1, \dots, a_m)^1 \mid t^n \in Q_1 \wedge t.A = a\} \text{ for } A = (A_1, \dots, A_m) \\ [[\sigma_C(q_1)]] &= \{t^n \mid t^n \in Q_1 \wedge t \models C\} \\ [[\alpha_{G,agg}(q_1)]] &= \{(t.G, res_1, \dots, res_n)^1 \mid t \in Q_1 \wedge \forall i \in \{1,n\} : res_i = agg_i(\Pi^B{}_{B_i}(\sigma_{G=t.G}(q_1))\} \end{aligned}$$

Join Operators

$$\begin{aligned} [[q_1 \times q_2]] &= \{(t_1, t_2)^{n \times m} \mid t_1{}^n \in Q_1 \wedge t_2{}^m \in Q_2\} \\ [[q_1 \Join_C q_2]] &= \{t^{n \times m} \mid t^{n \times m} \in q_1 \times q_2 \wedge t \models C\} \\ [[q_1 ⟕_C q_2]] &= \{(t_1, t_2)^{n \times m} \mid (t_1, t_2)^{n \times m} \in [[q_1 \Join_C q_2]]\} \\ &\cup \{(t_1, null(\mathbf{Q_2}))^n \mid t_1{}^n \in Q_1 \wedge (\nexists t_2 \in Q_2 : (t_1, t_2) \models C)\} \\ [[q_1 ⟖_C q_2]] &= \{(t_1, t_2)^{n \times m} \mid (t_1, t_2)^{n \times m} \in [[q_1 \Join_C q_2]]\} \\ &\cup \{(null(\mathbf{Q_1}), t_2)^n \mid t_2{}^n \in Q_2 \wedge (\nexists t_1 \in Q_1 : (t_1, t_2) \models C)\} \\ [[q_1 ⟗_C q_2]] &= \{(t_1, t_2)^{n \times m} \mid (t_1, t_2)^{n \times m} \in [[q_1 \Join_C q_2]]\} \\ &\cup \{(null(\mathbf{Q_1}), t_2)^n \mid t_2{}^n \in Q_2 \wedge (\nexists t_1 \in Q_1 : (t_1, t_2) \models C)\} \\ &\cup \{(t_1, null(\mathbf{Q_2}))^n \mid t_1{}^n \in Q_1 \wedge (\nexists t_2 \in Q_2 : (t_1, t_2) \models C)\} \end{aligned}$$

Set Operators

$$\begin{aligned} \big[\big[q_1 \cup^S q_2\big]\big] &= \{t \mid t^n \in Q_1 \vee t^m \in Q_2\} \\ \big[\big[q_1 \cap^S q_2\big]\big] &= \{t \mid t^n \in Q_1 \wedge t^m \in Q_2\} \\ \big[\big[q_1 -^S q_2\big]\big] &= \{t \mid t^n \in Q_1 \wedge t^m \notin Q_2\} \\ \big[\big[q_1 \cup^B q_2\big]\big] &= \{t^{n+m} \mid t^n \in Q_1 \wedge t^m \in Q_2\} \\ \big[\big[q_1 \cap^B q_2\big]\big] &= \{t^{min(n,m)} \mid t^n \in Q_1 \wedge t^m \in Q_2\} \\ \big[\big[q_1 -^B q_2\big]\big] &= \{t^{n-m} \mid t^n \in Q_1 \wedge t^m \in Q_2\} \end{aligned}$$

Sublink Expressions

$$\begin{aligned} [[e\ IN\ q_{sub}]] &= \exists t \in Q_{sub} : t = e & [[e\ NOTIN\ q_{sub}]] &= \neg\exists t \in Q_{sub} : t = e \\ [[e\ op\ ANY\ q_{sub}]] &= \exists t \in Q_{sub} : e\ op\ t & [[q_{sub}]] &= Q_{sub} \\ [[e\ op\ ALL\ q_{sub}]] &= \forall t \in Q_{sub} : e\ op\ t & [[\ EXISTS\ q_{sub}]] &= \exists t \in Q_{sub} \end{aligned}$$

Figure 3.1: Perm Relational Algebra

Description	**Shortcut**
Relation	$R, S, T \ldots$
Attribute	$a, b, c \ldots$
List or Set of Attributes	$A, B, \ldots$
Renaming attribute a to b	$a \rightarrow b$
Shortcut for comparing all attributes from a list $A = (a_1, \ldots, a_n)$ with attributes from a list $B = (b_1, \ldots, b_n)$	$A = B := a_1 = b_1 \wedge \ldots \wedge a_n = b_n$
Shortcut for renaming all attributes from a list $A = (a_1, \ldots, a_n)$ to attributes from a list $B = (b_1, \ldots, b_n)$	$A \rightarrow B := a_1 \rightarrow b_1, a_2 \rightarrow b_2, \ldots a_n \rightarrow b_n$
Concatenation of two tuples t_1 and t_2	(t_1, t_2)
Algebra expression	q
Result relation generated by evaluation of algebra expression q	Q or $[[q]]$
Schema of an relation R or of the result of evaluating algebra expression q	**R** and **Q**
Null-value	ε

Figure 3.2: Notational Conventions for the Relational Model and *Perm* Algebra

Sublink Expressions: SQL allows for nested sub-queries in, e.g., the *WHERE* clause. To be able to represent such sub-queries (which we refer to as *sublinks*) in the *Perm* algebra we introduce nesting expressions that resemble the nesting constructs of SQL (*ALL*, *ANY*, *IN*, *EXISTS*, and *scalar sublink*). Similar approaches have been presented in [BM95, AB03]. The $EXISTS\ q_{sub}$ expression evaluates to true, iff Q_{sub} contains at least one tuple. If an algebra expression q_{sub} is directly applied in an projection expressions or selection predicate, we call it a *scalar sublink*. A *scalar sublink* q_{sub} evaluates to the result of evaluating the algebra expression ($[[q_{sub}]]$). This kind of sublink is only defined if q_{sub} returns at most one tuple and $\mathbf{Q_{sub}}$ contains only a single attribute. If Q_{sub} returns the empty set, then this nested expression evaluates to ε. The sublink expression $e\ op\ ANY\ q_{sub}$ evaluates to true if for at least one tuple t from Q_{sub} the expression $e\ op\ t$ evaluates to true. Here op represents an arbitrary comparison operator. This nested expression is only defined if Q_{sub} contains a single attribute with a data type that is comparable to the result type of expression e [4]. The counterpart of the *ANY*-expression is the *ALL*-expression. $e\ op\ ALL\ q_{sub}$ evaluates to true, iff every tuple t from Q_{sub} fulfills the condition $e\ op\ t$. An ALL-sublink expression evaluates to true if Q_{sub} is the empty set. An ANY-sublink expression evaluates to false if $Q_{sub} = \emptyset$. Two additional nested expressions are provided for convenience: $e\ IN\ q_{sub}$ which is equivalent to $e\ =\ ANY\ q_{sub}$ and $e\ NOTIN\ q_{sub}$ which is equivalent to $\neg(e\ =\ ANY\ q_{sub})$.

As in SQL we allow for correlations between *sublinks* and the algebra expression they are used in (called outer expression or regular input of an operator). A *correlation* is a reference to an attribute of the outer expression from inside the *sublink*. For instance, in the expression $\sigma_{EXISTS\ \sigma_{S.b=R.a}(S)}(R)$ the attribute reference $R.a$ is a correlation, because it references an attribute from relation R that is the regular input of the selection. Sublinks with correlations are evaluated using so-called *nested iteration*. Nested iteration evaluates the sublink expression separately for each tuple from the regular input of the operator. We call a sublink expression *correlated* if it contains correlations and *uncorrelated* otherwise. A sublinks expression that contains another sublink expression is called *nested*. In spite of the fact that SQL supports sublinks in all clauses we limit the use of sublinks to projection and selection to simplify the provenance computation for these expressions and because this restriction does not limit the expressive power of the algebra. E.g., the following algebra expression $R \bowtie_{a\ IN\ \Pi_C(T)} S$ is equivalent to $\sigma_{a\ IN\ \Pi_C(T)}(R \times S)$ and $\alpha_{EXISTS\ (S), sum(a)}(R)$ is equivalent to $\alpha_{new, sum(a)}(\Pi^B{}_{EXISTS\ (S) \rightarrow new, a}(R))$.

Figure 3.2 shows notational conventions for algebra expressions that we will use throughout this thesis. Most of these shortcuts have already been used in the definition of the algebra.

[4] In principle a type system would be needed to decide if two data types are comparable. We do not formally define such a type system for the *Perm* algebra because it is not needed in the definition and discussion of contribution semantics and would needlessly increase the complexity of the algebra.

person

SSN	name
1-1	Peter Peterson
2-4	Jens Jensen
5-6	Knut Knutsen

newspaper

newsId	name	publisher
1	NZZ	IEEE
2	20 Minuten	Springer

reads

pSSN	nNewsId
1-1	1
1-1	2
2-4	1

$$q_1 = \sigma_{\neg\, EXISTS\, (q_{sub})}(person) \qquad q_{sub} = \sigma_{\neg\, EXISTS\, (\sigma_{pSSN=SSN \wedge newsId=nNewsId}(reads))}(newspaper)$$

$$q_2 = \alpha_{name,count(*)}(reads \bowtie_{pSSN=SSN} person)$$

$$q_3 = \Pi^S_{person.name \rightarrow person, newspaper.name \rightarrow paper}(person \mathbin{⟕}_{SSN=pSSN} (reads \bowtie_{nNewsId=newsId} newspaper))$$

```
q1 = SELECT * FROM person
     WHERE NOT EXISTS
         (SELECT * FROM newspaper
          WHERE NOT EXISTS
               (SELECT * FROM reads WHERE pSSN = SSN AND nNewsId = newsId));

q2 = SELECT name, count(*) FROM reads, person WHERE pSSN = SSN GROUP BY name;

q3 = SELECT p.name a AS person, n.name AS paper
     FROM person p LEFT JOIN
           (reads r JOIN newspaper n ON (nNewsId = newsId)) ON (SSN = pSSN);
```

Q₁

SSN	name
1-1	Peter Peterson

Q₂

name	count
Peter Peterson	2
Jens Jensen	1

Q₃

person	paper
Peter Peterson	NZZ
Peter Peterson	20 Minuten
Jens Jensen	NZZ
Knut Knutsen	*NULL*

Figure 3.3: Example Algebra Expressions and Evaluations

Example 3.1. *Figure 3.3 presents some example algebra expressions, equivalent formulations in SQL, and the results of evaluating them over an example database instance. The example database models newspapers, persons, and which person reads which newspapers. Query q_1 from the example returns the persons that are reading all newspapers stored in the database. This query can be expressed in SQL as a nested NOT EXISTS: Return all persons for whom no newspaper exists that is not read by this person. In the algebra this query is expressed using a nested EXISTS sublink expression in the condition of the selection operator. Query q_2 returns the number of newspapers read by each person. In SQL this query is expressed using the standard aggregation function count grouping on the person relation's name attribute. Hence, the equivalent algebra expression uses the aggregation operator. Query q_3 returns all persons and the newspapers they are reading. An outer join is used to also return persons that do not read any newspapers. The algebra version of q_3 is an example for the application of renaming in projection expressions.*

3.2 Influence Data Provenance Contribution Semantics

In this section we formally define *PI-CS* (*Perm Influence Contribution Semantics*), the main *data* provenance *CS* type implemented in *Perm*. We first introduce *Lineage-CS*, the *I-CS* type on which *PI-CS* is based on, and outline its shortcomings that constitute the motivation for extending this *CS* type for the use in *Perm*. Afterwards, we iteratively define and refine *PI-CS* , by porting the *Lineage-CS* definition for a more suited relational representation and addressing its problems with nested subqueries and other types of algebra expressions. For both *CS* types we first present a declarative definition and then prove the equivalence of this definition to a provenance construction that is based on the structure of algebra expressions. Finally, we compare the expressiveness of *Lineage-CS* and *PI-CS*.

3.2.1 Lineage Contribution Semantics

We base the *I-CS* definition used in *Perm* on *Lineage-CS* presented in [CW00a], because this definition has several advantages over alternative *I-CS* types. First, users tend to intentionally express queries in a certain way and, therefore, the strong dependency of *Lineage-CS* on the syntactical structure of a query is an advantage (see 2.3). Second, this *CS* type is defined for a larger set of algebra operators than other approaches. Third, provenance is defined for single algebra operators which allows easy extension to new algebra operators. We already discussed some of the properties of *Lineage-CS* in section 2.3. Here we extend this discussion and provide a formal definition of *Lineage-CS*. The definition below is taken from [CW00a] with the notation adapted to our conventions:

Definition 3.1 (Lineage-CS)**.** *For an algebra operator op with inputs $Q_1,\dots,Q_n$ from a database instance I and a tuple $t \in op(Q_1,\dots,Q_n)$ a list $\mathscr{W}(op,I,t) =< Q_1^*,\dots,Q_n^* >$ with $Q_i^* \subseteq Q_i$ is the* witness set *of t if it fulfills the following conditions:*

$$[[op(\mathscr{W}(op,I,t))]] = \{t^x\} \qquad \textbf{(1)}$$

$$\forall i,t' \in Q_i^* : \big[\big[op(< Q_1^*,\dots,Q_{i-1}^*,\{t'\},Q_{i+1}^*,\dots >)\big]\big] \neq \emptyset \qquad \textbf{(2)}$$

$$\neg\exists \mathscr{W}' \subseteq < Q_1,\dots,Q_n >: \mathscr{W}' \supset \mathscr{W}(op,I,t) \wedge \mathscr{W}' \models (1),(2) \qquad \textbf{(3)}$$

The first condition **(1)** in Definition 3.1 checks that the witness set produces exactly t and nothing else by evaluating operator op over the witness set. The second condition **(2)** checks that each tuple t' in the witness set contributes to t and, therefore, guarantees that no superficial tuples are included in the witness set. Finally, the third condition **(3)** checks that the witness set is the maximal list with these properties, meaning that no tuples that contribute to t are left out. Note that in condition 3 two lists of sets are compared according to their subsumption relationship ($\supset$). Below we formalize the notion of subsumption for lists of sets:

Definition 3.2 (List Subsumption)**.** *A list of sets U subsumes a list of sets V ($U \supset V$), iff both lists have the same length ($\mid U \mid = \mid V \mid$), each set in U contains the elements from the corresponding set in V, and at least one set from U contains an element that is not included in the corresponding set from V:*

$$U \supset V \Leftrightarrow (\mid U \mid = \mid V \mid) \wedge (\forall i \in \{1,\dots,\mid V \mid\} : U_i \supseteq V_i) \wedge (\exists i \in \{1,\dots,\mid V \mid\} : U_i \supset V_i)$$

As mentioned in section 2.3 we omit the instance I for a witness set if it is clear from the context. *Lineage-CS* was originally studied for selection, projection, join, cross product, aggregation, union and set difference. In our discussion we include also intersection and outer joins. Note that the definition presented here is defined for set semantics and under this semantics it was proven that $\mathscr{W}(q,t)$ is unique. We postpone the discussion of bag semantics to later in this section.

Example 3.2. *As an example for Lineage-CS provenance consider the algebra expression* $q = \Pi^{B}{}_{a}(R)$ *over the relation presented below.*

R

a	*b*
1	*2*
2	*3*

Q

a
1
2

The Lineage-CS provenance of the result tuple $t = (1)$ *from* q *is as follows:*

$$\mathscr{W}(q,(1)) =< \{(1,2)\} >$$

The first condition of definition 3.1 is obviously fulfilled. The result of evaluating q *over the set* $\{(1,2)\}$ *is a relation that only contains tuple* (1). *The second condition is trivially fulfilled, because the provenance contains only a single tuple. For the third condition we have to check that no super set of* $\mathscr{W}$ *fulfills conditions 1 and 2. In this case* $\mathscr{W}' = \{(1,2),(2,3)\}$ *the only super-set of* $\mathscr{W}$ *does not fulfill condition 1, because the result of applying* q *to* $\mathscr{W}'$ *contains tuple* $(2) \neq t$.

3.2.1.1 Transitivity and Sets of Output Tuples

Lineage-CS defines provenance to be transitive. I.e., if tuple t is in the provenance of tuple t' according to an operator op_1 and t' is in the provenance of t'' according to some operator op_2, then t belongs to the provenance of t'' according to $q = op_2(op_1)$. Therefore, the witness set of an algebra expression q is computed by recursively applying Definition 3.1 to each operator in q. An advantage of *Lineage I-CS* is that the focus on a single operator leads to a simple evaluation strategy and the witness set of each operator can be studied independently of the witness set of other operators. [CW00a] also defines the provenance of a set T of result tuples according to *Lineage-CS* as:

$$\mathscr{W}(q,T) = \bigsqcup_{t \in T} \mathscr{W}(q,t)$$

Here $\bigsqcup$ stands for the element-wise union of two lists. E.g.:

$$< \{a\},\{b\} > \sqcup < \{c\},\{d\} > = < \{a,c\},\{b,d\} >$$

3.2.1.2 Bag Semantics

Lineage-CS was also extended for bag semantics. Under bag semantics two duplicates of a tuple cannot be distinguished, therefore it is impossible to determine from which duplicate a result tuple is derived. Consider the query $q = R -^{B} S$ over the relations $R = \{(1)^2\}$ and $S = \{(1)\}$. The result tuple (1) could be either derived from the first or the second tuple in R. Furthermore, if the result of an algebra expression contains a tuple t with a multiplicity greater than one, each duplicate of t might have been derived from different input tuples and its not clear which duplicate of t should be associated with which input tuple. As an example for this problem consider expression $q = \Pi^{B}{}_{a}(R)$ over relation $R = \{(1,2),(1,3)\}$ with schema $\mathbf{R} = (\mathbf{a},\mathbf{b})$. If definition 3.1 is applied to this query, then there are two sets $\mathscr{W}(q,(1))_1 =< \{(1,2)\} >$ and $\mathscr{W}(q,(1))_2 =< \{(1,3)\} >$ that fulfill the conditions of the definition. Cui et al. present two solutions to this problem. One called the *derivation set* is the set of all possible witness sets $\mathscr{W}(q,t)$ and the second one called the *derivation pool* is generated by computing the bag union of the individual Q_i^* elements of all possible witness sets $\mathscr{W}(q,t)$. The derivation set of the example query above would be $\{\mathscr{W}(q,(1))_1, \mathscr{W}(q,(1))_2\}$ and the derivation pool would be $< \{(1,2),(1,3)\} >$. The derivation set has the disadvantage that its size is proportional to the number of different derivations of a result tuple. Therefore, we use only the derivation pool. The definition we have given for *Lineage-CS* generates the derivation pool.

$$
\begin{aligned}
\mathscr{W}(R,t) &= <\{u^n \mid u^n \in R \wedge u = t\}> \\
\mathscr{W}(\sigma_C(q_1),t) &= <\{u^n \mid u^n \in Q_1 \wedge u = t\}> \\
\mathscr{W}(\Pi_A(q_1),t) &= <\{u^n \mid u^n \in Q_1 \wedge u.A = t\}> \\
\mathscr{W}(\alpha_{G,agg}(q_1),t) &= <\{u^n \mid u^n \in Q_1 \wedge t.G = u.G\}> \\
\mathscr{W}(q_1 \bowtie_C q_2,t) &= <\{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{u^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\}> \\
\mathscr{W}(q_1 ⟕_C q_2,t) &= \begin{cases} <\{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, Q_2> & \text{if } t \not\models C \\ <\{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{u^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\}> & \text{else} \end{cases} \\
\mathscr{W}(q_1 ⟖_C q_2,t) &= \begin{cases} <Q_1, \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_2}\}> & \text{if } t \not\models C \\ <\{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{u^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\}> & \text{else} \end{cases} \\
\mathscr{W}(q_1 ⟗_C q_2,t) &= \begin{cases} <\{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, Q_2> & \text{if } t \not\models C \wedge t.\mathbf{Q_2} \text{ is } \varepsilon \\ <Q_1, \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_2}\}> & \text{if } t \not\models C \wedge t.\mathbf{Q_1} \text{ is } \varepsilon \\ <\{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{u^n \mid u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\}> & \text{else} \end{cases} \\
\mathscr{W}(q_1 \cup q_2,t) &= <\{u^n \mid u^n \in Q_1 \wedge u = t\}, \{u^n \mid u^n \in Q_2 \wedge u = t\}> \\
\mathscr{W}(q_1 \cap q_2,t) &= <\{u^n \mid u^n \in Q_1 \wedge u = t\}, \{u^n \mid u^n \in Q_2 \wedge u = t\}> \\
\mathscr{W}(q_1 - q_2,t) &= <\{u^n \mid u^n \in Q_1 \wedge u = t\}, \{u^n \mid u^n \in Q_2 \wedge u \neq t\}>
\end{aligned}
$$

Figure 3.4: Compositional Semantics for *Lineage-CS*

3.2.1.3 Compositional Semantics of *Lineage-CS*

To determine the provenance of an algebra expression using the conditions of definition 3.1 can be cumbersome, because the definition only states which conditions have to be fulfilled by the provenance, but not how to construct the provenance. Cui et. al presented how to generate *Lineage-CS* provenance using a construction based on the syntactical structure of an algebra expression. In the following we refer to the conditions of definition 3.4 as the *declarative semantics* of *Lineage-CS* and the semantics defined by the set construction as the *compositional semantics* of *Lineage-CS*. The construction rules for the *compositional semantics* are presented in Figure 3.4.

Example 3.3. *For example, the witness set of an output tuple t of a selection is always the singleton set containing t, because selection outputs unmodified input tuples. An output tuple t from an aggregation is derived from a set of input tuples that belong to the same group (have the same grouping attribute values as t).*

Example 3.4. *Figure 3.5 presents some examples of provenance according to Lineage-CS. Query q_a is an example for the representation of duplicate removal. The result tuple (1) from this query was generated from two result tuples of the inner join between R and S. Query q_b demonstrates the inclusion of the complete right input of the left join in the provenance for tuples that do not have join partners. An example for aggregation is given with query q_c. Note that all tuples from relation R with the same grouping attribute value 1 are in the provenance of the result tuple $(1,5)$. Both queries q_d and q_e illustrate the provenance for set difference operations. Query q_d is similar to an example from [Cui02] that was used to show that tuples from the right input of a set difference can contribute to the result of this operator. Tuple (2) from relation T belongs to the provenance of result tuple $u = (2)$, because it indirectly contributed to u by removing tuple (2) from the result of $(S - T)$. If t had not been in relation T, then u would not be in the result of q_d. Query q_e is a counterexample to this form of reasoning. According to Lineage-CS tuple $v = (2)$ from relation U is not in the provenance of result tuple $x = (2)$, but if we apply the same reasoning as used for query q_d it contributes to x.*

As apparent form the example presented above, the problem with *Lineage-CS* for set difference is that it is defined for single algebra operators. But to distinguish between the cases presented with query q_d and the one of query q_e, information about the context an operator is used in is needed. In spite of the fact that *Why-CS* is defined for an algebra expression it does not solve this problem, because it generates the same provenance for both cases.

Cui et. al. proved that the *compositional semantics* of *Lineage-CS* is equivalent to the *declarative semantics* from definition 3.1. Recall that *Lineage-CS* was originally not studied for intersection and outer join types. Therefore, we now prove the equivalence of provenance according to definition 3.1 and the compositional semantics for these operators. The interested reader is referred to [Cui02] for the proofs for the remaining operators.

Theorem 3.1 (Equivalence of Declarative and Compositional Semantics of *Lineage-CS*)**.** *The compositional and declarative semantics of Lineage-CS are equivalent.*

Proof. We only prove this theorem for the cases not handled in [Cui02]: outer joins and intersection. Let $\mathscr{W}(op,t)$ be the witness set produced by the declarative semantics and $\mathscr{O}(op,t)$ be the witness set produced by the compositional semantics. We have to show that $\mathscr{W}(op,t) = \mathscr{O}(op,t)$ holds for $op \in \{⟕, ⟖, ⟗, \cap\}$ and all $t \in [[op]]$. This proposition can be proven by proving that $\mathscr{O}(op,t)$ fulfills conditions 1 to 3 from definition 3.1 and, thus, is indeed equal to $\mathscr{W}(op,t)$.

Case ⟕:

We present the proof for the case $t \not\models C$ because if $t \models C$ holds then the behavior and provenance of the left outer join is the same as for the inner join. Thus, $\mathscr{O}(q_1 ⟕_C q_2, t) =< \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, Q_2 >$.

Condition (1): From the definition of the left outer join we know that there is no tuple t' in Q_2 for which $(t.\mathbf{Q_1}, t') \models C$ holds, because otherwise $t \models C$ would hold. Therefore, $[[\{u^n\} ⟕_C Q_2]] = \{t^x\}$.

Condition (2): Both $\left[\left[\{u^1\} ⟕_C Q_2\right]\right] \neq \emptyset$ and $[[\{u^n\} ⟕_C \{t'\}]] \neq \emptyset$ with $t' \in Q_2$ trivially holds because for a non empty left hand side input the left join operator never produces an empty result.

Condition (3): We prove the maximality of $\mathscr{O}(q_1 ⟕_C q_2)$ by contradiction. Assume a list $\mathscr{O}' \supset \mathscr{O}$ exists that fulfills conditions 1 and 2 from the definition. Since $Q_2{}^*$ cannot be extended, $Q_1{}^*$ from $\mathscr{O}'$ has to contain at least one tuple t' that is not in $\mathscr{O}$ with $u \neq t'$. From the semantics of the left outer join follows that $[[\{t'\} ⟕_C Q_2]]$ produces either a tuple $x = (t', t'')$ with t'' from Q_2 or a tuple $x' = (t', null(\mathbf{Q_2}))$ depending on the existence of an join partner for t' in Q_2. Because $t' \neq u$ we know that neither $t = x$ nor $t = x'$ holds, and, therefore condition 1 is violated. Hence, we conclude that, since no such $\mathscr{O}'$ can exists, $\mathscr{O}$ has to be maximal.

Case ⟖:

The proof for right outer join is analog to the proof for left outer join.

Case ⟗:

For full outer join we have to distinguish two cases: ($t \not\models C \wedge t.\mathbf{Q_1}$ is ε) and ($t \not\models C \wedge t.\mathbf{Q_2}$ is ε). Both cases can be proven analog to the proof for left join.

Case $\cap$:

According to the compositional semantics $\mathscr{O}(q_1 \cap q_2, t) =< \{u^n \mid u^n \in Q_1 \wedge u = t\}, \{u^n \mid u^n \in Q_2 \wedge u = t\} >$.

Condition (1): We have to prove $[[\{u^n\} \cap \{u^m\}]] = \{t^x\}$ for some x.

$$[[\{u^n\} \cap \{u^m\}]] = [[\{t^n\} \cap \{t^m\}]] = \{t^{min(n,m)}\} \qquad \text{(definition of } \cap\text{)}$$

Condition (2): Since intersection is symmetric it suffices to show that $\left[\left[\{u^n\} \cap \{u^1\}\right]\right] \neq \emptyset$ which trivially holds:

$$\left[\left[\{u^n\} \cap \{u^1\}\right]\right] = \{t^1\} \neq \emptyset$$

Condition (3): We prove the maximality of $\mathscr{O}(q_1 \cap q_2, t)$ by contradiction. Assume a super-set $\mathscr{O}'$ of $\mathscr{O}$ exists that fulfills conditions 1 and 2 from the definition. Then $\mathscr{O}'$ has to contain a tuple $t' \neq t$ that is not in $\mathscr{O}$. w.l.o.g. assume $t' \in Q_1^*$. Then for condition 2 to hold $[[\{t'\} \cap Q_2^*]] \neq \emptyset$ has to be true. Either Q_2^* contains a tuple t'' that is equal to t', then condition 2 is fulfilled, but condition 1 is no longer fulfilled because $[[Q_1^* \cap Q_2^*]] \neq \{t\}$. Or Q_2^* does not contain such an tuple and, therefore, $[[\{t'\} \cap Q_2^*]] = \emptyset$ would hold which contradicts condition 2. Using the same reasoning as for the left outer join case we conclude that $\mathscr{O}$ has to be maximal. □

R

a	b
1	2
1	3
2	3
2	5

S

c
2
3

T

d
2
5

U

e
2
5
6

Q_a

a
1
2

Q_b

a	b	c
1	2	2
1	3	3
2	3	3
2	5	ε

Q_c

a	sum(b)
1	5
2	8

Q_d

b
2
5

Q_e

c
2
3

Q_f

a
1
2

$q_a = \Pi^S{}_a(R \bowtie_{b=c} S)$ $\quad$ $q_b = R \mathbin{\supset\mkern-8mu\bowtie}_{b=c} S$ $\quad$ $q_c = \alpha_{a,sum(b)}(R)$

$q_d = \Pi^S{}_b(R) - (S - T)$ $\quad$ $q_e = S - (T - U)$ $\quad$ $q_f = \Pi^S{}_a(R \mathbin{\supset\mkern-8mu\bowtie}_{b=c} S)$

$\mathscr{W}(q_a, (1)) =< \{(1,2),(1,3)\},\{(2),(3)\} >$

$\mathscr{W}(q_b, (2,3,3)) =< \{(2,3)\},\{(3)\} >$ $\quad$ $\mathscr{W}(q_b, (2,5,\varepsilon)) =< \{(2,5)\},\{(2),(3)\} >$

$\mathscr{W}(q_c, (1,5)) =< \{(1,2),(1,3)\} >$

$\mathscr{W}(q_d, (2)) =< \{(2,3),(2,5)\},\{(3)\},\{(2),(5)\} >$

$\mathscr{W}(q_e, (2)) =< \{(2)\},\{\},\{\} >$

$\mathscr{W}(q_f, (2)) =< \{(2,3),(2,5)\},\{(2),(3)\} >$

$\mathscr{L}(q_a, (1)) = \{< (1,2),(2) >, < (1,3),(3) >\}$

$\mathscr{L}(q_b, (2,3,3)) = \{< (2,3),(3) >\}$ $\quad$ $\mathscr{L}(q_b, (2,5,\varepsilon)) = \{< (2,5),(2) >, < (2,5),(3) >\}$

$\mathscr{L}(q_c, (1,5)) = \{< (1,2) >, < (1,3) >\}$

$\mathscr{L}(q_d, (2)) = \{< (2,3),(3),(2) >, < (2,3),(3),(5) >, < (2,5),(3),(2) >, < (2,5),(3),(5) >\}$

$\mathscr{L}(q_e, (2)) = \{< (2), \perp, \perp >\}$

$\mathscr{L}(q_f, (2)) = \{< (2,3),(2) >, < (2,5),(2) >, < (2,5),(3) >\}$

$\mathscr{DD}(q_a, (1)) = \{< (1,2),(2) >, < (1,3),(3) >\}$

$\mathscr{DD}(q_b, (2,3,3)) = \{< (2,3),(3) >\}$ $\quad$ $\mathscr{DD}(q_b, (2,5,\varepsilon)) = \{< (2,5), \perp >\}$

$\mathscr{DD}(q_c, (1,5)) = \{< (1,2) >, < (1,3) >\}$

$\mathscr{DD}(q_d, (2)) = \{< (2,3), \perp, \perp >\}$

$\mathscr{DD}(q_e, (2)) = \{< (2), \perp, \perp >\}$

$\mathscr{DD}(q_f, (2)) = \{< (2,3),(2) >, < (2,5), \perp >\}$

Figure 3.5: Provenance According to *Lineage-CS*, *WL-CS*, and *PI-CS*

3.2.2 Perm Influence Contribution Semantics

Definition 3.1 generates useful provenance information and is defined for a larger subset of relational algebra than the original version *Lineage-CS*, but there are some issues with this definition that limit its usefulness:

1. **Representation**: Modeling provenance as independent sets of tuples has the disadvantage that the information about which input tuples were combined to produce a result tuple is not modeled and that in the provenance of a set of result tuples it is not clear to which result tuple a part of the provenance belongs too.
2. **Negation**: The maximization condition (**2**) is problematic if operations with negation or non-existence checks including set difference and outer joins are involved.
3. **Sublinks**: *Lineage-CS* is not unique and produces false positives for queries that use sublink expressions. We postpone the discussion of this problem and its solution to section 3.2.3.

Representation: As an example of the first problem consider query q_a from Fig. 3.5. The tuple $t = (1)$ from the result of q_a is derived from two tuples from relations R and S (all tuples from R and S that were joined and have an a-attribute value of 1). Which tuples contributed to t is apparent from *Lineage-CS* ($\mathscr{W}(q_a,t)$), but the information about which tuple from R was joined with which tuple from S is not modeled. To record this information we change the provenance representation from a list of subsets of the input relations to a set of *witness lists*. A witness list w is an element from $(Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ with $Q_i^\varepsilon = Q_i \cup \perp$. Thus, a witness list w contains a tuple from each input of an operator or the special value $\perp$. The value $\perp$ at position i in a witness list w indicates that no tuple from the ith input relation belongs to w (and, therefore, is useful in modeling outer joins and unions). Each witness list represents one combination of input relation tuples that were used together to derive a tuple.

Definition 3.3 (Witness List)**.** *For an algebra operator op with inputs $Q_1, \ldots, Q_n$ each element w from $(Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ with $Q_i^\varepsilon = Q_i \cup \perp$ is called a potential witness list of op. We use $w[i]$ to denote the ith component (tuple) of a witness list w and $w[i-j]$ to denote a sub-list containing only the ith till the jth component of w.*

Example 3.5. *Consider the following witness list: $w =< (1),(3),(5) >$. For example the third component of w is $w[3] = (5)$ and a list containing only the last two elements of w is denoted by $w[2-3] =< (3),(5) >$.*

The modified version of Definition 3.1 using the witness list representation is presented below. We call this new contribution semantics type *Witness-List-CS* or short *WL-CS*. Note that in condition 1 of the *WL-CS* definition we use the evaluation of an operator over a set $\mathscr{L}$ of witness lists. We define this evaluation as the evaluation of the operator over reconstructed subsets of the original input relations. The reconstructed input relation subsets will contain all the tuples contained in the witness lists from $\mathscr{L}$. The notation ${Q_i}^R$ is used to denote the reconstructed subset of input relation Q_i.

Definition 3.4 (Operator Evaluation over Witness Lists)**.** *The evaluation of an operator op with inputs $Q_1, \ldots, Q_n$ over a single witness list w or a set of witness lists W from the provenance of this operator is defined as:*

$$[[op(w)]] = [[op(\{w[1]\}, \ldots, \{w[n]\})]]$$

$$[[op(W)]] = \left[\left[op({Q_1}^R, \ldots, {Q_n}^R)\right]\right]$$

$${Q_i}^R = \{t^n \mid t^n \in Q_i \wedge \exists w \in \mathscr{L} : w[i] = t\}$$

${Q_i}^R$ contains all tuples with their original multiplicity that are mentioned by at least one witness list. E.g., for a query $q = R \bowtie_{a=b} S$ over relations $R = \{(1)^2\}$ and $S = \{(1)\}$, and witness lists $w_1 =< (1),(1) >$ and $w_2 =< (1),(1) >$ the result of $q(\mathscr{L}(q,(1,1))$ would be $Q = \{(1,1)^2\}$.

Definition 3.5 (Witness-List-CS (WL-CS))**.** *For an algebra operator op with inputs $Q_1,\ldots,Q_n$, and a tuple $t \in op(Q_1,\ldots,Q_n)$ a set $\mathscr{L}(op,t) \subseteq (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ with $Q_i^\varepsilon = Q_i \cup \perp$ is the set of witness lists of t according to WL-CS if it fulfills the following conditions:*

$$[[op(\mathscr{L}(op,t))]] = \{t^x\} \qquad \textbf{(1)}$$

$$\forall w \in \mathscr{L}(op,t) : [[op(w)]] \neq \emptyset \qquad \textbf{(2)}$$

$$\neg\exists \mathscr{L}' \subseteq (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon) : \mathscr{L}' \supset \mathscr{L}(op,t) \wedge \mathscr{D}\mathscr{D}' \models (1),(2) \qquad \textbf{(3)}$$

Example 3.6. *Some examples for WL-CS are shown in Fig. 3.5. For instance, the provenance of q_a demonstrates that under WL-CS we preserve the information that tuples $(1,2)$ and (2) were joined. This fact cannot be deduced from the Lineage-CS provenance of q_a. Note that besides the representation of provenance as witness-lists WL-CS bears some similarity with Lineage-CS. For example, under Lineage-CS the provenance of tuple (1) from query q_a would be $\{\{(1,2),(2)\},\{(1,3),(3)\}\}$. Thus, for queries like this Lineage-CS also captures which tuples were used together to derive a result tuple. In contrast to Lineage-CS, our definition also captures this information for, e.g., queries that combine aggregation with set projection . For instance, the Lineage-CS provenance of tuple $t = (1)$ from the result of query $\alpha_{a,sum(b)}(R ⟕_{b=c} S)$ would be a single witness, thus, the information which tuples were joined is lost in this representation.*

Negation: As an example of the problems that arise with operators that use some form of non-existence check, consider query q_b from Figure 3.5. According to Definition 3.1, the witness list of the result tuple $t = (2,5,\varepsilon)$ contains all tuples from relation S, but in fact none of them contributed to t. Definition 3.5 does not solve this problem - the *WL-CS* provenance includes witness lists which contain the tuple $(2,5)$ paired with every tuple in S. Thus, both *Lineage-CS* and *WL-CS* do not capture an intuitive notion of influence when an operator includes negation. We believe a better semantics for the provenance of tuple $t = (2,5,\varepsilon)$ from the result of q_b would be a witness list $<(2,5),\perp>$. This indicates that $(2,5)$ paired with no tuples from S contributed to t (rather than saying that *every* value of S is in the provenance of this tuple). To achieve this semantics, we extend the *WL-CS* provenance definition with an additional condition **(4)**. This condition states that we will exclude a witness list w from the provenance, if there is a "smaller" witness list w' in the provenance that subsumes w. A witness list w is subsumed by a witness list w' (denoted by $w \prec w'$) iff w' can be derived from w by replacing some input tuples from w with $\perp$.

Definition 3.6 (Witness List Subsumption)**.** *Let w and w' be two witness lists from $\mathscr{L}(q,t)$. We define the subsumption relationship between w and w' (written as $w \prec w'$) as follows:*

$$w \prec w' \Leftrightarrow (\forall i : w[i] = w'[i] \vee w'[i] = \perp) \wedge (\exists i : w[i] \neq \perp \wedge w'[i] = \perp)$$

We use the definition of subsumption between witness lists to define a *CS* type with the desired negation semantics which we call *Perm-Influence-CS* (*PI-CS*).

Definition 3.7 (Perm-Influence-CS (PI-CS))**.** *For an algebra operator op with inputs $Q_1,\ldots,Q_n$, and a tuple $t \in op(Q_1,\ldots,Q_n)$ a set $\mathscr{D}\mathscr{D}(op,t) \subseteq (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon)$ where $Q_i^\varepsilon = Q_i \cup \perp$ is the set of witness lists of t according to PI-CS if it fulfills the following conditions:*

$$[[op(\mathscr{D}\mathscr{D}(op,t))]] = \{t^x\} \qquad \textbf{(1)}$$

$$\forall w \in \mathscr{D}\mathscr{D}(op,t) : [[op(w)]] \neq \emptyset \qquad \textbf{(2)}$$

$$\neg\exists \mathscr{D}\mathscr{D}' \subseteq (Q_1^\varepsilon \times \ldots \times Q_n^\varepsilon) : \mathscr{D}\mathscr{D}' \supset \mathscr{D}\mathscr{D}(op,t) \wedge \mathscr{D}\mathscr{D}' \models (1),(2),(4) \qquad \textbf{(3)}$$

$$\forall w, w' \in \mathscr{D}\mathscr{D}(op,t) : w \prec w' \Rightarrow w \notin \mathscr{D}\mathscr{D}(op,t) \qquad \textbf{(4)}$$

Condition **(4)** removes superfluous witness lists from the provenance of queries with outer joins and other forms of negation. However, it changes the semantics for the $\cup$ operator from that of Definition 3.1.

Under Definition 3.1, the provenance of a union result tuple would be a single witness list $< t_1, t_2 >$ if the result of the union is generated from a tuple t_1 from the left input and a tuple t_2 from the right input. We feel this is a bit misleading as it indicates that these two tuples *used together* influence t, when in fact each, independently, influences t. *PI-CS* captures this intuition by defining the provenance as $\{< t_1, \perp>, <\perp, t_2 >\}$. If the union semantics of Definition 3.1 is desired, we can easily achieve it with a simple post-processing rule to "repair" the provenance for unions. We define an operation $+$ for two witness lists w_1 and w_2 that combines them into a new witness list w by taking an input tuple from w_1 if w_2 is $\perp$ on this input and vice versa. If both w_1 and w_2 are not $\perp$ on at least one input, the operation is undefined. Notice that this post-processing does not influence the provenance of outer joins as it is defined on the provenance of a single tuple t.

$$\forall w, w' \in \mathscr{D}\mathscr{D}(q,t) : w + w' = w'' \Rightarrow w'' \in \mathscr{D}\mathscr{D}(q,t) \wedge w, w' \notin \mathscr{D}\mathscr{D}(q,t) \qquad \textbf{(U)}$$

For some cases the provenance of set difference under *WL-CS* represents the semantics of this operation more accurately than *PI-CS* (For instance, query q_d from Figure 3.5). Therefore, we implement both *PI-CS* data provenance after definition 3.7, and an alternative semantics which uses the *WL-CS* definition for union and set difference operators.

3.2.2.1 Transitivity

We define the *PI-CS* provenance of an algebra expression q containing more than one operator as recursively substituting tuples in a witness list for one of the operators in q with the witness lists for this tuple. I.e., a witness list w of an operator op_1 in q contains tuples from the input.

Example 3.7. *The provenance of an algebra expression $q = \sigma_{a=b}(R \times S)$ is computed by first computing the provenance of $q' = \sigma_{a=b}([[R \times S]])$. Assume that $\mathscr{W}(q', (1,2)) = w_1 =< (1,2) >$. $(1,2)$ is a tuple from the result of the cross product. We proceed by computing the provenance of $w_2 = (1,2)$ in $R \times S$. The provenance of this tuple is the witness list $< (1), (2) >$. Now $(1,2)$ in w_1 is replaced by the contents of w_2 resulting in the witness list $< (1), (2) >$ for $\mathscr{D}\mathscr{D}(q, (1,2))$.*

Definition 3.8 stated below defines the provenance of an algebra expression according to *PI-CS*.

Definition 3.8 (*PI-CS* for Algebra Expressions)**.** *The provenance according to PI-CS for an algebra expression $q = un(q_1)$ or $q = q_1$ bin q_2 where un is an unary operator and bin is a binary operator is defined as:*

If $q = un(q_1)$:

$$\begin{aligned}\mathscr{D}\mathscr{D}(q,t) = &\{w =< v_1, \ldots, v_n >^p \mid \exists w' =< u >^m \in \mathscr{D}\mathscr{D}(un(Q_1),t) \wedge u \in Q_1 \wedge w^p \in \mathscr{D}\mathscr{D}(q_1,u)\} \\ &\cup \{w =<\perp, \ldots, \perp>^m \mid \exists w' =<\perp>^m \in \mathscr{D}\mathscr{D}(un(Q_1),t)\}\end{aligned}$$

If $q = q_1$ bin q_2:

$$\begin{aligned}\mathscr{D}\mathscr{D}(q,t) = \{&w =< u_1, \ldots, u_n, v_1, \ldots, v_m >^{q \times r} | w' =< u, v >^p \in \mathscr{D}\mathscr{D}(Q_1 \textit{ bin } Q_2, t) \\ &\qquad \wedge u \in Q_1 \wedge w' =< u_1, \ldots, u_n >^q \in \mathscr{D}\mathscr{D}(q_1, u) \\ &\qquad \wedge v \in Q_2 \wedge w'' =< v_1, \ldots, v_m >^r \in \mathscr{D}\mathscr{D}(q_2, v)\} \\ \cup\{&w =< u_1, \ldots, u_n, \perp, \ldots, \perp>^q | w' =< u, \perp>^p \in \mathscr{D}\mathscr{D}(Q_1 \textit{ bin } Q_2, t) \\ &\qquad \wedge u \in Q_1 \wedge w' =< u_1, \ldots, u_n >^q \in \mathscr{D}\mathscr{D}(q_1, u)\} \\ \cup\{&w =<\perp, \ldots, \perp, v_1, \ldots, v_m >^q | w' =<\perp, v >^p \in \mathscr{D}\mathscr{D}(Q_1 \textit{ bin } Q_2, t) \\ &\qquad \wedge v \in Q_2 \wedge w'' =< v_1, \ldots, v_m >^q \in \mathscr{D}\mathscr{D}(q_2, v)\} \\ \cup\{&w =<\perp, \ldots, \perp>^1 | w' =<\perp, \perp>^1 \in \mathscr{D}\mathscr{D}(Q_1 \textit{ bin } Q_2, t)\}\end{aligned}$$

$$
\begin{aligned}
\mathscr{DD}(R,t) &= \{< u >^n |\ u^n \in R \wedge u = t\} \\
\mathscr{DD}(\sigma_C(q_1),t) &= \{< u >^n |\ u^n \in Q_1 \wedge u = t\} \\
\mathscr{DD}(\Pi_A(q_1),t) &= \{< u >^n |\ u^n \in Q_1 \wedge u.A = t\} \\
\mathscr{DD}(\alpha_{G,agg}(q_1),t) &= \{< u >^n |\ u^n \in Q_1 \wedge u.G = t.G\} \cup \{<\perp>|\ Q_1 = \emptyset \wedge \mid G \mid = 0\} \\
\mathscr{DD}(q_1 \bowtie_C q_2,t) &= \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} \\
\mathscr{DD}(q_1 \text{⟕}_C q_2,t) &= \begin{cases} \{< u,\perp>^n |\ u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\} & \text{if } \neg t \models C \\ \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} & \text{else} \end{cases} \\
\mathscr{DD}(q_1 \text{⟖}_C q_2,t) &= \begin{cases} \{<\perp,u >^n |\ u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\} & \text{if } \neg t \models C \\ \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} & \text{else} \end{cases} \\
\mathscr{DD}(q_1 \text{⟗}_C q_2,t) &= \begin{cases} \{< u,\perp>^n |\ u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\} & \text{if } \neg t \models C \wedge t.\mathbf{Q_2} \text{ is } \varepsilon \\ \{<\perp,u >^n |\ u^n \in Q_2 \wedge u = t.\mathbf{Q_2}\} & \text{if } \neg t \models C \wedge t.\mathbf{Q_1} \text{ is } \varepsilon \\ \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} & \text{else} \end{cases} \\
\mathscr{DD}(q_1 \cup q_2,t) &= \{< u,\perp>^n |\ u^n \in Q_1 \wedge u = t\} \cup \{<\perp,u >^n |\ u^n \in Q_2 \wedge u = t\} \\
\mathscr{DD}(q_1 \cap q_2,t) &= \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t \wedge v^m \in Q_2 \wedge v = t\} \\
\mathscr{DD}(q_1 - q_2,t) &= \{< u,\perp>^n |\ u^n \in Q_1 \wedge u = t\}
\end{aligned}
$$

Figure 3.6: Compositional Semantics for *PI-CS* for Single Operators

Though it might appear to be quite complex, this definition simply states that the witness list of an algebra expression q is created by replacing tuples in each witness list w of the outmost operator of q with the content of witness lists for these tuples (or $\perp$, if w contains $\perp$).

3.2.2.2 Compositional Semantics

Like for *Lineage-CS* we present an compositional semantics for *PI-CS* and prove its equivalence to the declarative semantics defined by Definition 3.7. The compositional semantics of *PI-CS* for each algebra operator are shown in Figure 3.6.

Example 3.8. *The PI-CS provenance $\mathscr{DD}$ of the queries from the running example is presented in Figure 3.5. Note the difference between WL-CS and PI-CS for queries with outer joins and set difference (q_b,q_d,q_e, and q_f).*

Theorem 3.2 (Equivalence of Compositional and Declarative Semantics of *PI-CS*)**.** *The compositional and declarative semantics of PI-CS are equivalent.*

Proof. Let $\mathscr{I}(op,t)$ be the witness set produced by the declarative semantics and $\mathscr{DD}(op,t)$ be the witness set produced by the compositional semantics. We have to show that $\mathscr{I}(op,t) = \mathscr{DD}(op,t)$ holds. This proposition can be proven by showing that $\mathscr{DD}(op,t)$ fulfills conditions 1 to 4 from definition 3.7 and, thus, is indeed equal to $\mathscr{I}(op,t)$.

Case $q = R$:

Obvious from the definition of R.

Case $q = \sigma_C(q_1)$:

Condition **(1)**: Because $u = t$ and t is in the result of q we know that $t \models C$. Therefore, $[[\sigma_C(u^n)]] = \{t^n\}$ and condition 1 holds.

Condition **(2)**: Using the same reasoning as for condition 1 we deduce that $[[\sigma_c(\{u\})]] = \{u\} \neq \emptyset$.

Condition **(3)**: We prove that $\mathscr{DD}(q,t)$ is maximal by contradiction. Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1, 2, and 4. Then $\mathscr{O}$ contains a witness $w =< t' >$ with $t' \neq t$. If $t' \models C$ then $\mathscr{O}$ does not fulfill condition 1. Else $t' \not\models C$ holds. Hence, $[[\sigma_C(\{t'\})]] = \emptyset$ and condition 2 is not fulfilled.

Condition **(4)**: No witness w from $\mathscr{DD}$ contains $\perp$ and, therefore, condition 4 trivially holds.

Case $q = \Pi^{S/B}{}_A(q_1)$:

Condition **(1)**: Because $u.A = t$ we conclude that $\left[\left[\Pi^S{}_A(\{u^n\})\right]\right] = \{t^1\}$ and $\left[\left[\Pi^B{}_A(\{u^n\})\right]\right] = \{t^n\}$ holds. Thus, condition 1 is fulfilled.

Condition **(2)**: Using the same reasoning as in the proof of condition 1 we conclude that $\left[\left[\Pi^{S/B}{}_A(\{u\})\right]\right] = \{t\} \neq \emptyset$.

Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1,2, and 4. Then $\mathscr{O}$ contains a witness $w =< t' >$ with $t' \notin \mathscr{DD}$. From the compositional semantics of Π we know that $t'.A \neq t.A$, because otherwise t' would be contained in $\mathscr{DD}$. Therefore, $\left[\left[\Pi^{S/B}{}_A(\{t'\})\right]\right] \neq \{t^x\}$ and condition 1 is not fulfilled by $\mathscr{O}$.

Condition **(4)**: No witness w from $\mathscr{DD}$ contains $\perp$ and, therefore, condition 4 trivially holds.

Case $\alpha_{G,agg}(q_1)$:

Condition **(1)**: Assume $\left[\left[\alpha_{G,agg}({Q_1}^R)\right]\right] \neq \{t\}$. Then $\left[\left[\alpha_{G,agg}({Q_1}^R)\right]\right]$ contains a tuple $t' \neq t$. We know $t'.G = t.G$ because of the definition of aggregation and the compositional semantics of aggregation. Therefore, $t.agg \neq t'.agg$ holds. At least for one aggregation function agg_i from agg the tuples t and t' contain a different result: $t.agg_i \neq t'.agg_i$. But from

$$t.agg_i = agg_i(\Pi^B{}_{B_i}(\sigma_{G=t.G}(q_1))) = agg_i(\Pi^B{}_{B_i}(\{u \mid u.G = t.G \wedge u \in Q_1\})) = t'.agg_i$$

follows $t = t'$.

Condition **(2)**: We have to distinguish two cases. If aggregation is used without grouping, we know from the definition of the aggregation operator that the result of this operator is never the empty set. Thus, condition 2 holds. If aggregation is used with grouping then the compositional semantics generates witness lists that have the same group-by values as one of the output tuples of the aggregation. Evaluating the aggregation over one of this witness lists generates a single output tuple with this group-by value. Therefore, condition 2 holds for this case too.

Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1, 2, and 4. Then $\mathscr{O}$ contains a witness list $w =< t' >$ with $t' \notin \mathscr{DD}$. If $t'.G \neq t''.G$ for $t'' \in \mathscr{DD}$ holds, then condition 1 is not fulfilled, because t' belongs to another group than t'' which by definition belongs to the same group as t. If t' belongs to the same group as t'' then it would be included in $\mathscr{DD}$.

Condition **(4)**: No witness list w from $\mathscr{DD}$ contains $\perp$ or the $\mathscr{DD}$ contains only a single witness list. Therefore, condition 4 trivially holds.

Case $q_1 \bowtie_C q_2$:

Condition **(1)**: From the definition of the compositional semantics of *PI-CS* provenance for the join operator we know that $t = (u,v)$ holds for each witness list $w =< u,v >$ in $\mathscr{DD}$. Applying the definition of the join operator we get $[[\{u\} \bowtie_C \{v\}]] = \{t\}$ and according to the semantics attached to computing an operator over a set of witnesses: $[[op(\mathscr{DD})]] = \{t^{n \times m}\}$.

Condition **(2)**: From the proof of condition 1 we know that $[[\{u\} \bowtie_C \{v\}]] = \{t\} \neq \emptyset$ and, therefore, condition 2 holds.

Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1,2, and 4. Then $\mathscr{O}$ contains a witness $w =< u',v' > \notin \mathscr{DD}$. We know that $(u',v') \neq t$. Either $(u',v') \models C$ which breaks condition 1 or $(u',v') \not\models C$ which breaks condition 2.

Condition **(4)**: No witness w from $\mathscr{DD}$ contains $\perp$ and, therefore, condition 4 trivially holds.

Case $q_1 ⟕ q_2$:

We present only the proof for the case $t \not\models C$ because if $t \models C$ holds then the behavior and provenance of the left outer join is the same as for the inner join.

Condition **(1)**: For each witness list $w =< u,\perp > \in \mathscr{DD}$: $[[\{u\} ⟕_C \emptyset]] = \{t\}$. Therefore, $[[op(\mathscr{DD})]] = \{t^x\}$ and condition 1 holds.

Condition **(2)**: The left join operator never produces the empty set for an non empty left input. Thus, because each witness list in $\mathscr{DD}$ is of the form $< u,\perp >$ we can deduce that condition 2 holds.

Condition **(3)**: We prove the maximality of $\mathscr{DD}$ by contradiction. Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1, 2, and 4 from the *PI-CS* definition. Then $\mathscr{O}$ contains a witness list $w' = < u', v' > \notin \mathscr{DD}$. $u' = t.\mathbf{Q_1}$ has to hold because else condition 1 would not be fulfilled. It follows that $v' \neq \perp$, because $< u', \perp > \in \mathscr{DD}$. Clearly, $w' \prec < u', \perp >$, which breaks condition 4.

Condition **(4)**: Each witness in $\mathscr{DD}$ is of the form $< u, \perp >$. Thus, for two witness lists w and w' the condition $w \prec w'$ can never be fulfilled. If follows that condition 4 holds.

Case $q_1 ⟖ q_2$:

The proof for right outer join is analog to the proof for left outer join.

Case $q_1 ⟗ q_2$:

For full outer join we have to distinguish two cases: ($t \not\models C \wedge t.\mathbf{Q_1}$ is ε) and ($t \not\models C \wedge t.\mathbf{Q_2}$ is ε). Both cases can be proven analog to the proof for left outer join.

Case $q_1 \cup q_2$:

Condition **(1)**: Each witness list w from $\mathscr{DD}$ is either $< u, \perp >$ or $< \perp, u >$ with $u = t$. Applying the definition of the union operator we get $[[\{u\} \cup \emptyset]] = [[\emptyset \cup \{u\}]] = \{t\}$. Since $[[op(\mathscr{DD})]]$ is defined as the applying op to the union of the input to $[[op(w)]]$ for $w \in \mathscr{DD}$, condition 1 holds.

Condition **(2)**: Using the fact $[[\{u\} \cup \emptyset]] = [[\emptyset \cup \{u\}]] = \{t\}$ established in the proof of condition 1 we conclude that condition 2 is fulfilled.

Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1,2, and 4. Then $\mathscr{O}$ contains a witness list $w = < u', v' >$ with $w \notin \mathscr{DD}$. If either u' or v' are neither $\perp$ nor equal to t, then condition 1 is not fulfilled. If only one of u' and v' is equal to $\perp$ then w would be in $\mathscr{DD}$. For the remaining two cases ($w = < \perp, \perp >$ and $w = < t, t >$) either condition 2 or condition 4 is not fulfilled.

Condition **(4)**: All witness lists from $\mathscr{DD}$ are either of the form $< u, \perp >$ or $< \perp, u >$. Thus, there are no two witness lists w and w' from $\mathscr{DD}$ for which the precondition $w \prec w'$ from condition 4 is fulfilled and condition 4 holds.

Case $q_1 \cap q_2$:

Condition **(1)**: Each witness list w from $\mathscr{DD}$ is of the form $w = < u, v >$ with $u = v = t$. Since $[[\{t\} \cap \{t\}]] = \{t\}$ condition 1 holds.

Condition **(2)**: Since intersection is symmetric it suffices to show that $\left[\left[\{u^n\} \cap \{u^1\}\right]\right] \neq \emptyset$ which trivially holds.

Condition **(3)**: We prove the maximality of $\mathscr{DD}$ by contradiction. Assume a super-set $\mathscr{O}$ of $\mathscr{DD}$ exists that fulfills conditions 1,2 and 4 from the definition. Then $\mathscr{O}$ has to contain a witness list that includes a tuple $t' \neq t$ that is not in $\mathscr{DD}$. w.l.o.g. assume $t' \in {Q_1}^R$. Then for condition 2 to hold $\left[\left[\{t'\} \cap {Q_2}^R\right]\right] \neq \emptyset$ has to be true. Either ${Q_2}^R$ contains a tuple t'' that is equal to t', then condition 2 is fulfilled, but condition 1 is no longer fulfilled because $\left[\left[{Q_1}^R \cap {Q_2}^R\right]\right] \neq \{t^x\}$. Or ${Q_2}^R$ does not contain such an tuple and, therefore, $\left[\left[\{t'\} \cap {Q_2}^R\right]\right] = \emptyset$ would hold which contradicts condition 2. Hence, $\mathscr{DD}$ is maximal.

Condition **(4)**: No witness list w from $\mathscr{DD}$ contains $\perp$ and, therefore, condition 4 trivially holds.

Case $q_1 - q_2$:

Condition **(1)**: Each witness list in $\mathscr{DD}$ is of the form $< u, \perp >$. Since $[[\{u\} - \emptyset]] = \{u\}$ condition 1 holds.

Condition **(2)**: Follows from the proof of condition 1.

Condition **(3)**: Assume a set $\mathscr{O} \supset \mathscr{DD}$ exists that fulfills conditions 1,2, and 4. Then $\mathscr{O}$ contains a witness $w = < u', v' >$ with $w \notin \mathscr{DD}$. $u' = t$ has to hold otherwise condition 1 would break. If $v' = \perp$ then w would be in $\mathscr{DD}$. Else, a $w' = < t, \perp > \in \mathscr{DD}$ subsumes w which would break condition 4.

Condition **(4)**: Every witness list in $\mathscr{DD}$ is of the form $< u, \perp >$. Thus, there are no two witness lists w and w' from $\mathscr{DD}$ for which the precondition $w \prec w'$ from condition 4 is fulfilled and condition 4 holds. □

3.2.3 PI-CS for Sublink Expressions

Recall that the algebra presented in section 3.1 has constructs that model sublink queries in SQL. For instance, the sublink expression *e IN* q_{sub}. We now discuss the provenance of such constructs according to *PI-CS*. This discussion is quite complex, so we proceed one step at a time by first limiting the discussion to single sublink expressions used in the condition of a selection operator and then iteratively extend the scope towards sublinks in other operators, multiple sublinks, nested sublinks, and correlated sublinks. We will demonstrate that *PI-CS* does not naturally extend to these algebra constructs, but needs to be extended to deal properly with sublinks. Note that the problems with sublinks are not caused by the representation of *PI-CS*. The same problems arises for *Lineage-CS*, the *CS* type from which *PI-CS* is derived from.

3.2.3.1 Single Sublinks in Selection Conditions

At first we discuss the provenance of selection operators containing only a single uncorrelated sublink according to *PI-CS*. For a given query q of the form $\sigma_C(q_1)$ that contains a sublink expression C_{sub} with a sublink query q_{sub} in condition C, we are trying to find a set of witness lists with tuples from Q_1 and Q_{sub} that fulfill the conditions of Definition 3.7 and, thus, form the provenance of q. Some preliminaries are needed to be able to apply Definition 3.7 to our algebra extended with sublinks. Note that the value of a sublink expression is constant for a fixed tuple from the operator's regular input and fixed tuples of all correlated relations, if the sublink is correlated. According to Definition 3.7, a set of witness lists $\mathscr{DD}$ is the provenance of a tuple $t \in Q$ iff $\mathscr{DD}$ produces t (condition 1), each witness list from $\mathscr{DD}$ produces not the empty set (condition 2), $\mathscr{DD}$ is the maximal set of witness lists with these properties (condition 3), and $\mathscr{DD}$ does not contain any subsumed witness lists.

From the definition of the selection operator we know that if a tuple t is in the result of the operator, t is also in the input and t fulfills the condition C (denoted by $t \models C$). A sublink expression C_{sub} can play different roles in the condition C of a selection according to a regular input tuple t. One possible role is that condition C is only fulfilled, iff C_{sub} is true. The second possibility is that C is true iff C_{sub} is false. The last role is that C is true independent of the result of C_{sub}[5]. We refer to these three influence roles as *reqtrue*, *reqfalse* and *ind*. Note that the inclusion of tuples from Q_{sub} in witness lists can depend on the influence role of the sublink. Therefore, we consider the provenance of each sublink type for each influence role separately.

Definition 3.9 (Sublink Influence Roles). *Let $C(t)$ for an condition C be the result of evaluating C for a regular input tuple t. The influence role $\mathscr{R}(q, C_{sub}, t)$ of a sublink expression C_{sub} used in a selection condition C of an algebra expression of the form $q = \sigma_C(q_1)$ according to an result tuple t of q is defined as:*

$$\mathscr{R}(q, C_{sub}, t) = \begin{cases} reqtrue & if\, C_{sub}(t) \Leftrightarrow C(t) \\ reqfalse & if\, \neg C_{sub}(t) \Leftrightarrow C(t) \\ ind & else \end{cases}$$

Intuitively, the influence role definition states that a sublink expression is *reqtrue*/*reqfalse* if we cannot change its result without changing the result of evaluating C.

Example 3.9. *As an example for the influence roles consider query $q = \sigma_{a\ IN\ (S) \vee a=3}(R)$ over relations $R = \{(1),(3)\}$ and $S = \{(1)\}$. The influence role of $C_{sub} = a\ IN\ (S)$ for input tuple $t = (1)$ is reqtrue, because only if C_{sub} evaluates to true then C evaluates to true. For result tuple $t' = (3)$ the influence role of C_{sub} is ind, because $a = 3$ is fulfilled and, therefore, C evaluates to true independent of the result of C_{sub}.*

ANY-Sublink Expressions in Selections

The influence roles enable us to derive the provenance for *ANY*-sublinks according to *PI-CS*, by determining for each influence role which tuples from Q_{sub} have to be included in witness lists for conditions 1 to 4 of definition 3.7 to be fulfilled. First, we introduce two auxiliary sets that will simplify the derivation of the provenance of *ANY*- and *ALL*-sublinks: ${Q_{sub}}^{true}(t)$ and ${Q_{sub}}^{false}(t)$. Both sets are parameterized by a tuple t from the input of the selection the sublink expression is used in[6]. For an *ANY*-sublink ($e\ op\ ANY\ q_{sub}$)

[5] The fourth role is that C is false independent of the the result of C_{sub}. We do not consider this role, because if C is false then t is not in the result of the selection.

[6] If made clear from the context parameter t is omitted.

ANY-Sublink Expressions

Let $q = \sigma_C(q_1)$ with C containing an uncorrelated *ANY*-sublink expression C_{sub}.

$$\mathscr{DD}(q,t) = \begin{cases} \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t \wedge v^m \in {Q_{sub}}^{true}(t)\} & \text{if } \mathscr{R}(q,C_{sub},t) = reqtrue \\ \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t \wedge v^m \in Q_{sub}\} & \text{else} \end{cases}$$

ALL-Sublink Expressions

Let $q = \sigma_C(q_1)$ with C containing an uncorrelated *ALL*-sublink expression C_{sub}.

$$\mathscr{DD}(q,t) = \begin{cases} \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t \wedge v^m \in {Q_{sub}}^{false}(t)\} & \text{if } \mathscr{R}(q,C_{sub},t) = reqfalse \\ \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t \wedge v^m \in Q_{sub}\} & \text{else} \end{cases}$$

EXISTS- or Scalar-Sublink Expressions

Let $q = \sigma_C(q_1)$ with C containing an uncorrelated *EXISTS*- or scalar-sublink expression C_{sub}.

$$\mathscr{DD}(q,t) = \{< u,v >^{n\times m} |\ u^n \in Q_1 \wedge u = t \wedge v^m \in Q_{sub}\}$$

Figure 3.7: Compositional Semantics for Single Sublink Expressions in Selections

or *ALL*-sublink ($e\ op\ ALL\ q_{sub}$) these auxiliary sets are defined as:

$$\begin{aligned} {Q_{sub}}^{true}(t) &= \{t' \mid t' \in Q_{sub} \wedge t.e\ op\ t'\} \\ {Q_{sub}}^{false}(t) &= \{t' \mid t' \in Q_{sub} \wedge \neg(t.e\ op\ t')\} \end{aligned}$$

Let us state some observations about the behavior of *ANY*-sublink expressions. An *ANY*-sublink expression C_{sub} evaluates to true if the comparison condition $e\ op\ t'$ is fulfilled for at least one tuple t' from Q_{sub}. Thus, if C_{sub} is true for a regular input tuple t, then C_{sub} is true for all subsets of Q_{sub} that contain at least one tuple from ${Q_{sub}}^{true}(t)$. If C_{sub} is false for a regular input tuple t, then C_{sub} is false for all subsets of Q_{sub}, because there are no tuples in Q_{sub} that fulfill $e\ op\ t'$. These facts can be used to derive the provenance of an *ANY*-sublink. If the sublink expression is *reqtrue*, Q_{sub} makes C_{sub} true and, therefore, having a witness list for each tuple from Q_{sub} in $\mathscr{DD}$ fulfills condition 1 of the *PI-CS* definition, but condition 2 is only fulfilled for witness lists that contain a tuple from ${Q_{sub}}^{true}(t)$. Thus, $\mathscr{DD}$ contains all witness lists of the form $< t,t' >$ with $t' \in {Q_{sub}}^{true}(t)$. If the sublink expression is *reqfalse* or *ind*, having a witness list for each tuple from Q_{sub} is the provenance of t, because conditions 1 to 4 are fulfilled for this set. Figure 3.7 presents the compositional semantics for *PI-CS* for single sublinks.

ALL-Sublink Expressions in Selections

The provenance for *ALL*-sublinks is derived analogously to the provenance for *ANY*-sublinks, except that an *ALL*-sublink uses universal quantification instead of existential quantification. An *ALL*-sublink evaluates to true if the comparison condition is fulfilled for all tuples from Q_{sub}. If C_{sub} is true then $C_{sub}(Q)$ is true for all $Q \subseteq Q_{sub}$. It follows that Q_{sub} fulfills conditions 1 and 2, if C_{sub} is *reqtrue* or *ind*. If C_{sub} is false then Q_{sub} contains at least one tuple t' that does not fulfill condition $t.e\ op\ t'$, but is allowed to contain tuples that fulfill the condition. Q_{sub} fulfills condition 1 if C_{sub} is *reqfalse*, but only tuples from ${Q_{sub}}^{false}$ fulfill condition 2. It follows that ${Q_{sub}}^{R} = {Q_{sub}}^{false}$. Recall that an *ALL*-sublink expression evaluates to true if $Q_{sub} = \emptyset$. This would cause a witness list $w =< t, \perp >$ to belong to the provenance of such a sublink according to *PI-CS*. In consequence, because of condition 4 of Definition 3.7, all witness lists of the form $< t,v >$ with $v \neq \perp$ would be removed from the provenance. To avoid this unwanted behavior we modify condition 4 to restrict its impact to the parts of a witness list that correspond to the regular inputs of an algebra operator.

R

a	b
1	1
2	1
3	2

S

c	d
1	3
2	4
4	5

Q$_a$

a	b
1	1
2	1

Q$_b$

b	c
4	5

Q$_c$

a	b
2	1
3	2

$q_a = \sigma_{a\,=\,ANY\ (\Pi_c(S))}(R)$ $\quad q_b = \sigma_{c\,>\,ALL\ (\Pi_a(R))}(S)$ $\quad q_c = \sigma_{(a=3)\vee\neg(a\,<\,ALL\ (\sigma_{c\neq 1}(\Pi_c(S))))}(R)$

$\mathscr{DD}(q_a,(1,1)) = \{<(1,1),(1,3)>\}$ $\quad \mathscr{DD}(q_a,(2,1)) = \{<(2,1),(2,4)>\}$

$\mathscr{DD}(q_b,(4,5)) = \{<(1,1),(4,5)>,<(2,1),(4,5)>,<(3,2),(4,5)>\}$

$\mathscr{DD}(q_c,(2,1)) = \{<(2,1),(2,4)>\}$ $\quad \mathscr{DD}(q_c,(3,2)) = \{<(3,2),(2,4)>,<(3,2),(4,5)>\}$

Figure 3.8: Examples for the Provenance of Single Uncorrelated Sublink Expressions

Definition 3.10 (Modified PI-CS)**.** *For an algebra operator op with regular inputs $Q_1,\dots,Q_n$ and sublink expressions $C_{sub_1},\dots,C_{sub_m}$ and a tuple $t \in op(Q_1,\dots,Q_{sub_m})$ a set $\mathscr{DD}(op,t) \subseteq (Q_1^\varepsilon \times \dots \times Q_{sub_m}{}^{\varepsilon})$ is the set of witness lists of t according to PI-CS if it fulfills the following conditions:*

$$[[op(\mathscr{DD}(op,t))]] = \{t\} \qquad \textbf{(1)}$$
$$\forall w \in \mathscr{DD}(op,t) : [[op(w)]] \neq \emptyset \qquad \textbf{(2)}$$
$$\neg\exists \mathscr{DD}' \subseteq (Q_1^\varepsilon \times \dots \times Q_{sub_m}{}^{\varepsilon}) : \mathscr{DD}' \supset \mathscr{DD}(op,t) \wedge \mathscr{DD}' \models (1),(2),(4) \qquad \textbf{(3)}$$
$$\forall w,w' \in \mathscr{DD}(op,t) : w \prec_n w' \Rightarrow w \notin \mathscr{DD}(op,t) \qquad \textbf{(4)}$$

with

$$w \prec_n w' := w[1-n] \prec w'[1-n] \wedge \forall i \in \{1,\dots,m\} : w[n+i] = w'[n+i]$$

EXISTS-Sublink Expressions in Selections

For the provenance derivation of *EXISTS*-sublinks we can use the fact that an *EXISTS* sublink evaluates to true iff Q_{sub} produces a result with at least one tuple. Thus, $C_{sub}(Q)$ is true if Q is a non-empty subset of Q_{sub}. If C_{sub} is *reqtrue* Q_{sub} fulfills condition 1 and condition 2. If C_{sub} is *reqfalse* then $Q_{sub} = \emptyset$ and, thus, Q_{sub} fulfills conditions 1 and 2. In summary, $Q_{sub}{}^R = Q_{sub}$ independently of the influence role.

Scalar Sublink Expressions in Selections

The provenance derivation for scalar sublinks is trivial, because a scalar sublink either produces a single result tuple or the empty set. In both cases condition 1 is fulfilled for $Q_{sub}{}^R = Q_{sub}$. Condition 2 and 3 follows from condition 1, because Q_{sub} contains at most one tuple. Therefore, $Q_{sub}{}^R = Q_{sub}$.

Example 3.10. *The provenance of three example algebra expressions is given in Figure 3.8. For q_a, the sublink expression $C_{sub} = (a\ =\ ANY\ \Pi_c(S))$ is reqtrue for all input tuples t from R and for each t, only one tuple t' from S is in $(\Pi_c(S))^{true}(t)$. In query q_b the ALL-sublink is also reqtrue. For tuple $(4,5)$, the only tuple from S that fulfills condition C, all tuples from relation R are included in the provenance of $(4,5)$. In Query q_c the sublink expression is reqfalse for input tuple $(2,1)$ and reqind for input tuple $(3,2)$.*

We now formally prove that the compositional semantics presented in this section are correct.

Theorem 3.3 (Compositional Semantics of Single Uncorrelated Sublinks in Selections)**.** *The compositional semantics of PI-CS for single uncorrelated sublinks in selection as presented in Figure 3.7 is equivalent to the declarative semantics as defined by definition 3.10.*

Proof.

ANY-sublinks

For a selection $\sigma_C(q)$ with a sublink C_{sub} in condition C we have to show that $\mathscr{DD}$ generated by the compositional semantics is the set of witness lists fulfilling conditions 1 to 4. Let $C(Q)$ be condition C with relation Q substituted for Q_{sub}. We use C^R as a shortcut for $C({Q_{sub}}^R)$. Let $C_{sub}(t)$ be the sublink C_{sub} with $\{t\}$ substituted for Q_{sub}.

Case C_{sub} is *reqtrue*:

Condition 1: We have to show that $\left[\left[\sigma_{C({Q_{sub}}^R)}({Q_1}^R)\right]\right] = \{t^x\}$ holds. From the compositional semantics we can deduce that ${Q_{sub}}^R = {Q_{sub}}^{true}(t)$. Thus, proving $\left[\left[\sigma_{C^R}({Q_1}^R)\right]\right] = \{t^x\}$ is equivalent to proving $\left[\left[C^R\right]\right] = true$, because $t = u$ for $u \in {Q_1}^R$ is given and the semantics of the selection operator requires C to be true for a tuple to be in the result of the selection. From the definition of ${Q_{sub}}^{true}(t)$ and the definition of the *ANY*-sublink (existential quantification) we deduce that $[[C_{sub}(t')]]$ is true for every subset of Q_{sub} that contains at least one tuple $t' \in {Q_{sub}}^{true}(t)$ and, hence, also for ${Q_{sub}}^R$. From C_{sub} is *reqtrue* we can deduce that C^R is fulfilled.

Condition 2: Let w be a witness list from $\mathscr{DD}$. Then w is of the form $< u,v >$ with $u = t$ and $v \in {Q_{sub}}^{true}(t)$. From the definition of ${Q_{sub}}^{true}(t)$ we know that $C(v)$ is fulfilled if evaluated for a regular input tuple t. Thus, $\left[\left[\sigma_{C(v)}(\{t^n\})\right]\right] = \{t^n\} \neq \emptyset$ holds.

Condition 3: Assume a superset $\mathscr{O}$ of $\mathscr{DD}$ exists that fulfills conditions 1,2, and 4. We know that $\mathscr{O}$ has to contain a witness list $w =< u,v >$ with either $u \neq t$ or $v \notin {Q_{sub}}^{true}(t)$. In the first case either condition 1 would break, if $\left[\left[C^R\right]\right]$ is true for u or otherwise condition 2 is not fulfilled, because $\left[\left[\sigma_{C(v)}(\{u\})\right]\right] = \emptyset$. In the second case condition 2 breaks, because from the definition of ${Q_{sub}}^{true}$ we know the $[[C(v)]]$ is false.

Condition 4: All witness lists in $\mathscr{DD}$ are of the form $< t,v >$ with $v \in {Q_{sub}}^{true}(t)$. Therefore, there are no witness lists w and w' that fulfill the precondition $w \prec w'$ and condition 4 is fulfilled.

Case C_{sub} is *reqfalse* or *ind*:

Condition 1: Using the same argument as for the *reqtrue* case we conclude that condition 1 holds.

Condition 2: Let w be a witness list from $\mathscr{DD}$. Then w if of the form $< u,v >$ with $u = t$ and $v \in Q_{sub}$. If $\mathscr{R}(q,C_{sub},t) = reqfalse$ then $[[C_{sub}(v)]]$ is false and $[[C(v)]]$ is true. Therefore, $\left[\left[\sigma_{C(v)}(u)\right]\right] = \{t\} \neq \emptyset$. If $\mathscr{R}(q,C_{sub},t) = ind$ then $[[C(t')]]$ is true for all t' from Q_{sub} and $\left[\left[\sigma_{C(v)}(u)\right]\right] = \{t\} \neq \emptyset$ holds.

Condition 3: Assume a superset $\mathscr{O}$ of $\mathscr{DD}$ exists that fulfills conditions 1,2, and 4. We know that $\mathscr{O}$ has to contain a witness list $w =< u,v >$ with $u \neq t$ (otherwise w would be in $\mathscr{DD}$). If C is true for u then condition 1 breaks. If C is false then condition 2 breaks, because $\left[\left[\sigma_{C(v)}(u)\right]\right] = \emptyset$.

Condition 4: Since $Q_{sub} \supseteq {Q_{sub}}^{true}(t)$ we can apply the same reasoning as for the *reqtrue* case.

ALL-, EXISTS-, and scalar sublinks

We now prove the correctness of the compositional semantics for *ALL*-, *EXISTS*- and scalar sublinks.

Case C_{sub} is *reqtrue* or *ind*: We have to prove that ${Q_{sub}}^R = Q_{sub}$ holds by proving that the witness list $\mathscr{O}(op,t)$ generated by the compositional semantics fulfills conditions 1 to 4 from definition 3.10. The correctness of ${Q_1}^R$ is given by the compositional semantics of selection.

Condition 1: We have to show $\left[\left[\sigma_{C^R}({Q_1}^R)\right]\right] = \{t^x\}$, which is equivalent to showing that C^R evaluates to true as for *ANY*-sublinks. For all three types of sublink expressions we are treating here ${Q_{sub}}^R = Q_{sub}$. Thus, $C^R \Leftrightarrow C$ and C evaluates to true, because otherwise t would not be in the result of the selection.

Condition 2: Let w be a witness list from $\mathscr{DD}$. Then w is of the form $< u,v >$ with $u = t$ and $v \in Q_{sub}$. For EXISTS- and scalar-sublinks $C(v) \Leftrightarrow C$ trivially holds. For ALL-sublinks every tuple t' from Q_{sub} fulfills condition $e(t)\ op\ t'$. It follows that $C(v) \Leftrightarrow C$.

R

a
1
2
...
100

S

b
1
5

U

c
5

Q

c
5

$q = \sigma_{C_1 \vee C_2}(U)$ $\qquad$ $C_1 = (c\ =\ ANY\ R)$ $\qquad$ $C_2 = (c\ >\ ALL\ S)$

Solution 1

$\mathscr{W}(q,t) =< \{(5)\},\{(1),(5)\},\{(5)\} >$

$\mathscr{DD}(q,t) = \{< (5),(1),(5) >,< (5),(5),(5) >\}$

Solution 2

$\mathscr{W}(q,t) =< \{(1),\ldots,(100)\},\{(1)\},\{(5)\} >$

$\mathscr{DD}(q,t) = \{< (1),(1),(5) >,\ldots,< (100),(1),(5) >\}$

Figure 3.9: Ambiguity of *Lineage-CS* and *PI-CS* for Multiple Sublinks

Condition 3: Assume a superset $\mathscr{O}$ of $\mathscr{DD}$ exists that fulfills conditions 1,2, and 4. We know that $\mathscr{O}$ has to contain a witness list $w =< u,v >$ with $u \neq t$. If $C(v)$ evaluates to true for u then condition 1 is broken. Otherwise condition 2 is not fulfilled, because $\big[\big[\sigma_{C(v)}(u)\big]\big] = \emptyset$.

Condition 4: All witness lists in $\mathscr{DD}$ are of the form $< t,v >$ with $v \in Q_{sub}$. Therefore, there are no two witness lists w and w' from $\mathscr{DD}$ that fulfill the precondition $w \prec_n w'$ and condition 4 holds.

Case C_{sub} is *reqfalse*:

Condition 1: For EXISTS- and scalar-sublinks ${Q_{sub}}^R = Q_{sub}$ holds. As demonstrated for the *reqtrue* case it follows that condition 1 holds. For ALL-sublinks we have to show that $C \Leftrightarrow C({Q_{sub}}^{false}(t))$. An ALL-sublink is false if the condition $e\ op\ t'$ it not fulfilled for at least one tuple from Q_{sub}. From the definition of ${Q_{sub}}^{false}(t)$ we know that this set contains only tuples for which this condition is not fulfilled.

Condition 2: For EXISTS- and scalar-sublinks the same argument as for the *reqtrue* case applies. For ALL-sublinks we know from ${Q_{sub}}^R = {Q_{sub}}^{false}(t)$ that each witness list in $\mathscr{DD}(q,t)$ is of form $< t,v >$ with $e(t)\ op\ v$ evaluates to false. From C_{sub} is *reqfalse* follows that $\big[\big[\sigma_{C(v)}(\{t\})\big]\big] = \{t\} \neq \emptyset$.

Condition 3: Since ${Q_{sub}}^R = Q_{sub}$ holds for EXISTS- and scalar-sublink we only have to prove condition 3 for ALL-sublinks. Assume a superset $\mathscr{O}$ of $\mathscr{DD}$ exists that fulfills conditions 1,2, and 4. We know that $\mathscr{O}$ has to contain a witness list $w =< u,v >$ with either $u \neq t$ or $v \notin {Q_{sub}}^{false}$. For the first case if $C(u)$ is true then condition 1 breaks. If $C(u)$ is false then condition 2 breaks, because $\big[\big[\sigma_{C(v)}(u)\big]\big] = \emptyset$. For the second case we can deduce from the definition of ALL-sublinks and ${Q_{sub}}^{false}$ that $[[C(v)]] = true$ and, therefore, condition 2 is not fulfilled.

Condition 4: Proven using the same arguments as for the *reqtrue* case.

□

3.2.3.2 Multiple Sublinks Expressions

In this section we extend the results established for single uncorrelated sublinks to queries with multiple uncorrelated sublinks and show that if a selection condition contains more than one sublink the provenance according to *PI-CS* is not unique anymore. We also demonstrate that the same applies for *Lineage-CS*. Hence, this problem is not specific to our *contribution semantics* definition. To solve this problem, the definition of *PI-CS* is extended to produce meaningful and unique results for selections with multiple sublinks. The extended definition has the additional advantage that tuples are excluded from the provenance of single sublink queries if they do not contribute to the result.

Example 3.11. *As an example to illustrate this problem consider the query and database instance presented in Figure 3.9. For the tuple $t = (5)$ from U the sub-condition C_1 of the selection condition C evaluates to true and the sub-condition C_2 evaluates to false. The problem of Lineage-CS with this example is that there are no unique sets R^* and S^* that fulfill conditions 1 to 3, because a solution that maximizes one set requires that the other set is not maximized. A similar argument applies for PI-CS. If a witness list contains tuple (5) from relation S, then only tuple (5) from relation R can be used in a witness list. Otherwise if tuples (1) to (100) from relation R are included in the witness lists, then tuple (5) from relation S can not be used in any witness list without breaking conditions 1 to 4. This problem arises because both CS definitions only require the provenance to produce the same result tuples as the complete input relations, but not to produce the same results for sublink expressions in the query. This leads to the ambiguity examined in the example. Solution 2 fulfills the conditions of the CS definitions, but the results of evaluating C_1 and C_2 are different from the evaluation results of C_1 and C_2 produced by the original query.*

Intuitively, the provenance of a tuple t according to a sublink query Q_{sub} should include only tuples that produce the same result of the sublink C_{sub} as in the original query, because other tuples would only have contributed if a query from another sublink produced a different result. This behavior can be achieved if the *PI-CS* definition is extended with an additional condition that checks that the provenance causes all sublink expressions to evaluate to the same value as for the complete input relations. A side-effect of this restriction is that the *ind* influence role does not exist anymore, because it allows the provenance to produce a different result for a sublink expression.

To be able to define the extension for *PI-CS* we introduce some notational preliminaries. Recall that $w[i-j]$ denotes a sub sequence of a witness list w. We define a subset of $\mathscr{DD}$ using this notations:

$$\mathscr{DD}(< t_1,\ldots,t_n >) = \{w \mid w \in \mathscr{DD} \wedge w[1-n] =< t_1,\ldots,t_n >\}$$

$\mathscr{DD}(< t_1,\ldots,t_n >)$ is used to extract all witness lists that contain a certain combination of regular input relation tuples. The modified definition for *PI-CS* is presented below.

Definition 3.11 (Sublink-Safe *PI-CS*)**.** *For an algebra operator op with regular inputs $Q_1,\ldots,Q_n$, sublink expressions $C_{sub_1},\ldots,C_{sub_m}$, and a tuple $t \in op(Q_1,\ldots,Q_{sub_m})$ a set $\mathscr{DD}(op,t) \subseteq (Q_1^\varepsilon \times \ldots \times {Q_{sub_m}}^\varepsilon)$ is the set of witness lists of t according to PI-CS if it fulfills the following conditions:*

$$[[op(\mathscr{DD}(op,t))]] = \{t^x\} \tag{1}$$

$$\forall w \in \mathscr{DD}(op,t) : [[op(w)]] \neq \emptyset \tag{2}$$

$$\neg\exists \mathscr{DD}' \subseteq (Q_1^\varepsilon \times \ldots \times {Q_{sub_m}}^\varepsilon) : \mathscr{DD}' \supset \mathscr{DD}(op,t) \wedge \mathscr{DD}' \models (1),(2),(4),(5) \tag{3}$$

$$\forall w,w' \in \mathscr{DD}(op,t) : w \prec_n w' \Rightarrow w \notin \mathscr{DD}(op,t) \tag{4}$$

$$\begin{gathered}\forall \mathscr{O} = \mathscr{DD}(< {t_1}^*,\ldots,{t_n}^* >) : t_i^* \in {Q_i}^R : \forall j \in \{1,\ldots,m\} : \forall w \in \mathscr{O} : \\ \left[\left[C_{sub_j}(Q_{sub_j}, < {t_1}^*,\ldots,{t_n}^* >)\right]\right] = \left[\left[C_{sub_j}(w[n+j], < {t_1}^*,\ldots,{t_n}^* >)\right]\right]\end{gathered} \tag{5}$$

The extended *PI-CS* definition has the effect that the provenance of an operator with multiple sublinks is unique and the provenance for each sublink in a query is the same as for queries with a single uncorrelated sublink (Except that only the *reqtrue* and *reqfalse* influence roles apply). Condition 5 is not required for single sublink queries. However, it should be applied to these queries too, because otherwise the provenance can contain tuples that do not contribute to the result of the sublink query (false positives).

Example 3.12. *Consider the query $\sigma_{a=2 \vee a=ANY\ S}(R)$ over the relations from Figure 3.8. For the result tuple $t = (2,1)$ the sublink expression evaluates to true and has influence role ind. Therefore, the provenance of the sublink query is $S^R = S$, but only the tuple $t' = (2,4)$ from S contributed to the result of the sublink.*

As mentioned above, definition 3.11 generates the same provenance as the original definitions of *PI-CS* if applied to queries with no sublink expression or only a single sublink expression. For queries with more than one sublink expression the provenance of each sublink query is derived using the same compositional semantics as for the single sublink case. This means the provenance of each sublink expression can be evaluated independently of the existence of other sublink expressions in the same query. The following theorem formalizes this proposition:

Theorem 3.4 (Compositional Semantics according to Definition 3.11)**.** *The compositional semantics of PI-CS according to definition 3.11 coincide with the compositional semantics after definition 3.11 for all operators without sublinks and selections with at most one uncorrelated sublink, if influence role ind is replaced by reqtrue and reqfalse depending on the actual result of evaluating the sublink expression. For an operator with more than one sublink expression the provenance of each sublink expression is independent of the existence of the other sublink expressions and can be generated using the compositional semantics defined for single sublink expressions.*

Proof.
First we prove that the sublink-safe definition of *PI-CS* does not alter the provenance of operators without sublink expressions and for selections with at most one sublink expressions. The provenance of operators without sublink expressions is not influenced by the new conditions of definition 3.11, because these condition only applies to sublink expressions. We prove the equivalence of the definitions for selections with one uncorrelated sublink together with the claim that the compositional semantics are the same for selections with an arbitrary number of sublink expressions according to the new definitions and for one sublink according to the original definitions. This claim is proven by showing that the provenance generated by the compositional semantics for single sublinks fulfills the conditions of definition 3.11. Let t be a tuple from the output of a selection $\sigma_C(q_1)$ with a condition C that contains n sublink expressions C_{sub_1}, ..., C_{sub_n}. For a single input tuple from Q_1 the results of the sublink expressions are fixed and each $C_{sub_i}({Q_{sub_i}}^R)$ is required to produce this fixed result. Let ${Q_{sub_i}}'$ be the provenance of a sublink derived using the compositional semantics for single sublinks. Obviously, each of these sets fulfills condition 5 from Definition 3.11. It remains to show that these sets fulfill conditions 1 to 4 from definition 3.11.
Condition 1: We know that $[[C']] = \left[\left[C({Q_{sub_1}}', \dots, {Q_{sub_n}}')\right]\right]$ is fulfilled, because t is in the result of the selection and each $C_{sub_i}({Q_{sub_i}}')$ is required to evaluate to the same result as $C_{sub_i}(Q_{sub_i})$. Therefore, $\left[\left[\sigma_{C'}({Q_1}^R)\right]\right] = \{t^x\}$ holds.
Condition 2: Let $w = < t, t_1, \dots, t_n >$ be a witness list from $\mathscr{DD}$. We have to prove that $[[\sigma_{C'}(\{t\})]] \neq \emptyset$ for $C' = C(t_1, \dots, t_n)$. This is the case if for each sublink $\left[\left[C_{sub_i}(Q_{sub_i})\right]\right] = \left[\left[C_{sub_i}(\{t_i\})\right]\right]$ holds, because then $[[C']] = [[C]]$ would hold. The result of a sublink expression is independent of the results of other sublinks in the expression in C. Therefore, we can deduce from the proof for single sublinks that $C_{sub_i}(t_i)$ evaluates to the same result as $C_{sub_i}(Q_{sub_i})$. We follow that $[[\sigma_{C'}(\{t\})]] = \{t\} \neq \emptyset$ holds.
Condition 3: Assume a set $\mathscr{O} \supset \mathscr{DD}$ fulfills conditions 1,2,4 and 5 from the definition. Then $\mathscr{O}$ contains at least one witness list $w = < u, t_1, \dots, t_n > \notin \mathscr{DD}$. Either $u \neq t$ for which we know from the proofs for single sublinks that it would break one of the condition or w.l.o.g. $t_i \notin {Q_{sub_i}}^R$. It follows that $\left[\left[C_{sub_i}(t_i)\right]\right] \neq \left[\left[C_{sub_i}\right]\right]$ which breaks condition 5.
Condition 4: Condition 4 was explicitly redefined to not include sublink expressions.

□

3.2.3.3 Sublink Expressions in Projections

A sublink expression used in a projection list A appears in some projection expression a from the list. Given some input tuple t of the projection, a evaluates to a fixed value v instead of a boolean value as for a selection condition. Similar to selection the sublink expression evaluates to a fixed result for a given regular input tuple of the projection it is used in. To be able to derive the compositional semantics for projections containing sublinks we have to distinguish two cases:

1. For the duplicate removing version of projection different tuples from the regular input of the operator can produce the same result tuple. The set of tuples from the regular input Q_1 that produce a

R

a
1
2

S

b
1
3

Q_1

c
true

Q_2

c
true
true

$q_1 = \Pi^{S}{}_{C_1 \vee C_2 \to c}(R)$ $\quad q_2 = \Pi^{B}{}_{C_1 \vee C_2 \to c}(R)$ $\quad C_1 = (a\ =\ ANY\ S)$ $\quad C_2 = (a+1\ =\ ANY\ S)$

$$\mathscr{D}\mathscr{D}(q_{1/2},(true)) = \{<(1),(1),(1)>,<(1),(1),(3)>,<(2),(1),(3)>,<(2),(3),(3)>\}$$

Figure 3.10: Example for Sublinks in Projection

tuple t from the output of the projection is the provenance ${Q_1}^R$ of t according to Q_1 . Let w.l.o.g. ${Q_1}^R = \{t_1,\dots,t_n\}$. We know that for every $t_i \in {Q_1}^R$, the projection expression e in which C_{sub} is used in produces the same result, because otherwise ${Q_1}^R$ would not fulfill condition 5 of definition 3.11. For two tuples t_i and t_j from ${Q_1}^R$, $[[C_{sub}]]$ and the sets ${Q_{sub}}^{true}$ and ${Q_{sub}}^{false}$ can be different. For each tuple t_i from ${Q_1}^R$, the results established for duplicate preserving projection can be applied.

2. For the duplicate preserving version of projection the same reasoning applies, because two regular input tuples may be projected on the same attribute values generating a result tuple t with a multiplicity greater than one. This tuple t will contain both regular input tuples in its provenance (see the discussion of *derivation pool* vs. *derivation set* in section 3.2.1).

As we will demonstrate the sublink-safe definition of *PI-CS* extends naturally to projection.

Example 3.13. *Figure 3.10 presents an example for sublinks in projection. Both example queries q_1 and q_2 project their input on a tuple $(true)$. For regular input tuple (1) from relation R sublink C_1 evaluates to true and sublink C_2 evaluates to false. The opposite holds for regular input tuple (3). Therefore, the PI-CS provenance of both q_1 and q_2 contains two witness lists $<(1),(1),(1)>$ and $<(1),(1),(3)>$ for regular input tuple (1) and two witness lists $<(2),(1),(3)>$ and $<(2),(3),(3)>$ for regular input tuple (3).*

For a projection $q = \Pi^{S/B}{}_A(q_1)$ we define an auxiliary set $\bigcup \mathscr{D}\mathscr{D}(q,t)$ that will be used in the derivation of the provenance of projection sublinks. Intuitively, $\bigcup \mathscr{D}\mathscr{D}(q,t)$ is the combination of the *PI-CS* provenance if computed for each single regular input tuple that is projected on t. The formal definition is given below.

$$\bigcup \mathscr{D}\mathscr{D}(q,t) = \bigcup_{u:u \in Q_1 \wedge \Pi_A(\{u\})=\{t\}} \mathscr{D}\mathscr{D}(q,t,u)$$

$$\mathscr{D}\mathscr{D}(q,t,u) = \mathscr{D}\mathscr{D}(\Pi^{S/B}{}_A(\{u\}),t)$$

We now prove that the provenance of a result tuple t of a projection $q = \Pi^{S/B}{}_A(q_1)$ that contains uncorrelated sublinks is the union of the provenance derived for each regular input tuple $t' \in Q_1$ that is projected on t, if the provenance of each t' is derived using the compositional semantics for sublinks in selections.

Theorem 3.5 (Compositional Semantics of Uncorrelated Sublinks in Projection)**.** *Let $q = \Pi^{S/B}{}_A(q_1)$ be a projection that contains sublinks $C_{sub_1},\dots,C_{sub_m}$ in A. The provenance of a result tuple t according to PI-CS is:*

$$\mathscr{D}\mathscr{D}(q,t) = \{w = <u,v_1,\dots,v_m> \mid w \in \bigcup \mathscr{D}\mathscr{D}(q,t)\}$$

Proof.
For *EXISTS*- and scalar-sublinks, ${Q_{sub_i}}^R = Q_{sub_i}$ holds for every witness list set $\mathscr{D}\mathscr{D}(q,t,u)$ with $u \in {Q_1}^R$ and, thus, ${Q_{sub_i}}^R = Q_{sub_i}$ also holds for $\bigcup \mathscr{D}\mathscr{D}$. It follows that conditions 1 to 5 are fulfilled for $\mathscr{D}\mathscr{D} = \bigcup \mathscr{D}\mathscr{D}$.

To prove that $\mathscr{D}\mathscr{D} = \bigcup \mathscr{D}\mathscr{D}$ holds for *ANY* and *ALL* sublinks too, we have to show that the conditions from definition 3.11 are fulfilled. We only present the proof for *ANY*-sublinks. The proof for *ALL*-sublinks is analog. We write $A({Q_{sub_i}}')$ for the projection list A with ${Q_{sub_i}}'$ substituted for Q_{sub_i} and use A^* as a shortcut for $A({Q_{sub_1}}^R, \dots, {Q_{sub_m}}^R)$. Analog $A(u)$ for $u \in Q_1$ is the projection list A with attributes from $\mathbf{Q_1}$ substituted by the attribute values from u. $A^*(u)$ for $u \in Q_1$ is used as a shortcut for $A(u, {Q_{sub_1}}^R, \dots, {Q_{sub_m}}^R)$. Furthermore let ${Q_{sub_i}}^R(u)$ denote ${Q_{sub_i}}^R$ if computed for $\mathscr{D}\mathscr{D}(q,t,u)$ for a tuple $u \in {Q_1}^R$.

Condition 1:

We have to prove that every tuple u from ${Q_1}^R$ is projected on t: $\left[\left[\Pi_{A^*(u)}(\{u\})\right]\right] = \{t\}$. This condition is fulfilled if $[[A^*(u)]] = [[A(u)]]$ which in turn is true if $\left[\left[C_{sub_i}(u, Q_{sub_i})\right]\right] = \left[\left[C_{sub_i}(u, {Q_{sub_i}}^R)\right]\right]$ for every C_{sub_i}. If C_{sub_i} evaluates to true for all $u \in {Q_1}^R$, then $\left[\left[C_{sub_i}(u, Q_{sub_i})\right]\right] = \left[\left[C_{sub_i}(u, {Q_{sub_i}}^R)\right]\right]$ is fulfilled, if $\left[\left[C_{sub_i}(u, {Q_{sub_i}}^R)\right]\right]$ is true for each such u. From the definition of $\bigcup \mathscr{D}\mathscr{D}(q,t)$ follows that ${Q_{sub_i}}^R$ contains each ${Q_{sub_i}}^R(u)$ and, therefore, contains at least one tuple t' that fulfills condition $e(u)\ op\ t'$. In consequence $\left[\left[C_{sub_i}(u, Q_{sub_i})\right]\right] = \left[\left[C_{sub_i}(u, {Q_{sub_i}}^R)\right]\right]$. If C_{sub_i} evaluates to false for at least one $v \in {Q_1}^R$, no tuple from Q_{sub_i} fulfills condition $e(v)\ op\ t'$. In consequence also no tuple from ${Q_{sub_i}}^R = {Q_{sub_i}}^R(v) = Q_{sub_i}$ fulfills this condition. Hence, $\left[\left[C_{sub_i}(v, Q_{sub_i})\right]\right] = \left[\left[C_{sub_i}(v, {Q_{sub_i}}^R)\right]\right]$. Because of ${Q_{sub_i}}^R = Q_{sub_i}$, $\left[\left[C_{sub_i}(u, Q_{sub_i})\right]\right] = \left[\left[C_{sub_i}(u, {Q_{sub_i}}^R)\right]\right]$ also holds for all other tuples u from ${Q_1}^R$. We have proven that $\left[\left[C_{sub_i}(u, Q_{sub_i})\right]\right] = \left[\left[C_{sub_i}(u, {Q_{sub_i}}^R)\right]\right]$ holds for every C_{sub_i} Hence, condition 1 holds.

Condition 2:

From the definition of projection we know that the result of a projection with non empty regular input can never be empty. It follows that condition 2 is fulfilled.

Condition 3:

Assume a set $\mathscr{O} \supset \mathscr{D}\mathscr{D}$ fulfills conditions 1,2,4, and 5. $\mathscr{O}$ has to contain at least one witness list $w =< u, v_1, \dots, v_m >$ with $u \in {Q_1}^R$ (otherwise condition 1 or 5 would not be fulfilled) and at least one v_i in w is not used in any witness list $w' \in \mathscr{D}\mathscr{D}(q,t)$ with $w'[1] = u$. From the definition of $\bigcup \mathscr{D}\mathscr{D}(q,t)$ and the proof for single sublinks in selection we deduce that $\left[\left[C_{sub_i}(u, Q_{sub_i}) \neq C_{sub_i}(u, \{v\})\right]\right]$.

Condition 4:

Condition 4 only applies for the regular input and, since, the provenance of projection never contains $\perp$ the condition is fulfilled.

Condition 5:

Condition 5 is defined for a fixed combination of regular input tuples denoted by tup. In case of projection this is a single tuple u from Q_1. Since by definition $\bigcup \mathscr{D}\mathscr{D}$ contains all witness lists for u that would have been generated by the compositional semantics for selection, condition 5 holds.

□

3.2.3.4 Correlated Sublinks

The difference between correlated sublinks and uncorrelated sublinks is that for correlated sublinks not only C_{sub} depends on the regular input of the operator the sublink is used in, but Q_{sub} depends on the regular input too. If we consider single input operators like selection or projection, Q_{sub} is constant for a fixed input tuple t from the operators regular input Q. For selection, a single output tuple is derived from one tuple of the input. Thus, the provenance of a tuple t is defined as for uncorrelated sublinks.

For projections, more than one regular input tuple can belong to the provenance of an output tuple t. The attribute values of each of these input tuples t' parameterize the sublink query Q_{sub}. Therefore, the results of the sublink query Q_{sub} can differ, depending on which of the input tuples is used to parameterize it.

Example 3.14. *For the query $q = \Pi_{a\,=\,ALL\,(\sigma_{a=c}(S))}(R)$ on the relations from Figure 3.8, the sublink query $Q_{sub} = \sigma_{a=c}(S)$ produces three different results for the three tuples from R:*

$$\begin{aligned}
Q_{sub}(1,1) &= \{(1,3)\} \\
Q_{sub}(2,1) &= \{(2,4)\} \\
Q_{sub}(3,2) &= \emptyset
\end{aligned}$$

The notion of ${Q_{sub}}^{\varepsilon} = Q_{sub} \cup \perp$ does make sense, if there are several versions of Q_{sub}. To cope with this problem we modify the definition of *PI-CS* using a parametrization ${Q_{sub_i}}^{\varepsilon}(tup)$ of ${Q_{sub_i}}^{\varepsilon}$ for a combination $tup = < t_1, \ldots, t_n >$ of regular input tuples and adapting the conditions of the definition to this parameterization. In addition we need to parameterize ${Q_{sub_i}}^{R}$ too and change the evaluation of op over $\mathscr{D}\mathscr{D}(op,t)$ to use this parameterization. Hence, $op(\mathscr{D}\mathscr{D}(op,t))$ is evaluated as the bag union of $[[op(\mathscr{D}\mathscr{D}(< t_1, \ldots, t_n >)]]$ for each combination $t_1, \ldots, t_n$ of regular input tuples that is contained in at least one witness list from $\mathscr{D}\mathscr{D}$.

Definition 3.12 (Correlation-Safe *PI-CS*). *For an algebra operator op with regular inputs $Q_1, \ldots, Q_n$, sublink expressions $C_{sub_1}, \ldots, C_{sub_m}$, and a tuple $t \in op(Q_1, \ldots, Q_{sub_m})$ a set $\mathscr{D}\mathscr{D}(op,t)$ that is a subset of PW as defined below is the set of witness lists of t according to PI-CS if it fulfills the following conditions:*

$$\begin{aligned}
&[[op(\mathscr{D}\mathscr{D}(op,t))]] = \{t^x\} && \textbf{(1)} \\
&\forall w \in \mathscr{D}\mathscr{D}(op,t) : [[op(w)]] \neq \emptyset && \textbf{(2)} \\
&\neg\exists \mathscr{D}\mathscr{D}' \subseteq PW : \mathscr{D}\mathscr{D}' \supset \mathscr{D}\mathscr{D}(op,t) \wedge \mathscr{D}\mathscr{D}' \models (1),(2),(4),(5) && \textbf{(3)} \\
&\forall w, w' \in \mathscr{D}\mathscr{D}(op,t) : w \prec_n w' \Rightarrow w \notin \mathscr{D}\mathscr{D}(op,t) && \textbf{(4)} \\
&\forall \mathscr{O} = \mathscr{D}\mathscr{D}(< {t_1}^*, \ldots, {t_n}^* >) : {t_i}^* \in {Q_i}^* : \forall j \in \{1, \ldots, m\} : \forall w \in \mathscr{O} : \\
&\qquad \left[\left[C_{sub_j}(Q_{sub_j}, < {t_1}^*, \ldots, {t_n}^* >)\right]\right] = \left[\left[C_{sub_j}(w[n+j], < {t_1}^*, \ldots, {t_n}^* >)\right]\right] && \textbf{(5)}
\end{aligned}$$

$$PW = \bigcup_{<t_1,\ldots,t_n> \in Q_1 \times \ldots \times Q_n} (\{t_1\}^{\varepsilon} \times \ldots \times \{t_n\}^{\varepsilon} \times {Q_{sub_1}}^{\varepsilon}(< t_1, \ldots, t_n >) \ldots \times {Q_{sub_m}}^{\varepsilon}(< t_1, \ldots, t_n >))$$

Note that such a trick can only be applied for *PI-CS*, but not for *Linage-CS*. This is due to the fact that in *PI-CS* provenance is represented as combinations of input tuples which makes is possible to allow only certain combinations. Hence, we can restrict the provenance to witness lists that combine a regular input tuple u with a tuples from each $Q_{sub_i}(u)$. Applying the correlation safe definition to the example query q the following set of witness lists is produced:

$$\mathscr{D}\mathscr{D}(q,(true)) = \{< (1,1),(1,3) >, < (2,1),(2,4) >, < (3,2), \perp >\}$$

3.2.3.5 Nested Sublinks

As mentioned before, definition 3.12 defines the provenance of single operator expressions. The provenance of an algebra expression q that is composed of multiple operators is computed by recursively computing the provenance for each operator in q starting at the result of the outermost operator. This form of computation is sound, because provenance is defined to be transitive. Thus, the provenance of nested sublinks can be computed recursively too, by first computing the provenance of the outermost sublink and then using the generated results in the following computations.

Example 3.15. *Figure 3.11 presents the provenance for query q_1 from the newspaper database example presented at the beginning of this chapter.*

person

SSN	name
1-1	Peter Peterson
2-4	Jens Jensen
5-6	Knut Knutsen

newspaper

newsId	name	publisher
1	NZZ	IEEE
2	20 Minuten	Springer

reads

pSSN	nNewsId
1-1	1
1-1	2
2-4	1

$$q_1 = \sigma_{\neg\, EXISTS\, (\sigma_{\neg\, EXISTS\, (\sigma_{pSSN=SSN \wedge newsId=nNewsId}(reads))}(newspaper))}(person)$$

$\mathbf{Q_1}$

SSN	name
1-1	Peter Peterson

$$\mathscr{DD}(q_1, (1-1, Peter\ Peterson)) = \{< (1-1, Peter\ Peterson), \perp, \perp >\}$$

Figure 3.11: Example Provenance Derivation for an Algebra Expression using Nested and Correlated Sublink Expressions

3.2.4 Comparison of the Expressiveness of *Lineage-CS* and *PI-CS*

Having iteratively refined the definitions of *PI-CS* to produce meaningful provenance for a wide set of algebra expressions we now compare the expressiveness of the two contribution semantics. In detail, we answer the question: Does the provenance generated by both contribution semantics contain the same information? Of course this question can only be answered for the operators on which both semantics are meant to produce the same provenance. For instance, the provenance for left outer joins is different for *Lineage-CS* and *PI-CS*.

Intuitively, it is clear that *PI-CS* provenance contains information that is not modeled by *Lineage-CS*, because, for instance, for a set of result tuples of a join the representation used by *Lineage-CS* does not model which tuples were used together by the join. On the other hand, the representation of *PI-CS* does not model the original multiplicity of input tuples. Formally, this intuitions can be proven, by showing that every function that translates between these two representations cannot have an inverse.

Theorem 3.6 (Non Equivalence of *PI-CS* and *Lineage-CS*)**.** *Let function H be a function that maps a witness set to a set of witness lists with the property $H(\mathscr{W}(q,I,t)) = \mathscr{DD}(q,I,t)$. Let H' be a function that maps a set of witness lists to a witness set with $H'(\mathscr{DD}(q,I,t)) = \mathscr{W}(q,I,t)$. It follows that both H and H' are not invertible and, therefore, none of them can exist.*

Proof.
<u>**Case** *H*</u>:

If H has an inverse, then H has to be injective and surjective. Thus, if we find two queries q_1 and q_2, database instances I_1 and I_2, and tuples $t_1 \in Q_1$ and $t_2 \in Q_2$ for which $\mathscr{W}(q_1,I_1,t_1) \neq \mathscr{W}(q_2,I_2,t_2)$ and $\mathscr{DD}(q_1,I_1,t_1) = \mathscr{DD}(q_2,I_2,t_2)$ hold, we have proven our claim by demonstrating that H is not injective (given that either $q_1 \neq q_2$, $I_1 \neq I_2$, or $t_1 \neq t_2$). Consider the following queries, database instances, and result tuples:

$$\begin{aligned} q_1 &= R \rhd\!\!\lhd_{a=b} S & \qquad q_2 &= R \rhd\!\!\lhd_{a=b} S \\ t_1 &= (1,1) & t_2 &= (1,1) \\ I_1 &= \{R = \{(1)\}, S = \{(1)^2\}\} & I_2 &= \{R = \{(1)^2\}, S = \{(1)\}\} \end{aligned}$$

As shown below for this parameter combinations no injective function H can exists that fulfills the condition $H(\mathscr{W}(q,I,t)) = \mathscr{D}\mathscr{D}(q,I,t)$:

$$\begin{array}{rcl}\mathscr{W}(q_1,I_1,t_1) =< \{(1)\},\{(1)^2\} > & \neq & < \{(1)^2\},\{(1)\} > = \mathscr{W}(q_2,I_2,t_2) \\ \mathscr{D}\mathscr{D}(q_1,I_1,t_1) = \{< (1),(1) >,< (1),(1) >\} & = & \{< (1),(1) >,< (1),(1) >\} = \mathscr{D}\mathscr{D}(q_2,I_2,t_2)\end{array}$$

Case H':

We present an example for which $\mathscr{D}\mathscr{D}(q_1,t_1) \neq \mathscr{D}\mathscr{D}(q_2,t_2)$ and $\mathscr{W}(q_1,t_1) = \mathscr{W}(q_2,t_2)$ hold to prove that H' has no inverse:

$$\begin{array}{ll}q_1 = \Pi^S{}_a(R \bowtie_{b=c} S) & q_2 = \Pi^S{}_a(R \bowtie_{b\neq c} S) \\ t_1 = (1) & t_2 = (1) \\ I_1 = \{R = \{(1,1),(1,2)\}, S = \{(1),(2)\}\} & I_2 = \{R = \{(1,1),(1,2)\}, S = \{(1),(2)\}\}\end{array}$$

$$\begin{array}{rcl}\mathscr{W}(q_1,I_1,t_1) =< \{(1,1),(1,2)\},\{(1),(2)\} > & = & < \{(1,1),(1,2)\},\{(1),(2)\} > = \mathscr{W}(q_2,I_2,t_2) \\ \mathscr{D}\mathscr{D}(q_1,I_1,t_1) = \{< (1,1),(1) >,< (1,2),(2) >\} & \neq & \{< (1,1),(2) >,< (1,2),(1) >\} = \mathscr{D}\mathscr{D}(q_2,I_2,t_2)\end{array}$$

□

Despite the fact that *PI-CS* and *Lineage-CS* are not equivalent which we have proven above, these *CS* types are nonetheless related to each other in the sense that they consider the same input relation tuples to belong to the provenance (with the obvious exception of operators such as left outer join for which *PI-CS* was deliberately defined to generate a different provenance than *Lineage-CS*). We prove this claim by presenting a third representation of provenance and functions H and H' that translate from the *PI-CS* and *Lineage-CS* representations into this representation. If we are able to define these functions in a way that $H(\mathscr{W}(q,I,t)) = H'(\mathscr{D}\mathscr{D}(q,I,t))$ then we have shown that regarding the information stored in this new representation both *CS* types are equivalent. The reduced representation we use is the same as the one of *Lineage-CS* except that the $Q_i{}^*$ subsets of the input relations are sets instead of bags. Therefore, the translation from *Lineage-CS* to the new representation is trivial:

$$H(\mathscr{W}(q,I,t)) =< \{t \mid t^x \in Q_1{}^*\},\ldots,\{t \mid t^x \in Q_n{}^*\} >$$

The definition of H' is straightforward too:

$$H'(\mathscr{D}\mathscr{D}(Q,I,t)) =< \{t \mid \exists w \in \mathscr{D}\mathscr{D} \wedge w[1] = t\},\ldots,\{t \mid \exists w \in \mathscr{D}\mathscr{D} \wedge w[n] = t\} >$$

Theorem 3.7 (Equivalence of the Reduced Representation of *Lineage-CS* and *PI-CS*)**.** *The reduced representations of Lineage-CS and PI-CS are equivalent for algebra expressions containing only the following operators: $\Pi, \sigma, \bowtie, \cup, \cap, \alpha$. Thus, for the translation functions H and H' the following holds:*

$$H(\mathscr{W}(q,t)) = H'(\mathscr{D}\mathscr{D}(q,t))$$

Proof. We prove this theorem by induction over the structure of an algebra expression q and for an arbitrary result tuple t of q:

Base Case:

$q = \sigma_C(q_1)$:

$$\begin{aligned}&H(\mathscr{W}(q,t)) \\ =&H(< \{u^n \mid u^n \in Q_1 \wedge u = t\} >) \\ =&< \{t\} > \\ =&H'(\{< u >^n \mid u^n \in Q_1 \wedge u = t\}) \\ =&H'(\mathscr{D}\mathscr{D}(q,t))\end{aligned}$$

$\underline{q = \Pi_A(q_1)}$:

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u.A = t\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u.A = t\} >\\
=&H'(\{< u >^n \mid u^n \in Q_1 \wedge u.A = t\})\\
=&H'(\mathscr{D}\mathscr{D}(q,t))
\end{aligned}$$

$\underline{q = \alpha_{G,agg}(q_1)}$:

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u.G = t.G\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u.G = t.G\} >\\
=&H'(\{< u >^n \mid u^n \in Q_1 \wedge u.G = t.G\})\\
=&H'(\mathscr{D}\mathscr{D}(q,t))
\end{aligned}$$

$\underline{q = q_1 \bowtie_C q_2}$

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{v^m \mid v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1}\}, \{v \mid v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\} >\\
=&H'(\{< u,v >^{n\times m} \mid u^n \in Q_1 \wedge u = t.\mathbf{Q_1} \wedge v^m \in Q_2 \wedge v = t.\mathbf{Q_2}\})\\
=&H'(\mathscr{D}\mathscr{D}(q,t))
\end{aligned}$$

$\underline{q = q_1 \cup q_2}$

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u = t\}, \{v^m \mid v^m \in Q_2 \wedge v = t\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u = t\}, \{v \mid v^m \in Q_2 \wedge v = t >\\
=&H'(\{< u, \perp >^n \mid u^n \in Q_1 \wedge u = t\} \cup \{< \perp, v >^m \mid v^m \in Q_2 \wedge v = t\})\\
=&H'(\mathscr{D}\mathscr{D}(q,t))
\end{aligned}$$

$\underline{q = q_1 \cap q_2}$

$$\begin{aligned}
&H(\mathscr{W}(q,t))\\
=&H(< \{u^n \mid u^n \in Q_1 \wedge u = t\}, \{v^m \mid v^m \in Q_2 \wedge v = t\} >)\\
=&< \{u \mid u^n \in Q_1 \wedge u = t\}, \{v \mid v^m \in Q_2 \wedge v = t\} >\\
=&H'(\{< u,v >^{n\times m} \mid u^n \in Q_1 \wedge u = t \wedge v^m \in Q_2 \wedge v = t\})\\
=&H'(\mathscr{D}\mathscr{D}(q,t))
\end{aligned}$$

Induction step:

Follows from the transitivity of *Lineage-CS* and *PI-CS*. □

CS type	Description
Complete-Direct-Copy-CS (CDC-CS)	Only tuples that have been copied directly as a whole from the input to the output of a query are considered to belong to the provenance.
Partial-Direct-Copy-CS (PDC-CS)	Only tuples from which at least one attribute value has been copied directly from the input to the output belong to the provenance.
Complete-Transitive-Copy-CS (CTC-CS)	*CTC-CS* contains all directly copied tuples. In addition, implied equalities enforced by selection conditions are handled as copy operations.
Partial-Transitive-Copy-CS (PTC-CS)	Like *PDC-CS*, but implied equalities are considered as copy operations.

Figure 3.12: C-CS types

3.3 Extensions of PI-CS

PI-CS is a contribution semantics that is applicape to wide range of algebra expression and produces meaningful results, even for queries with complex sublink expressions. In this section we demonstrate that it is possible to define reasonable *C-CS* types and *transformation* provenance *CS* based on *PI-CS*.

3.3.1 Copy Data Provenance Contribution Semantics

For some use cases provenance generated by *PI-CS* is not suited very well, because for these use cases we are only interested in where result tuple values are derived from and not in all tuples that influenced a result tuple. For this use case *C-CS* provenance provides the needed information by explaining from where in the input values are copied. As mentioned above, we define the *C-CS* types of *Perm* as extensions of *PI-CS*. Therefore, the *C-CS* types also apply tuple level granularity and, thus, *C-CS* describes which tuples have been copied from the input to the output of an operator. Either we could define that only tuples that have been copied as a whole to the output of an operator belong to the provenance (*complete* copy) or also tuples that have been copied only partially are included in the provenance (*partial* copy). Also we can decide if we see equality constraints implied by selection conditions as a form of copy operation (*transitive* copy).

Example 3.16. *As an example of transitive copy consider an equi-join followed by a projection on the attributes of one of the join relations:*

$$q = \Pi_a(R \bowtie_{a=b} S)$$

For such a query no attributes of the other join relation are directly copied to the output, but implicitly the values of the attributes used in the equality condition are copied. In query q only attribute a from relation R is copied to the result, but the result attribute a is forced by the join condition to be equal to attribute b from relation S.

Figure 3.12 presents the four *C-CS* types that result from the decision between *complete* and *partial* copy, and between *direct* and *transitive* copy. Under *Complete-Direct-Copy-CS* only tuples that are copied completely to the output belong to the provenance. *Complete-Transitive-Copy-CS* (*PDC-CS*) also includes tuples that have been copied partially. *Complete-Transitive-Copy-CS* (*CTC-CS*) includes only completely copied tuples, but implied equalities are handled like direct copying. *Partial-Transitive-Copy-CS* (*PTC-CS*) extents *CTC-CS* with partially copied tuples.

R

a
1
3

S

b
1
5

U

c	d
3	2
3	6

V

e	f
3	3
5	4

Q1

a
1
3

Q2

c
3
5

Q3

x
1

Q4

c
3

$q_1 = \sigma_{a>2}(R)$ $q_2 = \Pi^S{}_c(U)$ $q_3 = \Pi_{a \to x}(R \bowtie_{a=b} S)$ $q_4 = \Pi_c(U \bowtie_{c=e} V)$

$$\begin{aligned}
&\mathscr{C}\mathscr{D}(q_1,(1)) = \mathscr{P}\mathscr{D}(q_1,(1)) = \mathscr{C}\mathscr{T}(q_1,(1)) = \mathscr{P}\mathscr{T}(q_1,(1)) = \{<(1)>\}\\
&\mathscr{C}\mathscr{D}(q_2,(3)) = \mathscr{C}\mathscr{T}(q_2,(3)) = \{<\perp>,<\perp>\}\\
&\mathscr{P}\mathscr{D}(q_2,(3)) = \mathscr{P}\mathscr{T}(q_2,(3)) = \{<(3,2)>,<(3,6)>\}\\
&\mathscr{C}\mathscr{D}(q_3,(1)) = \mathscr{P}\mathscr{D}(q_3,(1)) = \{<(1),\perp>\}\\
&\mathscr{C}\mathscr{T}(q_3,(1)) = \mathscr{P}\mathscr{T}(q_3,(1)) = \{<(1),(1)>\}\\
&\mathscr{C}\mathscr{D}(q_4,(3)) = \mathscr{C}\mathscr{T}(q_4,(3)) = \{<\perp,\perp>^2\}\\
&\mathscr{P}\mathscr{D}(q_4,(3)) = \{<(3,2),\perp>,<(3,6),\perp>\}\\
&\mathscr{P}\mathscr{T}(q_4,(3)) = \{<(3,2),(3,3)>,<(3,6),(3,3)>\}
\end{aligned}$$

Figure 3.13: *C-CS* Examples

Example 3.17. *To gain a better understanding of the differences between these C-CS types consider the example presented in Figure 3.13. Like for PI-CS the provenance under the presented C-CS types is modeled as witness lists. We use the following notations for the set of witness lists generated by the C-CS types: $\mathscr{C}\mathscr{D}$ for CDC-CS, $\mathscr{P}\mathscr{D}$ for PDC-CS, $\mathscr{C}\mathscr{T}$ for CTC-CS, and $\mathscr{P}\mathscr{T}$ for PTC-CS. Query q_1 from the example is a selection, which means it copies input tuples completely to the output. Therefore, all C-CS types generate the same provenance for this query. Query q_2 projects out one attribute from relation U. For this query the provenance of all C-CS types that requires complete copying of input tuples is empty. The provenance of C-CS types that require only partial copying contains the single input tuple from which the result tuple (3) is produced. Query q_3 joins relations R and S and projects the result on an attribute from R. For this query the provenance for relation S is non empty only for C-CS types that consider implied equalities. Finally, query q_4 requires a C-CS type with partial copying to include the input tuple from relation U and only for PTC-CS the tuple (3,3) from relation V is included in the provenance.*

It is apparent from the example that the presented *C-CS* types are not independent of each other and we can assume that several subset relationships hold. By subset we mean both subset in the set theoretic sense and subset in the sense that the smaller witness list set contains witness lists that are dominated by witness lists from the larger set. Intuitively, we would expect that *CDC-CS* is a subset of *PDC-CS*, because *PDC-CS* considers partially copied tuples in addition to tuples that were copied as a whole. Using the same argument we argue that *CTC-CS* is a subset of *PTC-CS*. *CTC-CS* should be a superset of *CDC-CS*, because it also handles implied equalities as copy operations. Furthermore, since *PI-CS* considers types of influence that are not copy operations, we expect *PI-CS* to be a superset of all presented *C-CS* types. Figure 3.14 shows these subset relationships.

We define the introduced *C-CS* types as restrictions of *PI-CS*. These restrictions are modeled as so-called *copy maps*. A copy map describes which attributes from the input of an algebra operator have been copied to its output. Formally, a copy map $\mathscr{C}\mathscr{M}$ is a function $\mathscr{C}\mathscr{M}(\mathscr{E},\mathscr{A},Wit,Tup) \to Pow(\mathscr{A})$. Here Tup denotes the set of all possible tuples, $Pow(S)$ denotes the power set of a set S, and Wit denotes the set of all possible witness lists. Recall that $\mathscr{A}$ denotes the set of all possible attribute names and $\mathscr{E}$ denotes the set of all possible algebra expressions. A copy map maps an attribute a from the schema $\mathbf{Q_i}$ of the input q_i of a query q, a result tuple t from Q, and a witness list w from $\mathscr{D}\mathscr{D}(q,t)$ to all output attributes of q to which the values of a are copied to. We define two types of copy maps. One is used for *C-CS*

$$\begin{tikzcd}
CDC-CS \arrow[r, "\subseteq"] \arrow[d, "\subseteq"] & PDC-CS \arrow[d, "\subseteq"] & \\
CTC-CS \arrow[r, "\subseteq"] & PTC-CS \arrow[r, "\subseteq"] & PI-CS
\end{tikzcd}$$

Figure 3.14: Subset Relationships between Contribution Semantics Types

types with direct copying: *CDC-CS* and *PDC-CS*. The other one is used for *C-CS* types with transitive copying: *CTC-CS* and *PTC-CS*. Note that the copy map for *C-CS* types that consider only direct copying is independent on the database instance, because it is derived using solely information about the structure of an algebra expression and the database schema (the only exception from this rule is the use of conditional expressions in projection expressions).

The provenance according to a *C-CS* type will contain the same witness lists as provenance according to *PI-CS*, but tuples from these witness lists are removed if they have not been copied according to the copy map function. Figures 3.15 and 3.16 presents the definitions of the copy maps. These definitions use the following notational conventions: $w[q_1]$ for a witness list w is the projection of w on tuples from base relations accessed by input query q_1, $\perp [q_1]$ is an witness list for q_1 that contains only $\perp$, and $C_{sub} \in C$ denotes that the expression C_{sub} is used in expression C.

For *CDC-CS* and *CTC-CS* we require that tuples are copied as a whole. Thus, for a witness list w from $\mathcal{DD}(q,t)$ each $w[i]$ is replaced with $\perp$ if the copy map for at least one attribute from an input relation Q_i is empty indicating that this attribute is not copied to the result of q. For *PDC-CS* and *PTC-CS* also partially copied tuples belong to the provenance. Hence, for these *CS* types only tuples from relations for which the copy map of all attributes is empty are replaced with $\perp$.

Example 3.18. *As an example reconsider query q_2 from the example presented in Figure 3.13. The PI-CS provenance of result tuple (1) is $\mathcal{DD}(q_2,(1)) = \{<(3,2)>,<(3,6)>\}$. According to the copy map definition given in Figures 3.15 and 3.16 the CDC-CS copy map for q_2, witness list $<3,2>$ and result tuple (1) is:*

$$\mathcal{CM}(q_2,c,<3,2>,(1)) = \{c\}$$
$$\mathcal{CM}(q_2,d,<3,2>,(1)) = \emptyset$$

In the definition presented below we formalize the *Perm C-CS* types:

Definition 3.13 (Copy-CS Types)**.** *A set $\mathcal{CD}/\mathcal{CT}(q,t)$ of witness lists is the provenance of a tuple t from the result of a query q according to CDC-CS/CTC-CS, iff:*

$$\begin{aligned}\forall w \in \mathcal{CD}/\mathcal{CT}(q,T) : \exists w' \in \mathcal{DD}(q,t) :& ((\nexists a \in \mathbf{Q_i} : \mathcal{CM}(q,a,w',t) = \emptyset) \Rightarrow w[i] = w'[i]) \\ & \wedge ((\exists a \in \mathbf{Q_i} : \mathcal{CM}(q,a,w',t) = \emptyset) \Rightarrow w[i] = \perp)\end{aligned} \quad \textbf{(1)}$$

$$\mid \mathcal{CD}/\mathcal{CT}(q,t) \mid = \mid \mathcal{DD}(q,t) \mid \quad \textbf{(2)}$$

A set $\mathcal{PD}/\mathcal{PT}(q,t)$ of witness lists is the provenance of a tuple t from the result of a query q according to PDC-CS/PTC-CS, iff:

$$\begin{aligned}\forall w \in \mathcal{PD}/\mathcal{PT}(q,T) : \exists w' \in \mathcal{DD}(q,t) :& ((\exists a \in \mathbf{Q_i} : \mathcal{CM}(q,a,w',t) \neq \emptyset) \Rightarrow w[i] = w'[i]) \\ & \wedge ((\nexists a \in \mathbf{Q_i} : \mathcal{CM}(q,a,w',t) \neq \emptyset) \Rightarrow w[i] = \perp)\end{aligned} \quad \textbf{(1)}$$

$$\mid \mathcal{PD}/\mathcal{PT}(q,t) \mid = \mid \mathcal{DD}(q,t) \mid \quad \textbf{(2)}$$

Having formally defined *C-CS* we now prove the assumptions about subset relationships between the different *CS* types.

Theorem 3.8 (Subset Relationships of CS types)**.** *The subset relationships presented in Figure 3.14 hold.*

Proof.
Case *PTC-CS* $\subseteq$ *PI-CS*:
The subset relationships between *PTC-CS* and *PI-CS* follows from the definition of the *C-CS* types. Each witness list in $\mathscr{PT}$ is either in $\mathscr{DD}$ or is dominated by an witness list from $\mathscr{DD}$.
Case *PDC-CS* $\subseteq$ *PTC-CS* and *CDC-CS* $\subseteq$ *CTC-CS*:
These *C-CS* types differ only in the definition of the copy map. The transitive copy map version is either equal to the direct copy map version or includes the direct copy map version (union with additional sets). Hence, the transitive *C-CS* types have to be supersets of the direct *C-CS* types.
Case *CDC-CS* $\subseteq$ *PDC-CS* and *CTC-CS* $\subseteq$ *PTC-CS*:
Condition 1 in the definition of the partial copy *CS* types follows from condition 1 for direct copy *CS* types. It follows that each witness list from $\mathscr{PT}$ / $\mathscr{PD}$ is equal to or subsumes a corresponding witness list from $\mathscr{CD}$ / $\mathscr{CT}$.

□

Note that the distinction between the different copy contribution semantics is not just academic. These *C-CS* types generate different provenance and each emphasizes different aspects of how the result was obtained. We now discuss the relationship between the presented *C-CS* types and *Where-CS*. The main difference is that *Where-CS* is defined with attribute value granularity while our *C-CS* types use tuple granularity. *IWhere-CS* bears more similarities with *CTC-CS* and *PTC-CS*, because it also considers implied equalities. Attribute values are considered atomic for *Where-CS*. Therefore, there is no notion of a *partial* copy of an attribute value.

$$
\begin{aligned}
\mathscr{CM}(R,a,w,t) &= \{a\}\\
\mathscr{CM}(\sigma_C(q_1),a,w,t) &= \mathscr{CM}(q_1,a,w,t)\\
\mathscr{CM}(q_1 \bowtie_C q_2,a,w,t) &= \mathscr{CM}(q_1,a,w[q_1],t.\mathbf{Q_1}) \cup \mathscr{CM}(q_2,a,w[q_2],t.\mathbf{Q_2})\\
\mathscr{CM}(q_1 ⟕_C q_2,a,w,t) &= \mathscr{CM}(q_1,a,w[q_1],t.\mathbf{Q_1}) \cup \{x \mid w \models C \wedge x \in \mathscr{CM}(q_2,a,w[q_2],t.\mathbf{Q_2})\}\\
\mathscr{CM}(q_1 ⟖_C q_2,a,w,t) &= \{x \mid w \models C \wedge x \in \mathscr{CM}(q_1,a,w[q_1],t.\mathbf{Q_1})\} \cup \mathscr{CM}(q_2,a,w[q_2],t.\mathbf{Q_2})\\
\mathscr{CM}(q_1 ⟗_C q_2,a,w,t) &= \{x \mid w \models C \wedge x \in \mathscr{CM}(q_1,a,w[q_1],t.\mathbf{Q_1})\}\\
&\quad \cup \{x \mid w \models C \wedge x \in \mathscr{CM}(q_2,a,w[q_2],t.\mathbf{Q_2})\}\\
&\quad \cup \{x \mid w \not\models C \wedge t.\mathbf{Q_1} \text{ is } \varepsilon \wedge x \in \mathscr{CM}(q_2,a,w[q_2],t.\mathbf{Q_2})\}\\
&\quad \cup \{x \mid w \not\models C \wedge t.\mathbf{Q_2} \text{ is } \varepsilon \wedge x \in \mathscr{CM}(q_1,a,w[q_1],t.\mathbf{Q_1})\}\\
\mathscr{CM}(\alpha_{G,agg}(q_1),a,w,t) &= \{y \mid y \in \mathscr{CM}(q_1,a,w,x) \mid x.G = t.G \wedge a \in G\}\\
\mathscr{CM}(\Pi_A(q_1),a,w,t) &= \{x \mid (x \in \mathscr{CM}(q_1,a,w,y) \wedge x \in A \wedge y.A = t)\}\\
&\quad \cup \{x \mid (b \rightarrow x) \in A \wedge b \in \mathscr{CM}(q_1,a,w,y) \wedge y.A = t)\}\\
&\quad \cup \{x \mid \mathit{if}\ (C)\ \mathit{then}\ (x)\ \mathit{else}\ (e) \in A \wedge x \in \mathscr{CM}(q_1,a,w,y) \wedge y.A = t \wedge w \models C)\}\\
&\quad \cup \{x \mid \mathit{if}\ (C)\ \mathit{then}\ (e)\ \mathit{else}\ (x) \in A \wedge x \in \mathscr{CM}(q_1,a,w,y) \wedge y.A = t \wedge w \not\models C)\}\\
\mathscr{CM}(q_1 \cup q_2,a,w,t) &= \{x \mid x \in \mathscr{CM}(q_1,a,w[q_1],t) \wedge w[q_1] \neq \perp [q_1]\}\\
&\quad \cup \{x \mid x \in \mathscr{CM}(q_2,a,w[q_2],t) \wedge w[q_2] \neq \perp [q_2]\}\\
\mathscr{CM}(q_1 \cap q_2,a,w,t) &= \mathscr{CM}(q_1,a,w[q_1],t) \cup \mathscr{CM}(q_2,a,w[q_2],t)\\
\mathscr{CM}(q_1 - q_2,a,w,t) &= \mathscr{CM}(q_1,a,w[q_1],t)
\end{aligned}
$$

Figure 3.15: Direct C-CS Copy-Map Definition

$$
\begin{aligned}
\mathscr{C}\mathscr{M}(R,a,w,t) &= \{a\}\\
\mathscr{C}\mathscr{M}(\sigma_C(q_1),a,w,t) &= \mathscr{C}\mathscr{M}(q_1,a,w,t) \cup \{x \mid \exists y : (x=y) \in C \wedge w \models (x=y) \wedge y \in \mathscr{C}\mathscr{M}(q_1,a,w,t)\}\\
\mathscr{C}\mathscr{M}(q_1 \bowtie_C q_2,a,w,t) &= \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1}) \cup \mathscr{C}\mathscr{M}(q_2,a,w[q_1],t.\mathbf{Q_2})\\
&\quad \cup \{x \mid \exists y : (x=y) \in C \wedge w \models (x=y)\\
&\quad \wedge (y \in \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1}) \vee y \in \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t.\mathbf{Q_2}))\}\\
\mathscr{C}\mathscr{M}(q_1 \sqsupset\!\!\bowtie_C q_2,a,w,t) &= \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1}) \cup \{x \mid w \models C \wedge x \in \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t.\mathbf{Q_2})\}\\
&\quad \cup \{x \mid \exists y : (x=y) \in C \wedge w \models (x=y) \wedge y \in \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1})\}\\
&\quad \cup \{x \mid \exists y : (x=y) \in C \wedge w \models C \wedge w \models (x=y) \wedge y \in \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t.\mathbf{Q_2}))\}\\
\mathscr{C}\mathscr{M}(q_1 \bowtie\!\!\sqsubset_C q_2,a,w,t) &= \{x \mid w \models C \wedge x \in \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1})\} \cup \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t.\mathbf{Q_2})\\
&\quad \cup \{x \mid \exists y : (x=y) \in C \wedge w \models C \wedge w \models (x=y) \wedge y \in \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1}))\}\\
&\quad \cup \{x \mid \exists y : (x=y) \in C \wedge w \models (x=y) \wedge y \in \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t.\mathbf{Q_2})\}\\
\mathscr{C}\mathscr{M}(q_1 \sqsupset\!\!\bowtie\!\!\sqsubset_C q_2,a,w,t) &= \{x \mid w \models C \wedge x \in \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1})\}\\
&\quad \cup \{x \mid w \models C \wedge x \in \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t.\mathbf{Q_2})\}\\
&\quad \cup \{x \mid w \not\models C \wedge t.\mathbf{Q_1} \text{ is } \varepsilon \wedge x \in \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t.\mathbf{Q_2})\}\\
&\quad \cup \{x \mid w \not\models C \wedge t.\mathbf{Q_2} \text{ is } \varepsilon \wedge x \in \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1})\}\\
&\quad \cup \{x \mid \exists y : (x=y) \in C \wedge w \models C \wedge w \models (x=y) \wedge y \in \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t.\mathbf{Q_1}))\}\\
&\quad \cup \{x \mid \exists y : (x=y) \in C \wedge w \models C \wedge w \models (x=y) \wedge y \in \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t.\mathbf{Q_2}))\}\\
\mathscr{C}\mathscr{M}(\alpha_{G,agg}(q_1),a,w,t) &= \{y \mid y \in \mathscr{C}\mathscr{M}(q_1,a,w,x) \mid x.G = t.G \wedge a \in G\}\\
\mathscr{C}\mathscr{M}(\Pi_A(q_1),a,w,t) &= \{x \mid (x \in \mathscr{C}\mathscr{M}(q_1,a,w,y) \wedge x \in A \wedge y.A = t)\}\\
&\quad \cup \{x \mid (b \rightarrow x) \in A \wedge b \in \mathscr{C}\mathscr{M}(q_1,a,w,y) \wedge y.A = t)\}\\
&\quad \cup \{x \mid \ if\ (C)\ then\ (x)\ else\ (e) \in A \wedge x \in \mathscr{C}\mathscr{M}(q_1,a,w,y) \wedge y.A = t \wedge w \models C)\}\\
&\quad \cup \{x \mid \ if\ (C)\ then\ (e)\ else\ (x) \in A \wedge x \in \mathscr{C}\mathscr{M}(q_1,a,w,y) \wedge y.A = t \wedge w \not\models C)\}\\
\mathscr{C}\mathscr{M}(q_1 \cup q_2,a,w,t) &= \{x \mid x \in \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t) \wedge w[q_1] \neq\perp [q_1]\}\\
&\quad \cup \{x \mid x \in \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t) \wedge w[q_2] \neq\perp [q_2]\}\\
\mathscr{C}\mathscr{M}(q_1 \cap q_2,a,w,t) &= \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t) \cup \mathscr{C}\mathscr{M}(q_2,a,w[q_2],t)\\
\mathscr{C}\mathscr{M}(q_1 - q_2,a,w,t) &= \mathscr{C}\mathscr{M}(q_1,a,w[q_1],t)
\end{aligned}
$$

Figure 3.16: Transitive C-CS Copy-Map Definition

3.3.2 Transformation Provenance Contribution Semantics

In this section we present a contribution semantics for *transformation* provenance based on *PI-CS*. *Data* provenance relates output and input data, but does not provide any information about how data was processed by a transformation. More specifically, it does not contain information about which parts of a transformation were used to derive an output tuple. As an example, consider a transformation that uses the duplicate preserving *union* operator. Each output tuple of the union is produced from exactly one of the relations that are the inputs of the union. Recall that the notion *transformation* provenance is used to describe this type of provenance information.

Transformation provenance is similar in motivation to *how*-provenance[GKT07a] that models some transformation information by recording alternative and conjunctive use of tuples by a query. Unlike *how*-provenance, *transformation* provenance is *operator-centric*, describing the contribution of each operator in a transformation. This is also in contrast to other *data CS* types, which are in general *data-centric*. Our approach is more similar to provenance approaches for workflow-management systems, that traditionally have focused more on transformations [SPG05a]. *Transformation* provenance is extremely useful in understanding how data is processed by an algebra expression, because it allows us to understand which parts of the expression (that is, which operators) produced a result data item. Hence, we model *transformation* provenance as what parts of a query contributed to an output tuple. This approach bears some similarities with *Why-not* provenance presented in [CJ09], so we will discuss the relation to this model and the superior evaluation strategy developed for our model while introducing transformation provenance.

We model the transformation provenance of a query q using an annotated algebra tree for q. For an output tuple t and a witness list w in the *PI-CS* data provenance of t, the transformation provenance will include 1 and 0 annotations on the operators of the transformation q. A 1 indicates this operator on w influences t, a 0 indicates it does not.

Example 3.19. *Consider query q_a in Fig. 3.17. The data provenance of output tuple (2) according to PI-CS contains two witness lists. The transformation provenance of (2) for the first witness list is a tree with every node annotated by a 1 (the left tree presented in Figure 3.17). The transformation provenance of (2) with the second witness is a tree with every node annotated by a 1, except the node for the base relation S which does not contribute and hence would have an annotation of 0 (the right tree presented in Figure 3.17).*

We now formalize annotations for algebra trees and then define *transformation* provenance based on the notion of *data* provenance according to *PI-CS*.

Definition 3.14 (Algebra Tree). *An algebra tree $Tree_q = (V,E)$ for a query q is a tree that contains a node for each algebra operator used in q (including the base relation accesses as leaves). In such tree there is a parent-child relationship between two nodes n_1 and n_2, iff the algebra operator represented by n_2 is an input of the algebra operator represented by n_1. We define a pre-order on the nodes to give each node an identifier (and to order the children of binary operators)[a].*

[a] This is necessary, because for non commutative operators like left outer join the order of inputs matters.

Given an algebra expression q and an operator op used in this expression, we denote the subtree under op by sub_{op}. We use $sub_{op}(w)$ to denote the evaluation of sub_{op} over the witness list w. Based on the concept of the algebra tree of an algebra expression we define annotated algebra trees and will use them to represent transformation provenance information.

Definition 3.15 (Annotated Algebra Tree). *An $\mathscr{A}$-annotated algebra tree for a transformation q is a pair $(Tree_q, \theta)$ where $\theta : V \in Tree_q \rightarrow Pow(\mathscr{A})$ is a function that associates each operator in the tree with a set of annotations from a domain $\mathscr{A}$.*

For transformation provenance, the annotations sets will be singleton sets from the domain $\mathscr{A} = \{0,1\}$ and we assign these annotations specific semantics. However, we include the more general definition as

R

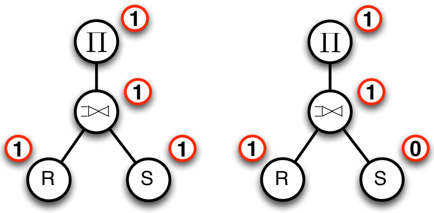
a	b
1	2
1	3
2	3
2	5

S

c
2
3

Q$_a$

a
1
2



$$q_a = \Pi_a(R ⟕_{b=c} S)$$

$$\mathscr{DD}(q_a,(1)) = \{<(1,2),(2)>,<(1,3),(3)>\} \qquad \mathscr{DD}(q_a,(2)) = \{<(2,3),(3)>,<(2,5),\perp>\}$$

$\mathscr{T}(q_c,(1))$:

$$\theta_{<(1,2),(2)>}(op) = 1 \qquad \theta_{<(1,3),(3)>}(op) = 1$$

$\mathscr{T}(q_c,(2))$:

$$\theta_{<(2,3),(3)>}(op) = 1 \qquad \theta_{<(2,5),\perp>}(op) = \begin{cases} 0 \text{ if } op = \mathbf{S} \\ 1 \text{ else} \end{cases}$$

Figure 3.17: Transformation Provenance Example

annotations could be used in a more general way to represent other provenance information (the developer who last checked-in a query, the origin of a transformation).

This representation is conceptually similar to the one use in [CJ09] to model *Why-not* provenance. Recall that *Why-not* provenance models why a certain input (represented as a pattern) does not contribute to some result. This information is presented as so-called *picky* parts of a query, which means the parts of a query where the input of interest ”got lost”. In our representation the picky operations would be labeled with 0 annotations. In contrast to their work we do not require a user to come up with a certain input that got lost, but define the annotations based solely on *data* provenance. As we will demonstrate in chapter 5 transformation provenance can be computed efficiently without the need to compute *data* provenance which is the way *Why-not* provenance is computed in [CJ09].

We now have the necessary preliminaries to formally define transformation provenance based on *data* provenance. Intuitively, each witness list of the *data* provenance of a tuple t represents one evaluation of an algebra expression q. For each witness list, each part of the algebra expression has either contributed to the result of evaluating q on w or not. Therefore, we represent the transformation provenance as a set of annotated algebra trees of q with one member per witness list w. We use *data* provenance to decide whether an operator op in q should get a 0 or a 1 annotation. Basically, if evaluating the subtree sub_{op} under op on w results in the empty set, then op has contributed nothing to the result t and should not be included in the transformation provenance.

Definition 3.16 (Transformation Provenance Contribution Semantics)**.** *The transformation provenance of an output tuple t of q is a set $\mathscr{T}(q,t)$ of $\{0,1\}$-annotated-trees defined as follows:*

$$\mathscr{T}(q,t) = \{(Tree_q, \theta_w) \mid w \in \mathscr{DD}(q,t)\}$$

$$\theta_w(op) = \begin{cases} 0 \text{ } if \text{ } [[sub_{op}(w)]] = \emptyset \\ 1 \text{ } else \end{cases}$$

Example 3.20. *Fig. 3.17 shows the PI-CS data and transformation provenance for both result tuples of query* q_a*. The PI-CS provenance of tuple* $t_1 = (1)$ *contains two witness lists* $w_1 =< (1,2),(2) >$ *and* $w_2 =< (1,3),(3) >$*. For both witness lists the transformation provenance annotation function* θ_w *annotates each operator of* q *with* 1*. We can verify that this is correct by computing* $[[sub_{op}(w_1)]]$ *and* $[[sub_{op}(w_2)]]$ *for each operator in the query. For the second result tuple* $t_2 = (2)$ *there are two witness lists* $w_3 =< (2,3),(3) >$ *and* $w_4 =< (2,5),\perp>$*. The annotation function* θ_{w_4} *for witness list* w_4 *annotates S with 0 and every other operator in q with 1. S is not contained in the transformation provenance, because no tuple from S was joined with the tuple* $(2,5)$ *from R to produce tuple* t_2 *and, therefore, the access to relation S does not contribute to* t_2 *according to* w_4*.*

$$\begin{tikzcd}
\mathbf{Lineage} - \mathbf{CS}(\mathbf{Def}.\mathbf{3}.\mathbf{1}) \arrow[r, "\text{Representation}"] & \mathbf{WL} - \mathbf{CS}(\mathbf{Def}.\mathbf{3}.\mathbf{5}) \arrow[r, "\text{Negation}"] & \mathbf{PI} - \mathbf{CS}(\mathbf{Def}.\mathbf{3}.\mathbf{7}) \arrow[d, "\text{ALL-sublinks}"'] \\
 & & \mathbf{PI} - \mathbf{CS}(\mathbf{Def}.\mathbf{3}.\mathbf{10}) \arrow[d, "\text{Multiple Sublinks}"'] \\
 & & \mathbf{PI} - \mathbf{CS}(\mathbf{Def}.\mathbf{3}.\mathbf{11}) \arrow[d, "\text{Correlated Sublinks}"'] \\
 & & \mathbf{PI} - \mathbf{CS}(\mathbf{Def}.\mathbf{3}.\mathbf{12})
\end{tikzcd}$$

Figure 3.18: Overview of Contribution Semantics Refinement

3.4 Summary

In this chapter we presented the *contribution semantics* realized in the *Perm* system. *CS* types are of immense importance for a provenance management system, because they define ”What provenance actually is”. We discussed how our definitions of *CS* relate to existing *CS* types and to each other, and how we extended the existing notion of *Lineage-CS* to overcome problems of the original definition and make it applicable to algebra expressions with sublinks. Figure 3.18 summarizes the refinement steps we applied to derive the final *PI-CS* definition from the original *Lineage-CS* definition [CW00a]. We have shown that the presented *C-CS* types are all subsets of *PI-CS* (which would be intuitively expected to hold). Furthermore, we introduced a contribution semantics for transformation provenance based on the *PI-CS data* provenance contribution semantics. In summary, in this chapter we presented sound formal definitions of the semantics of provenance, but did not discuss how to compute provenance according to a certain *CS* type. Note that the compositional semantics can be used to compute provenance by recursively applying the constructions from this semantics for each operator in a query, but, as we will demonstrate in the next chapter, there are more practical and efficient approaches for computing provenance.

Chapter 4

Provenance Computation through Algebraic Rewrite

In the last chapter we introduced the *CS* types developed for *Perm* and have proven several import properties of their compositional semantics and relationships to standard *CS* types. However, we did not discuss how provenance can be computed efficiently according to these *CS* types. In this chapter we present algorithms that allow the efficient computation of provenance by using algebraic rewrites. In detail, we demonstrate how to represent provenance information as normal relations and introduce rewrite rules that transform an algebra expression q into an algebra expression q^+ that computes the provenance of q in addition to its original result.

Out of the possible approaches to provenance computation we choose algebraic rewrite, because this approach has several important advantages over alternative approaches like, e.g., the inverse approach:

- **No Modification of Data Model**: Provenance information is modeled as normal relations which can be, e.g., stored in a standard DBMS, queried using SQL, and stored as a view.
- **No Modification of Execution Model**: The rewritten query q^+ that computes the provenance of a query q is expressed in the same algebra as q (To be more precise, the *Perm* algebra introduced in section 3.1). This will allow for the seamless integration of the rewrite rules into an existing DBMS and enables us to benefit from the advanced query optimizations applied by this system. Furthermore, provenance information can be queried using the same query language as for normal data.
- **Sound Theoretical Foundation**: The algebraic representation of provenance computation enables us to prove the correctness of the developed algorithms.

In the following we discuss how to represent provenance in the relational model in section 4.1 and demonstrate how we can transform a query q into a query q^+ that generates this representation for *PI-CS* provenance (section 4.2) and *C-CS* provenance (section 4.4). In section 4.5 we present a relational representation of *transformation* provenance and rewrite rules for this type of provenance are developed in section 4.6.

shop

name	numEmpl
Merdies	3
Joba	14

sales

sName	itemId
Merdies	1
Merdies	2
Merdies	2
Joba	3
Joba	3

items

id	price
1	100
2	10
3	25

Q

name	sum(price)
Merdies	120
Joba	50

$$q_{ex} = \alpha_{name,sum(price)}(\sigma_{name=sName \wedge itemId=id}(shop \times sales \times items))$$

$$\begin{aligned}\mathscr{DD}(q_{ex},(Merdies,120)) &= \{ <(Merdies,3),(Merdies,1),(1,100)>, \\ &\qquad <(Merdies,3),(Merdies,2),(2,50)>^{2}\} \\ \mathscr{DD}(q_{ex},(Joba,50)) &= \{ <(Joba,14),(Joba,3),(3,25)>^{2}\}\end{aligned}$$

Figure 4.1: Running Example

4.1 Relational Representation of Data Provenance Information

The witness lists used by the *Perm CS* types to represent *data* provenance have a natural representation in the relational model. For instance, the set $\mathscr{DD}$ of witness lists generated by *PI-CS* can be represented as a single relation, because each witness list contains tuples with the same schema (We postpone the discussion of the representation of $\perp$ for now).

Example 4.1. *As an example of this representation consider the provenance of query q_{ex} presented in Figure 4.1 that computes the total profit for each shop over an example database of shops (with name and number of employees), items they are selling, and purchases (sales relation). The witness lists from $\mathscr{DD}(q_{ex},(Merdies,120))$ can be represented as the following relation:*

$$\{(Merdies,3,Merdies,1,1,100),(Merdies,3,Merdies,2,2,10)^{2}\}$$

If this representation is used to represent complete sets of witness lists according to some *CS* type the problem arises that it is no longer clear to which output tuple of a query a witness lists belongs to. Recall that we identified this association between original query results and provenance as one of the main requirements of a *PMS*. Therefore, we include the original query results in our representation of *data* provenance. The extended representation models each witness list w and the original result tuple it is associated to as a single tuple.

Example 4.2. *The PI-CS provenance of query q_{ex} would be represented as (The part of a tuple marked in red corresponds to the original result tuple) :*

$$\begin{aligned}\{&(Merdies,120,Merdies,3,Merdies,1,1,100),\\ &(Merdies,120,Merdies,3,Merdies,2,2,10)^{2},\\ &(Joba,50,Joba,14,Joba,3,3,25)^{2},\}\end{aligned}$$

Let us now consider the provenance representation for an arbitrary algebra statement q. We use Q^{PI} to denote the relational representation of the provenance of a query q according to *PI-CS*. To produce the provenance relation Q^{PI}, the original result relation Q is extended with all attributes from all base

$\mathbf{Q_{ex}}^{\mathbf{PI}}$

name	sum(price)	$\mathscr{N}$(name)	$\mathscr{N}$(numEmpl)	$\mathscr{N}$(sName)	$\mathscr{N}$(itemId)	$\mathscr{N}$(id)	$\mathscr{N}$(price)
Merdies	120	Merdies	3	Merdies	1	1	100
Merdies	120	Merdies	3	Merdies	2	2	100
Merdies	120	Merdies	3	Merdies	2	2	100
Joba	50	Joba	14	Joba	3	3	25
Joba	50	Joba	14	Joba	3	3	25

Figure 4.2: Example Provenance Representation

relations accessed by q. Multiple references to a base relation are handled as separate relations. For each original result tuple t and witness list $w \in \mathscr{DD}(q,t)$ a tuple $(t, w[1], \ldots, w[n])$ is added to Q^{PI}. Hence, the original tuple has to be duplicated, if there is more than one witness list in the provenance of this tuple. The attribute names in the schema of Q^{PI} are used to indicated from which base relation attribute a result attribute is derived from. The attributes that correspond to the original result attributes of q are not renamed. To generate unique and predictable names for attributes storing provenance information we introduce a function $\mathscr{N}_q : \mathbb{N} \rightarrow \mathscr{A}$ that maps each attribute position in $\mathbf{Q^{PI}}$ to a unique name. The definition of $\mathscr{N}_q$ will be presented in chapter 5. Here we just assume that $\mathscr{N}$ exists and present an attribute name that is generated by $\mathscr{N}_q$ from a base relation attribute a as $\mathscr{N}(a)$[1]. For example, query q_{ex} accesses base relations $shop$, $sales$ and $items$. In consequence, the schema of $Q_{ex}{}^{PI}$ according to the simplified representation is:

$$\mathbf{Q_{ex}}^{\mathbf{PI}} = (name, sum(price), \mathscr{N}(name), \mathscr{N}(numEmpl), \mathscr{N}(sName), \mathscr{N}(itemId), \mathscr{N}(id), \mathscr{N}(price))$$

To be able to represent $\perp$ values in a witness list using the same schema as for regular tuple values we represent a $\perp$ at position i of witness list w as a tuple with schema $\mathscr{N}_q(\mathbf{R_i})$ and all attributes set to ε. For instance, a witness list $w =< (1), \perp >$ for tuple $t = (3,2)$ from the provenance of a query over relations R and S with schemas $\mathbf{R} = (a)$ and $\mathbf{S} = (b,c)$ would be represented as:

$$(3,4,1,\varepsilon,\varepsilon)$$

Below we present a formal definition of the relational representation of *data* provenance. This definition will be used in the correctness proofs of the query rewrite rules that generate queries to compute this representation.

Definition 4.1 (Relational PI-CS Data Provenance Representation). *Let q be an algebra expression over base relations $R_1, \ldots, R_n$. The relational representation Q^{PI} of the provenance of q according to PI-CS is defined as:*

$$Q^{PI} = \{(t, w[1]', \ldots, w[n]')^m \mid t^p \in Q \wedge w^m \in \mathscr{DD}(q,t)\}$$

$$w[i]' = \begin{cases} w[i] & if\, w[i] \neq \perp \\ null(\mathbf{R_i}) & else \end{cases}$$

Example 4.3. *Figure 4.1 shows the provenance representation for query q_{ex} according to definition 4.1. For instance, the first tuple in $Q_{ex}{}^{PI}$ represents the original result tuple $t = (Merdies, 1)$ and the relational representation of witness list $w =< (Merdies, 3), (Merdies, 1), (1, 100) >$ from the provenance of t.*

This representation is quite verbose, but has several advantages over alternative representations:

[1] We ignore the fact that this representation is not necessarily unique. Note that the function $\mathscr{N}_q$ presented in chapter 5 generates unique names.

- **Single Relation**: Provenance and original data are represented together in a single relation which can, e.g., be stored as a view.
- **Association between Provenance and Normal Data**: Which witness list belongs to which result tuple is explicitly stored in the representation, because each tuple in Q^+ contains one original result tuple and one of its witness lists.
- **Provenance represented as Complete Tuples**: We deliberately choose to represent provenance as complete tuples instead of tuple identifiers to simplify the interpretation and querying of provenance.

We demonstrate the disadvantages of non-relational provenance representations on hand of the running example from Figure 4.1 and for the representation used for *Lineage-CS* (e.g., [Cui02]). *Lineage-CS* provenance would represent the provenance of q_{ex} as the following list of relations[2]:

$$\begin{aligned}
<\{&(Merdies, 3), (Joba, 14)\},\\
&\{(Merdies, 1), (Merdies, 2), (Merdies, 2), (Joba, 3), (Joba, 3)\},\\
&\{(1, 100), (2, 10), (3, 25)\} >
\end{aligned}$$

This representation has two major disadvantages. First, a query having a list of relations as its result can not be expressed in relational algebra, because each algebra operator has only a single result relation. Thus, provenance queries and data are not in the same data model as the original data and queries. Second, the result only includes provenance data. There is no direct association between the original result and the contributing tuples. This is especially problematic if the provenance of a set of tuples is computed, because one would loose the information about which of the provenance tuples contributed to which of the original result tuples. The example above presents an extreme case of this problem where the provenance is the complete original database instance. As already demonstrated, these shortcomings are avoided by the provenance representation used by *Perm*.

[2] The actual representation would be different because we are using bag semantics here.

4.2 Rewrite Rules for Perm-Influence Contribution Semantics

Having presented the provenance representation for which rewrites should be produced, we now present how a query q is transformed by the *Perm* approach into a query q^+ that generates the desired provenance result schema and propagates provenance according to *PI-CS*. In this section we limit the discussion to algebra expressions without sublinks. Algebra expressions with sublinks will be handled in section 4.3. The algebraic rewrites we use to propagate provenance from the base relations to the result of a query is defined for single algebra operators. Recall, that we have discussed in chapter 2 that the principle of propagation of information attached to data is used by annotation systems like *DBNotes* [CTV05] or Mondrian [GKM05]. The approach used by Cui in [CW00a] for *Lineage-CS*, the *CS* from which *PI-CS* is derived from, is based on inversion of queries that trace the origin of a tuple (or set of tuples) from the result back to the source. A disadvantage of this approach is that it requires the instantiation of intermediate results for some operators like aggregation that are not invertible. The *Perm* approach omits the instantiation of intermediate results. The provenance computation for each operator in a query depends exclusively on the result relation of its rewritten inputs and is independent of the computation that generated this rewritten inputs. Thus, we do not have to keep earlier results to compute the current step. We will demonstrate in chapter 5 that this approach has additional advantages if the provenance of only a part of a query should be computed.

The *Perm* algebraic rewrites are modeled as a function $+ : (\mathscr{E}, <\mathscr{A}>) \rightarrow (\mathscr{E}, <\mathscr{A}>)$ that transforms a query q into a provenance computing query q^+. Recall that $\mathscr{E}$ denotes the set of all possible algebra expressions and $<\mathscr{A}>$ denotes the set of all possible lists of attribute names. The list of attribute names (called provenance attribute list $\mathscr{P}$ of a query) is needed to define $+$ as rewrite rules for each algebra operator that are independent of each other. $\mathscr{P}$ is used to store the list of attributes of $\mathbf{Q}^+$ that are used to store the witness lists. The result of $+$ for a query q that contains multiple algebra operators is computed by recursively applying the rewrite rules for to each operator in the query. To be able to rewrite a query incrementally, the rewrite rules have to be applicable to rewritten inputs, i.e., a rewrite has to distinguish between normal and provenance attributes in its input (This is why $\mathscr{P}$ is needed).

Each rewrite rule is modeled as a structural modification (the $\mathscr{E} \rightarrow \mathscr{E}$ part of $+$) and a modification of the provenance attribute list (the $<\mathscr{A}> \rightarrow <\mathscr{A}>$ part of $+$). For two provenance attribute lists $\mathscr{P}_1 = < a_1, \dots, a_n >$ and $\mathscr{P}_2 = < b_1, \dots, b_n >$, the list concatenation operation $\blacktriangleright$ is defined as $\mathscr{P}_1 \blacktriangleright \mathscr{P}_2 = < a_1, \dots, a_n, b_1, \dots, b_n >$.

Definition 4.2 (Provenance Rewrite Meta-Operator)**.** *The provenance rewrite meta-operator* $+ : (\mathscr{E}, <\mathscr{A}>) \rightarrow (\mathscr{E}, <\mathscr{A}>)$ *maps a pair* $(q, <>)$ *to a pair* $(q^+, \mathscr{P}(q^+))$*.* $+$ *is defined over the structure of* q *as rewrite rules for each algebra operator which are shown in Figure 4.3.*

We call $+$ a meta-operator, because, in contrast to algebra operators that transform relational data, it transforms algebra expressions.

4.2.1 Unary Operators Rewrite Rules

In Figure 4.3 the structural and provenance attribute list rewrites for each algebra operator are presented separately. We now discuss each rewrite rule in detail. The rewrite rule **(R1)** for base relation accesses duplicates the attributes from a base relation R and renames them according to the provenance attribute naming function $\mathscr{N}_q$[3]. The provenance attribute list of the rewritten base relation access contains the duplicated attributes.

Rewrite rule **(R2)**, the rule for $\Pi^{S/B}{}_A(q_1)$, rewrites a projection by adding the list of provenance attributes from $q_1{}^+$ to the projection list. For example, if q_1 is an access of base relation *items*, $\mathscr{P}(q^+)$ is $< \mathscr{N}(id), \mathscr{N}(price) >$. $(\Pi^{S/B}{}_A(item))^+$ preserves the complete tuples from relation $item$ that were used to compute the result of $\Pi_A(item)$.

The result of applying $+$ to a selection (rewrite rule **(R3)**) is generated by applying the unmodified selection to its rewritten input, because adding the provenance attributes to the input of the selection does not change the result of the selection condition. Therefore, only tuples that are extended versions of original

[3] Here q is the complete query that is rewritten, because the attribute names depend on the structure of the complete query.

Structural Rewrite

Unary Operators

$$q^+ = R^+ = \Pi_{\mathbf{R},\mathbf{R}\to\mathscr{N}(\mathbf{R})}(R) \tag{R1}$$

$$q^+ = (\sigma_C(q_1))^+ = \sigma_C({q_1}^+) \tag{R2}$$

$$q^+ = ({\Pi^{S/B}}_A(q_1))^+ = {\Pi^B}_{A,\mathscr{P}(q^+)}({q_1}^+) \tag{R3}$$

$$q^+ = (\alpha_{G,agg}(q_1))^+ = {\Pi^B}_{G,agg,\mathscr{P}(q^+)}(\alpha_{G,agg}(q_1) \mathbin{⟕}_{G=_nX} {\Pi^B}_{G\to X,\mathscr{P}({q_1}^+)}({q_1}^+)) \tag{R4}$$

Join Operators

$$q^+ = (q_1 \times q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \times {q_2}^+) \tag{R5.a}$$

$$q^+ = (q_1 \bowtie_C q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \bowtie_C {q_2}^+) \tag{R5.b}$$

$$q^+ = (q_1 \mathbin{⟕}_C q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \mathbin{⟕}_C {q_2}^+) \tag{R5.c}$$

$$q^+ = (q_1 \mathbin{⟖}_C q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \mathbin{⟖}_C {q_2}^+) \tag{R5.d}$$

$$q^+ = (q_1 \mathbin{⟗}_C q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^+)}({q_1}^+ \mathbin{⟗}_C {q_2}^+) \tag{R5.e}$$

Set Operations

$$q^+ = (q_1 \cup^{S/B} q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 \cup^S q_2 \mathbin{⟕}_{\mathbf{Q_1}=_nX} {\Pi^B}_{\mathbf{Q_1}\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \mathbin{⟕}_{\mathbf{Q_1}=_nY} {\Pi^B}_{\mathbf{Q_2}\to Y,\mathscr{P}({q_2}^+)}({q_2}^+)) \tag{R6.a}$$

$$q^+ = (q_1 \cup^{S/B} q_2)^+ = ({q_1}^+ \times null(\mathscr{P}({q_2}^+))) \cup^B ({\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}({q_2}^+ \times null(\mathscr{P}({q_1}^+)))) \tag{R6.b}$$

$$q^+ = (q_1 \cap^{S/B} q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 \cap^S q_2 \bowtie_{\mathbf{Q_1}=_nX} {\Pi^B}_{\mathbf{Q_1}\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \bowtie_{\mathbf{Q_1}=_nY} {\Pi^B}_{\mathbf{Q_2}\to Y,\mathscr{P}({q_2}^+)}({q_2}^+)) \tag{R7}$$

$$q^+ = (q_1 -^{S/B} q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}({\Pi^S}_{\mathbf{Q_1}}(q_1 -^{S/B} q_2) \bowtie_{\mathbf{Q_1}=_nX} {\Pi^B}_{\mathbf{Q_1}\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \mathbin{⟕}_{\mathbf{Q_1}\neq_n\mathbf{Q_2}} {q_2}^+) \tag{R8.a}$$

$$q^+ = (q_1 -^{S/B} q_2)^+ = {\Pi^B}_{\mathbf{Q_1},\mathscr{P}(q^+)}({\Pi^S}_{\mathbf{Q_1}}(q_1 -^{S/B} q_2) \bowtie_{\mathbf{Q_1}=_nX} {\Pi^B}_{\mathbf{Q_1}\to X,\mathscr{P}({q_1}^+)}({q_1}^+) \times null(\mathscr{P}({q_2}^+))) \tag{R8.b}$$

Provenance Attribute List Rewrite

$$\mathscr{P}(q^+) = \begin{cases} \mathscr{P}(q^+) & \text{if } q = \sigma_C(q_1) \vee q = \Pi_A(q_1) \vee q = \alpha_{G,agg}(q_1) \\ \mathscr{P}({q_1}^+) \blacktriangleright \mathscr{P}({q_2}^+) & \text{if } q = (q_1 \diamond_C q_2) \vee q = (q_1 \cup^{S/B} q_2) \vee q = (q_1 \cap^{S/B} q_2) \vee q = (q_1 -^{S/B} q_2) \\ \mathscr{N}(R) & \text{if } q = R \end{cases}$$

Figure 4.3: *PI-CS* Algebraic Rewrite Rules for Queries without Sublinks

result tuples are in the result of the rewritten selection. The provenance attribute list of a rewritten selection is the provenance attribute list of its rewritten input.

The rewrite rule **(R4)** rewrites an aggregation operation. We can not add additional tuples to the input of an aggregation or add additional attributes to its result schema without changing the values of the aggregation functions. This means we cannot propagate provenance information through an aggregation directly. Therefore, the rewritten query contains the original aggregation. The result of the original aggregation is joined with the rewritten version of q_1 using an equality condition on the grouping attributes. This is feasible because, according to the compositional semantics of *PI-CS*, all tuples with the same grouping attribute values as an result tuple t belong to the witness lists of t. We use an left outer join to handle the case of an aggregation with an empty input. According to the semantics of the aggregation operator the result of the aggregation is a single tuple in this case (with empty provenance). Note that the comparison operator $=_n$ instead of normal equality is used in the rewrite rule. This comparison operator is defined as:

$$a =_n b \Leftrightarrow a = b \vee (a \text{ is } \varepsilon \wedge b \text{ is } \varepsilon)$$

If the input of an aggregation contains tuples with *null* values in the group-by attributes, one output tuple is generated for this group of tuples. The $=_n$ comparison operator guarantees that the provenance of such a group is handled correctly. The provenance attribute list of a rewritten aggregation is the provenance attribute list of its rewritten input.

4.2.2 Join Operator Rewrite Rules

The rewrite rules **(R5.a)** to **(R5.e)** rewrite join operators.The provenance attribute list of a rewritten join operator is the concatenation of the provenance attribute lists of its rewritten inputs, because each witness list in the provenance of a join result contains a witness list for the left input and a witness list for a right input. $\diamond$ is used in the provenance attribute list construction as a placeholder for one of the join types of the *Perm* algebra. A join operator is rewritten by applying the join to the rewritten inputs and using a projection to produce the correct ordering of the result attributes (provenance attribute after normal attributes). This is possible, because adding provenance attributes to the input relations q_1 and q_2 does not change the result of the join condition.

4.2.3 Set Operations Rewrite Rules

The provenance attribute list of a rewritten set operation is the concatenation of the provenance attribute lists of its rewritten inputs. Recall that we defined two contribution semantics for union. One that combines two tuples from both inputs into a single witness list if they contributed to the same result tuple (this is the behaviour of *Lineage-CS*) and one that generates two separate witness lists for this case. Rewrite rule **(R6.a)** implements the first version. The desired combination of rewritten input tuples into a single output cannot be achieved by applying the union to the rewritten inputs. Thus, each input is rewritten separately and then joined with original union query on the complete set of original result attributes (Recall that according to the compositional semantics for union, input tuples belong to a witness list of a result tuple t if they are equal to t). We have to use outer joins to preserve tuples that are derived from only one of the inputs. The projection that is applied to the output of the joins removes superficial attributes introduced by the joins.

Rewrite rule **(R6.b)** implements the other contribution semantics for union. This rewrite rule simply extends the tuples from both rewritten inputs with null values to make them union compatible. Each tuple in the result of the rewritten query is derived from either the left or the right input and has the provenance attributes of the other rewritten input set to *null*. Hence, each tuple models a witness list of type $< t, \perp >$ or $< \perp, t >$ which are the two types of witness lists produced by the second compositional semantics of union.

The rewrite of an intersection operator (rewrite rule **(R7)**) is similar to rule (R6.a). It also uses joins to attach the provenance attributes from the rewritten inputs to the original result tuples of the intersection. In contrast to union it is not necessary to use outer joins, because each result tuple of an intersection is derived from tuples of both inputs.

Like for union, we proposed two contribution semantics for set difference. One that, like *Lineage-CS*, includes tuples from the right input of the set difference in the provenance and the other one that includes only tuples from the left input. Rewrite rule **(R8.a)** implements the first semantics in a similar way as the union and intersection rewrite rules. Rewrite Rule **(R8.b)** only joins the rewritten left input and fills the provenance attributes of the rewritten right input with null values. Note that the joins applied by the set operation rewrite rules use the $=_n$ comparison operator to deal with input tuples that contain null values.

4.2.4 Example Query Rewrite

As an example, reconsider query q_{ex} from the running example. Figure 4.4 presents the application of $+$ to this query. The top level operator of q_{ex} is an aggregation operator. Applying rewrite rule (R4) we get ${q_{ex}}^+$ as shown in 4.4 (**step 1**). Rule (R4) states that the $\mathscr{P}$-list for ${q_{ex}}^+$ equals $\mathscr{P}({q_1}^+)$. At this point, q^+ has not been computed, thus, $\mathscr{P}({q_{ex}}^+)$ is not expanded further in this step. The remaining sub-query q is a selection, which is left untouched by the rewrite (R2). The cross-product $q_{cross} = shop \times sales \times item$ is handled by rewrite rule (R5.a) (see 4.4 (**step 2**)). The $\mathscr{P}$-list of a rewritten cross product is the concatenation of the $\mathscr{P}$-lists of its rewritten inputs. In this case, the provenance attribute lists of rewritten

Original Query	$q_{ex} = \alpha_{name,sum(price)}(\sigma_{name=sName \wedge itemid=id}(shop \times sales \times items))$
Step 1	$q_{ex}{}^{+} = \Pi_{name,sum(price),\mathscr{P}(q^{+})}(\alpha_{name,sum(price)}(q) \bowtie_{name=x} \Pi_{name \rightarrow x,\mathscr{P}(q^{+})}(q^{+}))$ $q = \sigma_{name=sName \wedge itemid=id}(shop \times sales \times items)$ $\mathscr{P}(q_{ex}{}^{+}) = \mathscr{P}(q^{+})$
Step 2	$q^{+} = \Pi_{name,numEmpl,sName,itemId,id,price,\mathscr{P}(q^{+})}(\sigma_{name=sName \wedge itemid=id}(q_{cross}{}^{+}))$ $q_{cross}{}^{+} = shop^{+} \times sales^{+} \times items^{+}$ $\mathscr{P}(q^{+}) = \mathscr{P}(shop^{+}) \blacktriangleright \mathscr{P}(sales^{+}) \blacktriangleright \mathscr{P}(items^{+})$
Step 3	$shop^{+} = \Pi_{name,numEmpl,name \rightarrow \mathscr{N}(name),numEmpl \rightarrow \mathscr{N}(numEmpl)}(shop)$ $\mathscr{P}(shop^{+}) = (\mathscr{N}(name), \mathscr{N}(numEmpl))$
Step 4	$sales^{+} = \Pi_{sName,itemId,sName \rightarrow \mathscr{N}(sName),itemId \rightarrow \mathscr{N}(itemid)}(sales)$ $\mathscr{P}(sales^{+}) = (\mathscr{N}(sName), \mathscr{N}(itemId))$
Step 5	$items^{+} = \Pi_{id,price,id \rightarrow \mathscr{N}(id),price \rightarrow \mathscr{N}(price)}(items)$ $\mathscr{P}(items^{+}) = (\mathscr{N}(id), \mathscr{N}(price))$

$\mathbf{Q_{ex}}^{+}$

name	sum(price)	$\mathscr{N}$(name)	$\mathscr{N}$(numEmpl)	$\mathscr{N}$(sName)	$\mathscr{N}$(itemId)	$\mathscr{N}$(id)	$\mathscr{N}$(price)
Merdies	120	Merdies	3	Merdies	1	1	100
Merdies	120	Merdies	3	Merdies	2	2	100
Merdies	120	Merdies	3	Merdies	2	2	100
Joba	50	Joba	14	Joba	3	3	25
Joba	50	Joba	14	Joba	3	3	25

Figure 4.4: Example Application of the Provenance Rewrite Meta-Operator

base relations *shop*, *sales* and *items*. In Figure 4.4 (**steps 3-5**) rewrite rule (R1) is used to derive the rewritten base relations $shop^{+}$, $sales^{+}$ and $items^{+}$. The result of $q_{ex}{}^{+}$ is presented at the lower part of Figure 4.4. Note that $q_{ex}{}^{+}$ generates exactly the *PI-CS* provenance representation introduced in section 4.1.

If one takes a careful look at this example, it is obvious that if q_{ex} had been represented as an operator-tree, we would have computed the structural rewrite top-down and computed the $\mathscr{P}$-lists in a second bottom-up tree-traversal according to the sequence of operations applied in the example. A single bottom-up computation of a rewrite is possible as well, because the rewrite rules do not enforce a specific evaluation order. It seems that the bottom-up approach is advantageous, because the $\mathscr{P}$-lists of rewritten sub-expressions of a query q needed to compute q^{+}'s $\mathscr{P}$-list are immediately available, but as we will see in chapter 5, the bottom-up approach has other disadvantages.

Using the set of rewrite rules, we are able to transform a algebra expression q into a *single* relational algebra expression q^{+} generating the provenance of q. A major advantage of this approach is that provenance computation and normal queries are expressed with the same query language which enables provenance to be queried like normal data. For example, if a user needs to know which items where sold by shops with a total sales bigger than 100, this query can be represented as $q_1 = \Pi_{pId}(\sigma_{sum(price)>100}(q_{ex}{}^{+}))$. Note that it is possible to write down the algebra expression of this query as a query solely on $q_{ex}{}^{+}$, because of the direct association between provenance and original data in $Q_{ex}{}^{+}$, i.e., we can use provenance and original attributes in conditions and projections.

4.2.5 Proof of Correctness and Completeness

As mentioned before computing provenance by evaluating algebra expressions allows us to prove the correctness of provenance computation. To show that the $+$ meta-operator generates a query that computes provenance according to *PI-CS* we have to prove that the result Q^+ of the query q^+ generated by the algebraic rewrite rules is equal to Q^{PI}, the relational representation of provenance according to *PI-CS*.

Theorem 4.1 (Correctness and Completeness of the PI-CS Rewrite Rules)**.** *Let q be an algebra expression without sublink expressions. The result Q^+ of the algebra expression q^+ generated by applying the provenance rewrite meta-operator to $(q, <>)$ is equal to Q^{PI} and, thus, generates provenance according to PI-CS:*

$$Q^+ = Q^{PI}$$

Proof. We have to show that each tuple in Q^+ is of the form $(t, w[1]', \ldots, w[n]')$ for an original result tuple t and one of its witness lists w (soundness) and for each combination of $t \in Q$ and $w \in \mathscr{DD}(q,t)$ there is a corresponding tuple $(t, w[1]', \ldots, w[n]')$ in Q^+ (completeness). The proof of this theorem is twofold. First, we show that each tuple in Q^+ is an extension of an tuple from Q and that for each tuple in Q there is a corresponding extended tuple in Q^+. This proves that the tuples in Q^+ and Q^{PI} agree on the original result attributes. We call this property *result preservation*, because it states that all the original result tuples of q and nothing else is stored in the original result attributes of Q^+. Second, we prove that each extension $t^+ = (t, w[1]', \ldots, w[n]') \in Q^+$ of $t \in Q$ contains the relational representation of a witness list w from $\mathscr{DD}(q,t)$ and that Q^+ contains an extended tuple t^+ for each witness list in $\mathscr{DD}(q,t)$. We refer to this property as *witness list preservation*.

Result Preservation

The result preservation property is proven by showing that $\Pi^S{}_{\mathbf{Q}}(q^+) = \Pi^S{}_{\mathbf{Q}}(q)$. We prove this proposition by induction over the structure of an algebra expression. Assuming we have proven $\Pi^S{}_{\mathbf{Q}}(q^+) = \Pi^S{}_{\mathbf{Q}}(q)$ for all algebra expressions with maximal operator nesting depth i we have to show that $\Pi^S{}_{\mathbf{OP(Q)}}((op(q))^+) = \Pi^S{}_{\mathbf{OP(Q)}}(op(q))$ holds for every unary operator op and that $\Pi^S{}_{\mathbf{Q_1\,OP\,Q_2}}((q_1\,op\,q_2)^+) = \Pi^S{}_{\mathbf{Q_1\,OP\,Q_2}}(q_1\,op\,q_2)$ holds for every binary operator.

<u>Induction Start</u>:
The only algebra expression with nesting depth 0 is a base relation access R. The other nullary operator t is not of interest because its result is not derived from base data but instead generated by the query itself. Therefore, the provenance of this operator is empty.

$$\begin{aligned}
&\Pi^S{}_{\mathbf{R}}(R^+) && \\
=&\Pi^S{}_{\mathbf{R}}(\Pi^B{}_{\mathbf{R},\mathbf{R}\rightarrow\mathscr{N}(\mathbf{R})}(R)) && \text{(algebraic equivalences)}\\
=&\Pi^S{}_{\mathbf{R}}(R) &&
\end{aligned}$$

<u>Induction Step</u>:
<u>Case $q = \Pi^{S/B}{}_A(q_1)$:</u>

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+) && \\
=&\Pi^S{}_A((\Pi^{S/B}{}_A(q_1))^+) && \\
=&\Pi^S{}_A((\Pi^{S/B}{}_{A,\mathscr{P}(q_1{}^+)}(q_1{}^+))) && \\
=&\Pi^S{}_A(q_1{}^+) && \\
=&\Pi^S{}_A(\Pi^S{}_{\mathbf{Q_1}}(q_1{}^+)) && \text{(because expression A is defined solely over attributes from } \mathbf{Q_1}\text{)}\\
=&\Pi^S{}_A(\Pi^S{}_{\mathbf{Q_1}}(q_1)) && \text{(induction hypothesis)}\\
=&\Pi^S{}_A(\Pi^{S/B}{}_A(q_1)) && \\
=&\Pi^S{}_A(q) &&
\end{aligned}$$

Case $q = \sigma_C(q_1)$:

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+)\\
=&\Pi^S{}_{\mathbf{Q}}(\sigma_C(q_1{}^+))\\
=&\Pi^S{}_{\mathbf{Q}}(\sigma_C(\Pi^S{}_{\mathbf{Q_1}}(q_1{}^+))) && \text{(since C is defined solely over attributes from $\mathbf{Q}$ and $\mathbf{Q} = \mathbf{Q_1}$)}\\
=&\Pi^S{}_{\mathbf{Q}}(\sigma_C(\Pi^S{}_{\mathbf{Q_1}}(q_1))) && \text{(induction hypothesis)}\\
=&\Pi^S{}_{\mathbf{Q}}(\sigma_C(q_1))\\
=&\Pi^S{}_{\mathbf{Q}}(q)
\end{aligned}$$

Case $q = \alpha_{G,agg}(q_1)$:

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+)\\
=&\Pi^S{}_{G,agg}(\alpha_{G,agg}(q_1) ⟕_{G=_nX} \Pi^B{}_{G\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+))\\
=&\Pi^S{}_{G,agg}(\alpha_{G,agg}(q_1)) && \text{(semantics of left join and duplicate removal of projection)}\\
=&\Pi^S{}_{\mathbf{Q}}(q)
\end{aligned}$$

Case $q = q_1 \diamond_C q_2$:

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+)\\
=&\Pi^S{}_{\mathbf{Q_1},\mathbf{Q_2}}(q_1{}^+ \diamond_C q_2{}^+)\\
=&\Pi^S{}_{\mathbf{Q_1},\mathbf{Q_2}}(\Pi^B{}_{\mathbf{Q_1}}(q_1{}^+) \diamond_C \Pi^B{}_{\mathbf{Q_2}}(q_2{}^+)) && \text{(equivalence of $\Pi^B{}_{\mathbf{Q}}(q)$ with q)}\\
=&\Pi^S{}_{\mathbf{Q_1},\mathbf{Q_2}}(\Pi^S{}_{\mathbf{Q_1}}(q_1{}^+) \diamond_C \Pi^S{}_{\mathbf{Q_2}}(q_2{}^+)) && \text{(pushing duplicate removal into the join)}\\
=&\Pi^S{}_{\mathbf{Q_1},\mathbf{Q_2}}(\Pi^S{}_{\mathbf{Q_1}}(q_1) \diamond_C \Pi^S{}_{\mathbf{Q_2}}(q_2)) && \text{(induction hypothesis)}\\
=&\Pi^S{}_{\mathbf{Q}}(q)
\end{aligned}$$

Case $q = q_1 \cup^{S/B} q_2$ **(R6.a)**:

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+)\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 \cup^{S/B} q_2 ⟕_{\mathbf{Q_1}=_nX} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+) ⟕_{\mathbf{Q_1}=_nY} \Pi^B{}_{\mathbf{Q_2}\rightarrow Y,\mathscr{P}(q_2{}^+)}(q_2{}^+)))\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1}}(q_1 \cup^{S/B} q_2 ⟕_{\mathbf{Q_1}=_nX} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+) ⟕_{\mathbf{Q_1}=_nY} \Pi^B{}_{\mathbf{Q_2}\rightarrow Y,\mathscr{P}(q_2{}^+)}(q_2{}^+)))\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^S{}_{\mathbf{Q_1}}(q_1 \cup^{S/B} q_2)) && \text{(semantics of left outer join and duplicate removal of projection)}\\
=&\Pi^S{}_{\mathbf{Q}}(q)
\end{aligned}$$

Case $q = q_1 \cup^{S/B} q_2$ **(R6.b)**:

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+)\\
=&\Pi^S{}_{\mathbf{Q_1}}((q_1{}^+ \times null(\mathscr{P}(q_2{}^+))) \cup^B (\Pi_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_2{}^+ \times null(\mathscr{P}(q_1{}^+)))))\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^S{}_{\mathbf{Q_1}}((q_1{}^+ \times null(\mathscr{P}(q_2{}^+)))) \cup^B \Pi^S{}_{\mathbf{Q_1}}((\Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_2{}^+ \times null(\mathscr{P}(q_1{}^+))))))
\end{aligned}$$

Pushing projection into union:

$$\begin{aligned}
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^S{}_{\mathbf{Q_1}}(q_1{}^+)) \cup^B \Pi^S{}_{\mathbf{Q_2}}(\Pi^B{}_{\mathbf{Q_2}}(q_2{}^+))\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^S{}_{\mathbf{Q_1}}(q_1)) \cup^B \Pi^S{}_{\mathbf{Q_2}}(\Pi^S{}_{\mathbf{Q_2}}(q_2)) && \text{(induction hypothesis)}\\
=&\Pi^S{}_{\mathbf{Q}}(q)
\end{aligned}$$

Case $q = q_1 \cap^{S/B} q_2$:

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+)\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 \cap^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \bowtie_{\mathbf{Q_1}=_n Y} \Pi^B{}_{\mathbf{Q_2}\rightarrow Y,\mathscr{P}(q_2{}^+)}(q_2{}^+)))\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1}}(q_1 \cap^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \bowtie_{\mathbf{Q_1}=_n Y} \Pi^B{}_{\mathbf{Q_2}\rightarrow Y,\mathscr{P}(q_2{}^+)}(q_2{}^+)))
\end{aligned}$$

From the semantics of intersection and the induction hypothesis we know that every tuple from $q_1 \cap q_2$ will find join partners in $\Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+)$ and $\Pi^B{}_{\mathbf{Q_2}\rightarrow Y,\mathscr{P}(q_2{}^+)}(q_2{}^+)$.

$$\begin{aligned}
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1}}(q_1 \cap^{S/B} q_2))\\
=&\Pi^S{}_{\mathbf{Q}}(q)
\end{aligned}$$

Case $q = q_1 -^{S/B} q_2$ **(R8.a)**:

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+)\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 -^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \sqsupset\!\!\bowtie_{\mathbf{Q_1}\neq_n \mathbf{Q_2}} q_2{}^+))\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1}}(q_1 -^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \sqsupset\!\!\bowtie_{\mathbf{Q_1}\neq_n \mathbf{Q_2}} q_2{}^+))
\end{aligned}$$

From the semantics of set difference we know that every tuple from $q_1 -^{S/B} q_2$ will find at least one join partner in $\Pi_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+)$.

$$\begin{aligned}
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1}}(q_1 -^{S/B} q_2))\\
=&\Pi^S{}_{\mathbf{Q}}(q)
\end{aligned}$$

Case $q = q_1 -^{S/B} q_2$ **(R8.b)**:

$$\begin{aligned}
&\Pi^S{}_{\mathbf{Q}}(q^+)\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 -^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi_{\mathbf{Q_1}\rightarrow X,\mathscr{P}(q_1{}^+)}(q_1{}^+) \times null(\mathscr{P}(q_2{}^+))))\\
=&\Pi^S{}_{\mathbf{Q_1}}(\Pi^B{}_{\mathbf{Q_1}}(q_1 -^{S/B} q_2)) && \text{(same argument as for rewrite rule \textbf{(R8.a)})}\\
=&\Pi^S{}_{\mathbf{Q}}(q)
\end{aligned}$$

Witness List Preservation

To prove the *witness list preservation* property we have to show that each tuple in Q^+ is an extension of an original result tuple t with the relational representation of one of t's *PI-CS* witness lists. Let q be an algebra expression defined over base relations $R_1,\dots,R_n$ then we have to proof the following equivalence:

$$Q^+ = Q^{PI}$$

This equality can be expressed as:

$$u = (t, v_1, \dots, v_n) \in Q^+ \Leftrightarrow w \in \mathscr{DD}(q,t) \wedge \forall i : w[i]' = v_i$$

This is equivalent to $Q^+ = Q^{PI}$ because we have already proven that q^+ fulfills the result preservation property. Otherwise we would have to include an additional condition $t \in Q$ on the right side of the equivalence. As for *Result Preservation* we proof this property by induction over the structure of an algebra expression.

<u>**Induction Start**</u>:
<u>Case $q = R$</u>:
<u>$\Rightarrow$</u>:

$$\begin{aligned}
& u^m = (t, v_1)^m \in R^+ && \\
\Rightarrow & v_1 = t \wedge t^m \in R && (R^+ \text{ duplicates the attribute values of attributes in } \mathbf{R}) \\
\Rightarrow & w^m \in \mathscr{DD}(q,t) \wedge w[1]' = v_1 && (\text{since } \mathscr{DD}(R,t) = \{< t >^m \mid t^m \in R\})
\end{aligned}$$

<u>$\Leftarrow$</u>:

$$\begin{aligned}
& w^m \in \mathscr{DD}(q,t) \wedge w[1]' = v_1 && \\
\Rightarrow & v_1 = t && (\text{compositional semantics of } \mathscr{DD}) \\
\Rightarrow & u = (t, v_1)^m \in R^+ && (R^+ \text{ duplicates values of attributes in } \mathbf{R})
\end{aligned}$$

<u>**Induction Step for Unary Operators**</u>:

Assuming that *witness list preservation* holds for algebra expressions with a maximal nesting depth i we have to show that this property holds for algebra expressions with maximal nesting depth $i+1$. Hence, for unary operators op we have to show that for an algebra expression $q = op(q_1)$ with nesting depth $i+1$ the following holds under the assumption that $(t, v_1, \dots, v_n) \in {Q_1}^+ \Leftrightarrow w \in \mathscr{DD}(q_1, t) \wedge w[i]' = v_i$ holds:

$$(t, v_1, \dots, v_n)^m \in Q^+ \Leftrightarrow w^m \in \mathscr{DD}(q,t) \wedge w[i]' = v_i$$

Using the definition of transitivity for $\mathscr{DD}$ the right hand side can be transformed into:

$$\begin{aligned}
& (w^x \in \mathscr{DD}(op(Q_1),t) \wedge w = < t' > \wedge {w_1}^m \in \mathscr{DD}(q_1,t') \wedge w_1[i]' = v_i) \\
& \vee (w^m \in \mathscr{DD}(op(Q_1),t) \wedge w = <\perp> \wedge m = 1 \wedge \forall i : v_i = (\varepsilon, \dots, \varepsilon))
\end{aligned}$$

Substituting the induction hypothesis we get:

$$\begin{aligned}
& (w^x \in \mathscr{DD}(op(Q_1),t) \wedge w = < t' > \wedge (t', v_1, \dots, v_n)^m \in {Q_1}^+) \\
& \vee (w^m \in \mathscr{DD}(op(Q_1),t) \wedge w = <\perp> \wedge m = 1 \wedge \forall i : v_i = (\varepsilon, \dots, \varepsilon))
\end{aligned}$$

Thus, we have to prove the following equivalence:

$$\begin{aligned}
& (t, v_1, \dots, v_n) \in Q^+ \\
\Leftrightarrow & \\
& (w^x \in \mathscr{DD}(op(Q_1),t) \wedge w = < t' > \wedge (t', v_1, \dots, v_n)^m \in {Q_1}^+) \\
& \vee (w^m \in \mathscr{DD}(op(Q_1),t) \wedge w = <\perp> \wedge m = 1 \wedge \forall i : v_i = (\varepsilon, \dots, \varepsilon))
\end{aligned}$$

This means q^+ produces correct results if it propagates witness list representations correctly. For a witness list representation w' from the rewritten input, q^+ has to attach w' to a tuple t in the output iff t is derived from a tuple t' from the input and in the rewritten input t' is attached to w'.

<u>Case $q = \sigma_C(q_1)$</u>:
Applying rewrite rule (R2) to q we get $q^+ = \sigma_C({q_1}^+)$. From the compositional semantics of *PI-CS* for selection we know that a witness list w in $\mathscr{DD}(\sigma_C(Q_1),t)$ will never contain $\perp$. Therefore, the right disjunct of the right hand side of the equivalence we have to prove is never fulfilled. It follows that we can simplify the right hand side to:

$$(w^x \in \mathscr{DD}(op(Q_1),t) \wedge w = < t' > \wedge (t', v_1, \dots, v_n)^m \in {Q_1}^+)$$

Using the simplified equivalence we get:

$$\begin{aligned}
&(t, v_1, \dots, v_n)^m \in Q^+ \\
\Leftrightarrow &(t', v_1, \dots, v_n)^m \in {Q_1}^+ \wedge t' = t && \text{(definition of selection)} \\
\Leftrightarrow &(t', v_1, \dots, v_n) \in {Q_1}^+ \wedge w^m =< t' >\in \mathscr{D}\mathscr{D}(\sigma_C(Q_1), t) && \text{(compositional semantics of PI-CS)}
\end{aligned}$$

Case $q = \Pi^{S/B}{}_A(q_1)$:
Applying rewrite rule (R3) to q we get $q^+ = \Pi^B{}_{A,\mathscr{P}(q^+)}({q_1}^+) = \Pi^B{}_{A,\mathscr{P}({q_1}^+)}({q_1}^+)$. Using the same arguments as applied in the proof for selection we deduce that the simplified equivalence can be applied in the proof for projection too.

$$\begin{aligned}
&(t, v_1, \dots, v_n)^m \in Q^+ \\
\Leftrightarrow &(t', v_1, \dots, v_n)^m \in {Q_1}^+ \wedge t'.A = t && \text{(definition of projection)} \\
\Leftrightarrow &(t', v_1, \dots, v_n)^m \in {Q_1}^+ \wedge w^m =< t' >\in \mathscr{D}\mathscr{D}(\Pi^{S/B}{}_A(Q_1), t) && \text{(compositional semantics of PI-CS)}
\end{aligned}$$

Case $q = \alpha_{G,agg}(q_1)$:
Applying rewrite (R2) to q we get $q^+ = \Pi^B{}_{G,agg,\mathscr{P}(q^+)}(\alpha_{aggr,G}(q_1) \mathbin{⟕}_{G=_nX} \Pi^B{}_{G \rightarrow X,\mathscr{P}({q_1}^+)}({q_1}^+))$ with $\mathscr{P}(q^+) = \mathscr{P}({q_1}^+)$. We distinguish between two cases:

1. $G = ()$ and $Q_1 = \emptyset$
2. else

Case 1: For the first case we know that $\mathscr{D}\mathscr{D}(q,t) = \{<\perp>\}$. Therefore, the left disjunction of the right-hand side of the equivalence we have to prove can be removed. In the following let $v = (v_1, \dots, v_n)$.

$$\begin{aligned}
&(t, v_1, \dots, v_n)^m \in Q^+ \\
\Leftrightarrow &t \in Q \wedge (((u,v)^m \in Q' \wedge q' = \Pi^B{}_{G \rightarrow X,\mathscr{P}({q_1}^+)}({q_1}^+) \wedge u.X =_n t.G) \vee (v = \varepsilon, \dots \varepsilon \wedge Q_1 = \emptyset)) \\
\Leftrightarrow &t \in Q \wedge (v = \varepsilon, \dots, \varepsilon) \in Q' \\
\Leftrightarrow &<\perp>\in \mathscr{D}\mathscr{D}(\alpha_{G,agg}(Q_1), t)
\end{aligned}$$

Case 2: For the second case we know that every result tuple t in $[[\alpha_{G,agg}(q_1)]]$ will find a join partner in $\Pi^B{}_{G \rightarrow X,\mathscr{P}({q_1}^+)}({q_1}^+)$, because otherwise t would not in Q.

$$\begin{aligned}
&(t, v_1, \dots, v_n)^m \in Q^+ \\
\Leftrightarrow &(u,v)^m \in Q' \wedge q' = \Pi^B{}_{G \rightarrow X,\mathscr{P}({q_1}^+)}({q_1}^+) \wedge u.X = t.G \wedge u.v = t.v && \text{(definition of left join)} \\
\Leftrightarrow &(t', v_1, \dots, v_n)^m \in {Q_1}^+ \wedge t'.G = t.G \\
\Leftrightarrow &(t', v_1, \dots, v_n)^m \in {Q_1}^+ \wedge w^x =< t' >\in \mathscr{D}\mathscr{D}(\alpha_{G,agg}(Q_1), t)
\end{aligned}$$

Induction Step for Binary Operators:

To prove witness list preservation for binary operators we use the same approach as applied for unary operators. The difference is that applying the transitivity definition of *PI-CS* to $q = q_1 \; op \; q_2$ generates a more complex equivalence:

$$(t, v_1, \dots, v_n, u_1, \dots, u_m)^{p \times r} \in Q^+ \Leftrightarrow case_1 \vee case_2 \vee case_3 \vee case_4$$

where

$$
\begin{aligned}
case_1 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< t', t'' > \\
& \wedge w[q_1]^p \in \mathscr{DD}(q_1, t') \wedge w[i]' = v_i \wedge w[q_2]^r \in \mathscr{DD}(q_2, t'') \wedge w[i+n]' = u_i) \\
case_2 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< t', \perp > \\
& \wedge w[q_1]^p \in \mathscr{DD}(q_1, t') \wedge w[i]' = v_i \wedge \forall i : u_i = (\varepsilon, \dots, \varepsilon) \wedge r = 1) \\
case_3 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< \perp, t' > \\
& \wedge \forall i : v_i = (\varepsilon, \dots, \varepsilon) \wedge p = 1 \wedge w[q_2]^r \in \mathscr{DD}(q_2, t') \wedge w[n+i]' = u_i) \\
case_4 =& (w \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< \perp, \perp > \wedge \forall i : v_i = (\varepsilon, \dots, \varepsilon) \wedge \forall i : u_i = (\varepsilon, \dots, \varepsilon) \wedge p = r = 1)
\end{aligned}
$$

Substituting the induction hypothesis for each of these cases produces the following equivalent formulation:

$$
\begin{aligned}
case_1 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< t', t'' > \\
& \wedge (t', v_1, \dots, v_n)^p \in Q_1{}^+ \wedge (t', u_1, \dots, u_m)^r \in Q_2{}^+) \\
case_2 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< t', \perp > \\
& \wedge (t', v_1, \dots, v_n)^p \in Q_1{}^+ \wedge \forall i : u_i = (\varepsilon, \dots, \varepsilon) \wedge r = 1) \\
case_3 =& (w^x \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< \perp, t' > \\
& \wedge \forall i : v_i = (\varepsilon, \dots, \varepsilon) \wedge (t', u_1, \dots, u_m)^r \in Q_2{}^+) \\
case_4 =& (w \in \mathscr{DD}(Q_1\ op\ Q_2, t) \wedge w =< \perp, \perp > \wedge \forall i : v_i = (\varepsilon, \dots, \varepsilon) \wedge \forall i : u_i = (\varepsilon, \dots, \varepsilon) \wedge p = r = 1)
\end{aligned}
$$

We prove this equivalence by identifying under which pre-conditions each of the cases is fulfilled (the individual case are non-overlapping) and then under assumption of these pre-conditions prove the equivalence of the left hand side with this case. Note that some of these cases can be precluded for several of the binary operators, because the provenance of these operators never contains witness lists of the requested format.

Case $q = q_1 \times q_2$:

According to the compositional semantics of *PI-CS* for cross product each witness list is of the form $w =< u, v >$ where u respective v are tuples from Q_1 respective Q_2. Therefore, all cases except $case_1$ can be excluded.

$$
\begin{aligned}
& (t, v_1, \dots, v_n, u_1, \dots, u_m)^{p \times r} \in Q^+ \\
\Leftrightarrow & (t.\mathbf{Q_1}, v_1, \dots, v_n)^p \in Q_1{}^+ \wedge (t.\mathbf{Q_2}, u_1, \dots, u_m)^r \in Q_2{}^+ && \text{(semantics of projection and cross product)} \\
\Leftrightarrow & (t', v_1, \dots, v_n)^p \in Q_1{}^+ \wedge (t'', u_1, \dots, u_m)^r \in Q_2{}^+ \wedge t' = t.\mathbf{Q_1} \wedge t'' = t.\mathbf{Q_2} \\
\Leftrightarrow & w^x =< t', t'' > \in \mathscr{DD}(Q_1 \times Q_2, t) && \text{(compositional semantics for cross product)} \\
& \wedge (t', v_1, \dots, v_n)^p \in Q_1{}^+ \wedge (t'', u_1, \dots, u_m)^r \in Q_2{}^+
\end{aligned}
$$

Case $q = q_1 \bowtie_C q_2$:

According to the compositional semantics of *PI-CS* for join each witness list is of the form $w =< u, v >$ where u respective v are tuples from Q_1 respective Q_2. Therefore, all cases except case 1 can be excluded.

$$
\begin{aligned}
&(t, v_1, \dots, v_n, u_1, \dots, u_m)^{p \times r} \in Q^+ \\
\Leftrightarrow &(t.\mathbf{Q_1}, v_1, \dots, v_n)^p \in {Q_1}^+ \wedge (t.\mathbf{Q_2}, u_1, \dots, u_m)^r \in {Q_2}^+ && \text{(semantics of projection and join)} \\
\Leftrightarrow &(t', v_1, \dots, v_n)^p \in {Q_1}^+ \wedge (t'', u_1, \dots, u_m)^r \in {Q_2}^+ \wedge t' = t.\mathbf{Q_1} \wedge t'' = t.\mathbf{Q_2} \\
\Leftrightarrow &w^x = < t', t'' >^x \in \mathscr{DD}(Q_1 \bowtie_C Q_2, t) \wedge (t', v_1, \dots, v_n)^p \in {Q_1}^+ \wedge (t'', u_1, \dots, u_m)^r \in {Q_2}^+ \\
& && \text{(compositional semantics for join)}
\end{aligned}
$$

Case $q = q_1 ⟕_C q_2$:

Each witness list w in the *PI-CS* provenance for left outer join is either of the form $< u, v >$ with u from Q_1 and v from Q_2 or of form $< u, \perp >$ with u from Q_1. Hence, only cases 1 and 2 have to be considered.

Case 1: According to the compositional semantics for left outer join case 1 applies if $t \models C$. In this case the semantics of left outer join, its compositional semantics and rewrite rule coincide with the inner join case.

Case 2: Case 2 applies if $t \not\models C$ and, therefore, $t = (t', \varepsilon, \dots, \varepsilon)$ where $t' \in Q_1$. This means t is generated from a left hand side input tuple that does not have a join partner in Q_2.

$$
\begin{aligned}
&(t, v_1, \dots, v_n, \varepsilon, \dots, \varepsilon)^p \in Q^+ \wedge t \not\models C \\
\Leftrightarrow &(t', v_1, \dots, v_n)^p \in {Q_1}^+ \wedge t = (t', \varepsilon, \dots, \varepsilon) \in Q \wedge t \not\models C \\
\Leftrightarrow &(t', v_1, \dots, v_n)^p \in {Q_1}^+ \wedge w^x = < t', \perp >^x \in \mathscr{DD}(Q_1 ⟕_C Q_2, t)
\end{aligned}
$$

Case $q = q_1 ⟖_C q_2$:

For right outer join only cases 1 and 3 apply. The proves for both cases are analog to the proves for left outer join.

Case $q = q_1 ⟗_C q_2$:

For full outer join case 1 to 3 apply and are proven as for the other outer join types.

Case $q = q_1 \cup^{S/B} q_2$ (*PI* semantics):

All witness lists in the provenance of a union are either $< u, \perp >$ or $< \perp, v >$ with $u \in Q_1$ and $v \in Q_2$ if *PI-CS* semantics are applied. This means only cases 2 and 3 apply.

Case 2:

$$
\begin{aligned}
&(t, v_1, \dots, v_n, \varepsilon, \dots, \varepsilon)^p \in Q^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in {Q_1}^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in {Q_1}^+ \wedge w^x \in \mathscr{DD}(Q_1 \cup Q_2) \wedge w = < t, \perp >
\end{aligned}
$$

Case 3: Is symmetric to the proof of case 2.

Case $q = q_1 \cup^{S/B} q_2$ (alternative semantics):

For the alternative semantics of the provenance fo union (rewrite rule **6.a**) cases 1,2, and 3 apply.

Case 1:

$$
\begin{aligned}
&(t, v_1, \dots, v_n, u_1, \dots, u_m)^{p \times r} \in Q^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in {Q_1}^+ \wedge (t, u_1, \dots, u_m)^r \in {Q_2}^+ && \text{(semantics of left outer join)} \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in {Q_1}^+ \wedge (t, u_1, \dots, u_m)^r \in {Q_2}^+ \wedge w^x \in \mathscr{DD}(Q_1 \cup^{S/B} Q_2) \wedge w = < t, t >
\end{aligned}
$$

Case 2:

$$
\begin{aligned}
&(t, v_1, \dots, v_n, \varepsilon, \dots, \varepsilon)^p \in Q^+ \wedge \neg \exists (t, u_1, \dots, u_m) \in {Q_2}^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in {Q_1}^+ \wedge \neg \exists (t, u_1, \dots, u_m) \in {Q_2}^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in {Q_1}^+ \wedge w^x \in \mathscr{DD}(Q_1 \cup^{S/B} Q_2) \wedge w = < t, \perp >
\end{aligned}
$$

Case 3: Is symmetric to the proof of case 2.
Case $q = q_1 \cap^{S/B} q_2$:
For intersection all witness lists are of the form $< u, v >$ with $u \in Q_1$ and $v \in Q_2$. Only case 1 applies.

$$\begin{aligned}
&(t, v_1, \dots, v_n, u_1, \dots, u_m)^{p \times r} \in Q^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in Q_1{}^+ \wedge (t, u_1, \dots, u_m)^r \in Q_2{}^+ && \text{(semantics of join)} \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in Q_1{}^+ \wedge (t, u_1, \dots, u_m)^r \in Q_2{}^+ \wedge < t, t >^x \in \mathscr{DD}(Q_1 \cap^{S/B} Q_2)
\end{aligned}$$

Case $q = q_1 -^{S/B} q_2$ (*PI-CS* semantics):
All witness lists for set difference are of the form $< u, \perp >$ with $u \in Q_1$. Therefore, only case 2 applies

$$\begin{aligned}
&(t, v_1, \dots, v_n, \varepsilon, \dots, \varepsilon)^p \in Q^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in Q_1{}^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in Q_1{}^+ \wedge < t, \perp >^x \in \mathscr{DD}(Q_1 -^{S/B} Q_2)
\end{aligned}$$

Case $q = q_1 -^{S/B} q_2$ (alternative semantics):
Under the alternative semantics a witness lists from $\mathscr{DD}(q,t)$ is either of form $< t, v >$ with $t \in Q_1$ and $v \in Q_2$ if Q_2 contains tuples that are not equal to t or of form $< t, \perp >$ otherwise. This means cases 1 and 2 apply.
Case 1:

$$\begin{aligned}
&(t, v_1, \dots, v_n, u_1, \dots, u_m)^{p \times r} \in Q^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in Q_1{}^+ \wedge (t', u_1, \dots, u_m)^r \in Q_2{}^+ \wedge t \neq t' \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in Q_1{}^+ \wedge (t', u_1, \dots, u_m)^r \in Q_2{}^+ \wedge < t, t' >^x \in \mathscr{DD}(Q_1 -^{S/B} Q_2, t)
\end{aligned}$$

Case 2:

$$\begin{aligned}
&(t, v_1, \dots, v_n, \varepsilon, \dots, \varepsilon)^p \in Q^+ \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in Q_1{}^+ \wedge \nexists (t', u_1, \dots, u_m) \in Q_2{}^+ : t \neq t' \\
\Leftrightarrow &(t, v_1, \dots, v_n)^p \in Q_1{}^+ \wedge < t, \perp >^x \in \mathscr{DD}(Q_1 -^{S/B} Q_2, t)
\end{aligned}$$

□

With the prove of theorem 4.1 we have established an very important property. Given a query q (without sublinks) we know how to transform it into a query q^+ by applying the meta-operator $+$. q^+ is guaranteed to compute the provenance of q according to *PI-CS* alongside with the original results of q. Furthermore, in the result of Q^+ each original tuple t is extended with the relational representations of its *PI-CS* witness lists. The correctness and completeness of $+$ was proven in two steps. First, we demonstrated that q^+ preserves the original result tuples of q (*result preservation*). I.e., if Q contains a tuples t then Q^+ will contain extended versions of t and all tuples (t, v) in Q^+ are extended versions of tuples from Q. Second, we have proven that each extended version of a tuple t in Q^+ is generated by attaching the relational representation of a witness list from $\mathscr{DD}(q,t)$ to t (*witness list preservation*). This means only relational representations of witness lists from $\mathscr{DD}(q,t)$ are included in Q^+ and q^+ generates the correct associations between witness lists and original result tuples as requested by the relational representation Q^{PI} of *PI-CS* provenance. In the next section we extend the rewrite rules for algebra expressions with sublinks and prove the correctness and completeness of these extensions.

4.3 *PI-CS* Rewrite Rules for Queries with Sublinks

In this section we present rewrite rules that compute the $+$ meta-operator for algebra expression that contain sublinks. We first introduce one set of rewrite rules, called the *Gen* rewrite strategy, that is applicable to all algebra expressions with sublinks, but is not very efficient. Afterwards, we present several specialized rewrite strategies that use un-nesting and de-correlation (see, e.g., [Kim82]) to reduce the complexity of the rewritten algebra expressions. Thus, increasing the likelihood that the optimizer of a DBMS generates an efficient execution plan for the rewritten query. The disadvantage of these specialized rewrite strategies is that they are only applicable to algebra expressions that fulfill certain preconditions. E.g., some strategies are only applicable to algebra expressions that contain only uncorrelated sublink expressions.

4.3.1 Gen Rewrite Strategy

We now present the *Gen* rewrite strategy for algebra expressions that contain sublinks and prove that the rewrite rules of the strategy compute provenance according to Definition 3.12. The *Gen* strategy uses the *PI-CS* rewrite rules for standard algebra expressions and additional rewrite rules to transform sublinks.

The two main problems in developing rewrite rules for sublinks are that (1) the result of a sublink query is not included in the query result which complicates the propagation of provenance information, and that (2) it is not immediately clear how to determine the influence role of a sublink (see 3.2.3.1) which is needed to determine the provenance of the sublink. We approach the first problem by joining the original query with the sublink query, and the second by restricting the join to filter out tuples according to the influence role of the sublink. For correlated sublinks it is not possible to simply join the sublink query, because correlations are only allowed in sublinks but not in standard algebra expressions.

One approach to overcome this problem is to completely de-correlate the query by injecting the top query into the sublink query, produce results for each correlated attribute binding and propagate the bindings throughout the query. The propagated attributes are then used to bind the correlated attributes values in the join condition. This is similar to the query un-nesting and de-correlation problem studied in area of query optimization (see [Cha98]). However, the solutions from this field are only applicable for specific correlations. Even if we do not consider the performance of rewritten queries, the limitations to a subset of sublink expressions is not acceptable. Therefore, we first aim at developing a strategy that is applicable to all algebra expressions with sublinks and postpone the discussion of more efficient strategies with limited applicability to section 4.3.2.

To circumvent de-correlation the *Gen* strategy joins the original query with all theoretically possible relational witness list representations and simulates a join by an additional sublink that filters out witness list representations that do not belong to the provenance. All possible witness list representations for the provenance of a sublink can be produced using the cross product of all base relations accessed by the sublink query. In some cases, e.g. if a base relation is the empty set or a witness list contains $\perp$, a tuple consisting of *null* values is the representation of a witness list for this relation. Therefore, we extend each base relation R with a tuple $null(R)$. Recall that $null(R)$ is a singleton relation that has the schema $\mathbf{R}$ with all attributes set to ε.

Let $R_1, \ldots, R_n$ be the list of all base relations accessed by a sublink query q_{sub}. The algebra expression $CrossBase(q_{sub})$ is defined as follows:

$$CrossBase(q_{sub}) = \Pi^B{}_{\mathbf{R_1} \rightarrow \mathscr{N}(R_1)}(R_1 \cup^B null(\mathbf{R_1})) \times \ldots \times \Pi^B{}_{\mathbf{R_n} \rightarrow \mathscr{N}(R_n)}(R_n \cup null(\mathbf{R_n}))$$

$CrossBase(q_{sub})$ is the set of all relational representations of witness lists that may occur in the provenance of a sublink q_{sub} according to *PI-CS*.

To restrict the *CrossBase* of a sublink to the representations of witness lists from $\mathscr{DD}(q,t)$, we need to know which influence role C_{sub} has for every regular input tuple t. In addition, we need to know the set of witness lists for the sublink query q_{sub}. The provenance of the sublink query can be computed by applying $+$ to produce ${q_{sub}}^+$. To access the influence role, we can make use of the sublink condition C_{sub} itself. The join condition can then be formulated using C_{sub} and a condition ${C_{sub}}'$ that filters out the witness list representations from the *CrossBase* according to the influence role of C_{sub}.

$$(\sigma_C(q_1))^+ = \sigma_{C \wedge C_{sub_1}{}^+ \ldots \wedge C_{sub_n}{}^+}(q_1{}^+ \times CrossBase(q_{sub_1}) \ldots \times CrossBase(q_{sub_n})) \quad \textbf{(G1)}$$

$$(\Pi^{S/B}{}_A(q_1))^+ = \sigma_{C_{sub_1}{}^+ \wedge \ldots \wedge C_{sub_n}{}^+}(\Pi^B{}_{A,\mathscr{P}(q_1{}^+)}(q_1{}^+) \times CrossBase(q_{sub_1}) \ldots \times CrossBase(q_{sub_n})) \quad \textbf{(G2)}$$

$$\begin{aligned} C_{sub_i}{}^+ = EXISTS\ (\sigma_{J_{sub_i} \wedge \mathscr{P}(q_{sub_i}{}^+) =_n X}(\Pi^B{}_{\mathscr{P}(q_{sub_i}{}^+) \to X}(q_{sub_i}{}^+)) \\ \vee (\neg\ EXISTS\ (q_{sub_i}) \wedge \mathscr{P}(q_{sub_i}{}^+) \text{ is } \varepsilon) \end{aligned}$$

Figure 4.5: *Gen* Strategy Rewrite Rules

As presented in section 3.2.3, the provenance of a sublink contains either all tuples from Q_{sub}, $Q_{sub}{}^{true}$ or $Q_{sub}{}^{false}$. Thus, the correct filter condition for filtering out witness list representations is either $true$, $e\ op\ t$ or $\neg(e\ op\ t)$. We use the following notations:

$$\begin{aligned} C_{sub}{}' &= e\ op\ t \\ \neg C_{sub}{}' &= \neg(e\ op\ t) \end{aligned}$$

$C_{sub}{}'$ and C_{sub} are used in combination to define a selection condition J_{sub} for each sublink type that is used to filter out witness list representations from the provenance of $q_{sub}{}^+$ that do not belong to the provenance according to the influence role of q_{sub}:

$$\begin{aligned} J_{sub} &= (C_{sub} \wedge C_{sub}{}') \vee \neg C_{sub} && \text{(ANY-sublink)} \\ J_{sub} &= C_{sub} \vee (\neg C_{sub} \wedge \neg C_{sub}{}') && \text{(ALL-sublink)} \\ J_{sub} &= true && \text{(EXISTS- or scalar sublink)} \end{aligned}$$

By applying logical equivalences, J_{sub} for *ANY*- and *ALL*-sublink can be simplified into:

$$\begin{aligned} J_{sub} &= C_{sub}{}' \vee \neg C_{sub} && \text{(ANY-sublink)} \\ J_{sub} &= C_{sub} \vee \neg C_{sub}{}' && \text{(ALL-sublink)} \end{aligned}$$

The condition J_{sub} is used to restrict $q_{sub}{}^+$ to the actual provenance of C_{sub}. For instance, if $C_{sub} = a\ =\ ANY\ (\sigma_{b>3}(S))$ then J_{sub} would be:

$$C_{sub}{}' \vee C_{sub} = (a = b) \vee \neg(a\ =\ ANY\ (\sigma_{b>3}(S)))$$

Before presenting the application of J_{sub} let us discuss how the join between *CrossBase* and q^+ is simulated with equality conditions between the attributes from *CrossBase* and the attributes from $q_{sub}{}^+$. In this comparison we have to consider *null* values as equal and, therefore, use the comparison operator $=_n$. The rewritten sublink expression $C_{sub}{}^+$ that simulates the join condition between *CrossBase* and $q_{sub}{}^+$ and applies J_{sub} is defined as follows:

$$\begin{aligned} C_{sub}{}^+ = EXISTS\ (\sigma_{J_{sub} \wedge \mathscr{P}(q_{sub}{}^+) =_n X}(\Pi^B{}_{\mathscr{P}(q_{sub}{}^+) \to X}(q_{sub}{}^+))) \\ \vee (\neg\ EXISTS\ (q_{sub}) \wedge \mathscr{P}(q_{sub}{}^+) \text{ is } \varepsilon) \end{aligned}$$

The first *EXISTS* sublink expression in $C_{sub}{}^+$ checks that a tuple from the *CrossBase* actually is a part of the relational representation of a witness list that belongs to the provenance of the sublink C_{sub}. To belong to the provenance a tuple has to be an element of $q_{sub}{}^+$. This is checked by the condition $\mathscr{P}(q_{sub}{}^+) =_n X$. In addition, the tuple has to fulfill the condition J_{sub}. The second *EXISTS* sublink expression is needed to handle the special case of an empty sublink query result. In this case the provenance attributes are filled with *null* values.

$C_{sub}{}^+$ enables us to define the *Gen* strategy rewrite rules for selections and projections with multiple sublinks. These rules are presented in Figure 4.5 as rewrite rules **(G1)** and **(G2)**.

R

a	b
1	1
2	1
3	2

S

c
1
2
4

Q

a	b
1	1

$$q = \sigma_{C_{sub}}(R) \qquad C_{sub} = (a \; = \; ANY \; (\sigma_{c=b}(S)))$$

$$\mathcal{DD}(q,(1,1)) = \{< (1,1),(1) >\}$$

$$\begin{aligned} q^+ =& \sigma_{C_{sub} \wedge C_{sub}{}^+}(\Pi_{a,b,a\to\mathcal{N}(a),b\to\mathcal{N}(b)}(R) \times \Pi_{c\to\mathcal{N}(c)}(S \cup^B \{(\varepsilon)\})) \\ C_{sub}{}^+ =& EXISTS \; (\sigma_{(a=c \vee \neg C_{sub}) \wedge \mathcal{N}(c) =_n xc}(\Pi_{c,\mathcal{N}(c)\to xc}(\sigma_{c=b}(\Pi_{c,c\to\mathcal{N}(c)}(S))))) \\ & \vee (\neg \; EXISTS \; (\sigma_{c=b}(S)) \wedge \mathcal{N}(c) \text{ is } \varepsilon) \end{aligned}$$

Q^+

a	b	$\mathcal{N}(a)$	$\mathcal{N}(b)$	$\mathcal{N}(c)$
1	1	1	1	1

Figure 4.6: Example Application of the *Gen* Strategy

Example 4.4. *As an example for an application of the Gen strategy consider the query q presented in Figure 4.6. The sublink query in C_{sub} accesses only base relation S. Therefore, the CrossBase of C_{sub} is $\Pi^B{}_{c\to\mathcal{N}(c)}(S \cup^B null(\mathbf{S})) = \Pi^B{}_{c\to\mathcal{N}(c)}(S \cup \{(\varepsilon)\})$. The regular input of the selection in q is the base relation access R. Hence, the cross produce between the rewritten regular input and the CrossBase is $q_{cross} = \Pi^B{}_{a,b,a\to\mathcal{N}(a),b\to\mathcal{N}(b)}(R) \times \Pi^B{}_{c\to\mathcal{N}(c)}(S \cup \{(\varepsilon)\})$. The result of q_{cross} contains all tuples from the rewritten regular input and the provenance attribute part of all potential result tuples of $q_{sub}{}^+$: $\{(1),(2),(4),(\varepsilon)\}$. The rewritten sublink condition $C_{sub}{}^+$ uses $q_{sub}{}^+$ to retrieve the provenance of $q_{sub}{}^+$. The witness list representations in the provenance are filtered according to the influence role of C_{sub}. In this case C_{sub} is reqtrue for all regular input tuples. Therefore, only tuples that fulfill the condition $a = c$ are not filtered out by the J_{sub} condition $(a = c \vee \neg C_{sub})$. The condition $a = c$ holds for all regular input tuples for which the selection condition C is true and for tuples from $Q_{sub}{}^+$ that are extended versions of tuples from $Q_{sub}{}^{true}$. Finally, the condition $\mathcal{N}(c) =_n xc$ checks that only tuples from the CrossBase that have the same provenance attribute values as the tuples from $Q_{sub}{}^+$ are in the result of Q^+. The result relation Q^+ is presented at the lower part of Figure 4.6. As expected it contains a single tuple that represents the result tuple $(1,1)$ and the only witness list $< (1,1),(1) >$ in the set $\mathcal{DD}(q,(1,1))$.*

4.3.1.1 Proof of Correctness and Completeness

In the last section we demonstrated by means of an example that the *Gen* strategy generates correct provenance information. Let us now formally prove the correctness of this strategy for arbitrary algebra expressions.

Theorem 4.2 (Correctness and Completeness of the Gen Strategy)**.** *For a query q with sublinks, the provenance computed by the rewritten query q^+ according to rules (G1) and (G2) is the provenance derived according to definition 3.12:*

$$Q^+ = Q^{PI}$$

Proof.
To prove the correctness of the *Gen* rewrite rules, we have to show that each tuple t produced by the rewritten query is an original tuple with attached provenance, and that for every original tuple with attached provenance this tuple is included in the result of the rewritten query. Similar to the proof for the standard operator rewrite rules, we proof this equivalence by showing that the *result preservation* and *witness list preservation* properties are fulfilled.

Result Preservation:
Case (G1):
To show that the original result attributes $\mathbf{Q}$ of a rewritten query q^+ contain exactly the result tuples from Q, we have to show that $\Pi^S{}_{\mathbf{Q}}(q^+) = \Pi^S{}_{\mathbf{Q}}(q)$. For rewrite rule $G1$ the input to the outer most selection is: $inn = {q_1}^+ \times CrossBase(q_{sub_1}) \ldots \times CrossBase(q_{sub_n})$. Trivially $\Pi^S{}_{\mathbf{Q}}(inn) = \Pi^S{}_{\mathbf{Q}}(q)$, because none of the *CrossBases* evaluates to the empty set. Therefore, $\Pi^S{}_{\mathbf{Q}}(q^+) = \Pi^S{}_{\mathbf{Q}}(q)$ iff the selection condition C of the outermost selection of q^+ is fulfilled for ever tuple from ${Q_1}^+$ and for each condition ${C_{sub_i}}^+$ there is at least one tuple t_i from $CrossBase(Q_{sub_i})$ that fulfills condition ${C_{sub_i}}^+$. The first requirement is always meet, because every tuple from the result of q fulfills the selection condition C. The selection condition ${C_{sub_i}}^+$ filters out tuples from $CrossBase(q_{sub_i})$ that do not belong to the provenance of C_{sub_i}. If no tuples belong to the provenance of C_{sub_i}, the second *EXISTS* condition selects a tuple with all attributes set to ε. Thus, the second requirement is fulfilled too.
Case (G2):
The only difference between (G1) and (G2) is that ${q_1}^+$ is a projection and, therefore, we have to prove that $\Pi^S{}_A(q^+) = \Pi^S{}_A(q)$ holds. As for selection $\Pi^S{}_{\mathbf{Q}}(inn) = \Pi^S{}_{\mathbf{Q}}(q)$ trivially holds. Also each tuple from ${q_1}^+$ fulfills the selection condition of the outermost selection in q^+, because none of the attributes from $\mathbf{Q_1}$ are used in this condition. Finally, the ${C_{sub_i}}^+$ conditions are defined in the same way as for (G1). Hence, the same argument holds.

Witness List Preservation:
Recall from the proof of witness list preservation for the standard algebra operators that proving this property is equivalent to showing that the following equivalence holds for unary operators:

$$\begin{aligned}
&(t, v_1, \ldots, v_n) \in Q^+\\
\Leftrightarrow\\
&(w^x \in \mathscr{DD}(op(Q_1), t) \wedge w = <t'> \wedge (t', v_1, \ldots, v_n)^m \in {Q_1}^+)\\
&\vee (w^m \in \mathscr{DD}(op(Q_1), t) \wedge w = <\perp> \wedge m = 1 \wedge \forall i : v_i = (\varepsilon, \ldots, \varepsilon))
\end{aligned}$$

For queries with sublinks each witness list also contains tuples from the base relations accessed by the sublink queries. Let q be an query with sublink expressions C_{sub_1} to C_{sub_s} used in selection predicate C or projection list A. Then we have to show that the following equivalence holds:

$$\begin{aligned}
&(t, v_1, \ldots, v_n, v_{1_1}, \ldots, v_{1_n}, \ldots, v_{s_1}, \ldots, v_{s_n})^p \in Q^+\\
\Leftrightarrow\\
&w^p \in \mathscr{DD}(q, t) \wedge w[i]' = v_i
\end{aligned}$$

We prove this equivalence individually for both *Gen* strategy rewrite rules by induction over the number of sublink expressions in the selection predicate C respective projection expression list A. This is a correct approach, because we have demonstrated in chapter 3 that the provenance of a sublink used in an algebra expression q is independent of the existence of other sublinks in q.
Induction Start:
For an algebra expressions $q = \sigma_C(q_1)$ or $q = \Pi^{S/B}{}_A(q_1)$ with zero sublinks the witness list preservation property has been proven in the proof of theorem 4.1.
Induction Step:
Given that the witness list preservation property holds for algebra expressions $q = \sigma_C(q_1)$ with at most s sublink expressions in C we have to show that this property also holds for algebra expressions with $s+1$

sublink expressions. Given query $q = \sigma_C(q_1)$ with sublink expressions C_{sub_1} to $C_{sub_{s+1}}$ in selection predicate C we define two auxiliary queries $q_s(t)$ and $q_{s+1}(t)$ that are defined for a result rule t of q. $q_s(t)$ is derived from q by substituting $C_{sub_{s+1}}$ with the result of evaluating $C_{sub_{s+1}}$ for t ($C_{sub_{s+1}}(t)$). We denote the selection condition that is the result of this substitution as C_s. Note that $q_s(t)$ is a query with s sublink expressions and has the property that for a regular input tuple t it generates the same result as q (namely t) and for all sublink expressions C_{sub_i} the modified query $q_s(t)$ generates the same evaluation result as q. Thus, given the fact that the provenance of a sublink expression is independent of the existence of other sublinks in the query we know that if $w =< v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_{s+1}}, \dots, v_{n_{s+1}} >$ is a witness lists in $\mathscr{D}\mathscr{D}(q,t)$ then $w_1 =< v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_s}, \dots, v_{n_s} >$ is a witness list in $\mathscr{D}(q_s(t),t)$ and if $w_1 \in \mathscr{D}(q_s(t),t)$ if follows that $\exists v_{1_{s+1}}, \dots, v_{n_{s+1}} : w =< v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_{s+1}}, \dots, v_{n_{s+1}} >\in \mathscr{D}\mathscr{D}(q,t)$. From the induction hypothesis we know that in this case a tuple $(t, w_1[1]', \dots)$ is in $Q_s(t)^+$. The second auxiliary query $q_{s+1}(t)$ is derived by replacing every sublink expression except $C_{sub_{s+1}}$ with its evaluation over t and replacing q_1 with Q_1. For input tuple t, q_{s+1} produces the same result tuple t as q and agrees with q on the result of evaluating $C_{sub_{s+1}}$. From the independence of the provenance of sublinks from each other we can follow that iff w as defined above is in $\mathscr{D}\mathscr{D}(q,t)$ then $< t,t' >\in \mathscr{D}\mathscr{D}(q_{s+1},t)$ and $< v_{1_{s+1}}, \dots, v_{n_{s+1}} >\in \mathscr{D}\mathscr{D}(q_{sub_{s+1}},t')$. Using the facts stated about $q_s(t)$ and $q_{s+1}(t)$ we deduce that the following equivalence holds:

$$
\begin{aligned}
& w^{p\times r} =< v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_{s+1}}, \dots, v_{n_{s+1}} >^{p\times r}\in \mathscr{D}\mathscr{D}(q,t) \\
\Leftrightarrow & {w_1}^p =< v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_s}, \dots, v_{n_s} >^p\in \mathscr{D}\mathscr{D}(q_s(t),t) \\
& \wedge (({w_2}^x =< t,t' >^x\in \mathscr{D}\mathscr{D}(q_{s+1}(t),t) \wedge {w_3}^r =< v_{1_{s+1}}, \dots, v_{n_{s+1}} >^r\in \mathscr{D}\mathscr{D}(q_{sub_{s+1}}(t),t')) \vee \\
& ({w_2}^x =< t,\perp >^x\in \mathscr{D}\mathscr{D}(q_{s+1}(t),t) \wedge \forall i \in \{1_{s+1}, \dots, n_{s+1}\} : w[i] =\perp))
\end{aligned}
$$

Thus, we can transform the equivalence that we have to prove into:

$$
\begin{aligned}
& (t, v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_{s+1}}, \dots, v_{n_{s+1}})^{p\times r} \in Q^+ \\
\Leftrightarrow & {w_1}^p =< v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_s}, \dots, v_{n_s} >^p\in \mathscr{D}\mathscr{D}(q_s(t),t) \\
& \wedge (({w_2}^x =< t,t' >^x\in \mathscr{D}\mathscr{D}(q_{s+1}(t),t) \wedge {w_3}^r =< v_{1_{s+1}}, \dots, v_{n_{s+1}} >^r\in \mathscr{D}\mathscr{D}(q_{sub_{s+1}}(t),t')) \vee \\
& ({w_2}^x =< t,\perp >^x\in \mathscr{D}\mathscr{D}(q_{s+1}(t),t) \wedge \forall i \in \{1_{s+1}, \dots, n_{s+1}\} : w[i] =\perp \wedge r = 1))
\end{aligned}
$$

Substituting the induction hypothesis and using the witness list preservation property of algebra expressions without sublinks the ride hand side of this equivalence can be rewritten into:

$$
\begin{aligned}
& (t, v_1, \dots, v_{n_s})^p \in Q_s(t)^+ \wedge \\
& (({w_2}^x =< t,t' >^x\in \mathscr{D}\mathscr{D}(q_{s+1}(t),t) \wedge (t', v_{1_{s+1}}, \dots, v_{n_{s+1}})^r \in Q_{s+1}(t)^+) \vee \\
& ({w_2}^x =< t,\perp >^x\in \mathscr{D}\mathscr{D}(q_{s+1}(t),t) \wedge \forall i \in \{1_{s+1}, \dots, n_{s+1}\} : v_i = \varepsilon, \dots, \varepsilon))
\end{aligned}
$$

Hence, it remains to prove that the following equivalence holds:

$$
\begin{aligned}
& (t, v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_{s+1}}, \dots, v_{n_{s+1}})^{p\times r} \in Q^+ \\
\Leftrightarrow & (t, v_1, \dots, v_{n_s})^p \in Q_s(t)^+ \wedge \\
& (({w_2}^x =< t,t' >^x\in \mathscr{D}\mathscr{D}(q_{s+1}(t),t) \wedge (t', v_{1_{s+1}}, \dots, v_{n_{s+1}})^r \in Q_{s+1}(t)^+) \vee \\
& ({w_2}^x =< t,\perp >^x\in \mathscr{D}\mathscr{D}(q_{s+1}(t),t) \wedge \forall i \in \{1_{s+1}, \dots, n_{s+1}\} : v_i = \varepsilon, \dots, \varepsilon))
\end{aligned}
$$

We prove this equivalence in two steps. First, we prove that $(t, v_1, \dots, v_{n_s})^p \in {Q_s}^+$ follows from $(t, v_1, \dots, v_n, v_{1_1}, \dots, v_{1_n}, \dots, v_{1_{s+1}}, \dots, v_{n_{s+1}})^{p\times x} \in Q^+$ for some x. Second, we prove the remaining disjunction by separately handling the two disjuncts.
Applying rewrite rule (G1) to q and $q_s(t)$ we get:

$$
\begin{aligned}
q^+ =& \sigma_{C \wedge {C_{sub_1}}^+ \dots \wedge {C_{sub_{s+1}}}^+}({q_1}^+ \times CrossBase(q_{sub_1}) \dots \times CrossBase(q_{sub_{s+1}})) \\
q_s(t)^+ =& \sigma_{C_s \wedge {C_{sub_1}}^+ \dots \wedge {C_{sub_s}}^+}({q_1}^+ \times CrossBase(q_{sub_1}) \dots \times CrossBase(q_{sub_s}))
\end{aligned}
$$

If computed only for regular input tuple t using algebraic equivalences the relationship between q^+ and $q_s(t)^+$ can be expressed as:

$$q^+ = \sigma_{{C_{sub_{s+1}}}^+}(q_s(t)^+ \times CrossBase(q_{sub_{s+1}}))$$

Since $CrossBase(q_{sub_{s+1}})$ contains at least one tuple, the assumed equivalence holds if ${C_{sub_{s+1}}}^+$ if fulfilled for at least one tuple from the result of the cross product. The condition ${C_{sub_{s+1}}}^+$ is defined as:

$$\begin{aligned}&EXISTS\ (\sigma_{J_{sub_{s+1}} \wedge \mathscr{P}({q_{sub_{s+1}}}^+) =_n X}(\Pi^B{}_{\mathscr{P}({q_{sub_{s+1}}}^+) \rightarrow X}({q_{sub_{s+1}}}^+))\\ &\vee (\neg EXISTS(q_{sub_{s+1}}) \wedge \mathscr{P}({q_{sub_{s+1}}}^+) \text{ is } \varepsilon)\end{aligned}$$

If $Q_{sub} = \emptyset$ then the sub-condition $(\neg EXISTS(q_{sub}) \wedge \mathscr{P}({q_{sub}}^+) =_n null)$ is fulfilled for the tuple $(\varepsilon, \dots, \varepsilon)$ contained in $CrossBase(q_{sub_{s+1}})$. Otherwise ${Q_{sub_{s+1}}}^+$ contains at least one tuple $(t', u_1, \dots, u_m)$. Since $CrossBase(q_{sub_{s+1}})$ contains all possible relational representations of witness lists for $Q_{sub_{s+1}}$, for each tuple u from $q' = \Pi^B{}_{\mathscr{P}(q_{sub_{s+1}}) \rightarrow X}({q_{sub_{s+1}}}^+)$, there exists a tuple in $CrossBase(q_{sub_{s+1}})$ for which the condition $\mathscr{P}(q_{sub_{s+1}}) =_n X$ is fulfilled. It remains to show that at least one tuple from q' fulfills the selection predicate $J_{sub_{s+1}}$. For *EXISTS*- and scalar sublinks $J_{sub_{s+1}} = true$ and, therefore the condition is always fulfilled. For *ANY*-sublinks $J_{sub_{s+1}} = C'_{sub_{s+1}} \vee \neg C_{sub_{s+1}}$. If $C_{sub_{s+1}}$ evaluates to false then $J_{sub_{s+1}}$ is fulfilled. If $C_{sub_{s+1}}$ evaluates to true then $J_{sub_{s+1}} = e_{s+1}\ op_{s+1}\ t$ and we know from the semantics of the *ANY*-sublink expression that at least one tuple from q' fulfills this condition. The argument for *ALL*-sublinks is analog. The proof for rewrite rule (G2) uses the same argumentation.

It remains to show that all tuples from $CrossBase(q_{sub_{s+1}})$ that are attached to an original result tuple t by q^+ actually belong to the provenance. If $C_{sub_{s+1}}$ is an *EXISTS*-sublink or scalar sublink then all tuples from ${Q_{sub_{s+1}}}^+$ belong to the provenance and the following equivalence trivially holds:

$$\begin{aligned}&(t, v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_{s+1}}, \dots, v_{n_{s+1}})^{p \times r} \in Q^+\\ \Leftrightarrow &(t, v_1, \dots, v_{n_s})^p \in Q_s(t)^+ \wedge\\ &((w_2{}^x = < t, t' >^x \in \mathscr{DD}(q_{s+1}(t), t) \wedge (t', v_{1_{s+1}}, \dots, v_{n_{s+1}})^r \in Q_{s+1}(t)^+) \vee\\ &(w_2{}^x = < t, \perp >^x \in \mathscr{DD}(q_{s+1}(t), t) \wedge \forall i \in \{1_{s+1}, \dots, n_{s+1}\} : v_i = \varepsilon, \dots, \varepsilon) \wedge r = 1)\end{aligned}$$

If $C_{sub_{s+1}}$ is an *ANY*-sublinks that evaluates to false for t, then $J_{sub_{s+1}}$ evaluates to true for each tuple from ${Q_{sub_{s+1}}}^+$ which is correct because according to the compositional semantics all witness lists of form $< t', u >$ with $u \in Q_{sub_{s+1}}$ belong to the provenance of $q_{s+1}(t)$. If $C_{sub_{s+1}}$ evaluates to true then the condition $J_{sub_{s+1}} = e_{s+1}\ op_{s+1}\ t$ guarantees that only tuples from ${Q_{sub_{s+1}}}^{true}(t)$ belong to the result.

If $C_{sub_{s+1}}$ is an *ALL*-sublink that evaluates to true for t, then $J_{sub_{s+1}} = C_{sub_{s+1}} \vee \neg C'_{sub_{s+1}}$ evaluates to true for all tuples from ${Q_{sub_{s+1}}}^+$ which is the correct behaviour according to the compositional semantics of *PI-CS* provenance. If $C_{sub_{s+1}}$ evaluates to false, then $J_{sub_{s+1}} = \neg(e_{s+1}\ op_{s+1}\ t)$ filters out tuples that do not belong to ${Q_{sub_{s+1}}}^{false}(t)$.
We deduce that the equivalence holds:

$$\begin{aligned}&(t, v_1, \dots, v_n, v_{1_1}, \dots, v_{n_1}, \dots, v_{1_{s+1}}, \dots, v_{n_{s+1}})^{p \times r} \in Q^+\\ \Leftrightarrow &(t, v_1, \dots, v_{n_s})^p \in {Q_s}^+ \wedge\\ &((w_2{}^x = < t, t' >^x \in \mathscr{DD}(q_{s+1}(t), t) \wedge (t', v_{1_{s+1}}, \dots, v_{n_{s+1}})^r \in {Q_{s+1}}^+) \vee\\ &(w_2{}^x = < t, \perp >^x \in \mathscr{DD}(q_{s+1}(t), t) \wedge \forall i \in \{1_{s+1}, \dots, n_{s+1}\} : v_i = \varepsilon, \dots, \varepsilon) \wedge r = 1)\end{aligned}$$

We conclude that the *Gen* strategy rewrite rules fulfill the witness list preservation property. □

4.3.2 Specialized Rewrite Strategies

The *Gen* strategy presented in the last section is inefficient, because it uses a cross product that is restricted by complex sublink expressions and sublinks in general are hard to optimize. Hence, we developed strategies that are more efficient but are only applicable to algebra expressions that fulfill certain preconditions. These strategies utilize results from query optimization that are used to un-nest and de-correlated sublinks. Some of these results are directly applicable to provenance computations, while other had to be adapted to preserve the semantics of a provenance computation. We present two rewrite strategies (*Left* and *Move* strategy) for uncorrelated sublinks that apply the original sublink expressions in the rewritten query and use joins to add the provenance of the sublinks to the query result. Afterwards, we present the *Unn* and the *Unn-Not* strategies that un-nest uncorrelated sublinks. The *JA* and *Exists* strategies un-nest and de-correlate correlated sublinks.

Rewrite Rules

$$(\sigma_C(q_1))^+ = \Pi^B{}_{\mathbf{Q},\mathscr{P}(q_1{}^+),\mathscr{P}(q_{sub_1}{}^+),\ldots,\mathscr{P}(q_{sub_n}{}^+)}(\sigma_C(q_1{}^+ \text{⟕}_{J_{sub_1}} q_{sub_1}{}^+ \ldots \text{⟕}_{J_{sub_n}} q_{sub_n}{}^+)) \quad \textbf{(L1)}$$

$$(\Pi^{S/B}{}_A(q_1))^+ = \Pi^B{}_{A,\mathscr{P}(q_1{}^+),\mathscr{P}(q_{sub_1}{}^+),\ldots,\mathscr{P}(q_{sub_n}{}^+)}(q_1{}^+ \text{⟕}_{J_{sub_1}} q_{sub_1}{}^+ \ldots \text{⟕}_{J_{sub_n}} q_{sub_n}{}^+) \quad \textbf{(L2)}$$

Preconditions

1. All sublinks in C/A are uncorrelated

Figure 4.7: *Left* Strategy Rewrite Rules and Preconditions

4.3.2.1 Left Strategy

Uncorrelated sublinks enable us to directly join rewritten sublink queries with the rewritten regular input of a query. The *Left* strategy uses left outer joins to join the rewritten regular input of a query with the rewritten sublink queries. To join only tuples from a rewritten sublink query that belong to the provenance of the sublink we can utilize the join condition J_{sub} from the last section. We have to use outer joins to produce correct results for empty sublink queries. The rewrite rules **(L1)** and **(L2)** of the *Left* strategy are presented in Figure 4.7.

Example 4.5. *For instance, consider example query q_a presented below. Applying rewrite rule (L1) generates the rewritten query $q_a{}^+$ as shown in this figure.*

$$\begin{aligned}
q_a &= \sigma_{a\,=\,ALL\,(S)}(R)\\
q_a{}^+ &= \Pi^B{}_{a,b,\mathscr{N}(a),\mathscr{N}(b),\mathscr{N}(c)}(\sigma_{C_{sub}}(\Pi^B{}_{a,b,a\rightarrow\mathscr{N}(a),b\rightarrow\mathscr{N}(b)}(R)) \text{⟕}_{C_{sub} \vee \neg C_{sub}'} \Pi^B{}_{c,c\rightarrow\mathscr{N}(c)}(S))\\
C_{sub} &= (a\ =\ ALL\ (S))\\
C_{sub}' &= (a = c)
\end{aligned}$$

Theorem 4.3 (Correctness and Completeness of the Left Strategy)**.** *For a query q with sublinks, the provenance computed by the rewritten query q^+ according to rules (L1) and (L2) is the provenance derived according to definition 3.12:*

$$Q^+ = Q^{PI}$$

Proof.
Similar to the proofs presented beforehand we prove the correctness and completeness of the *Left* strategy by demonstrating that it fulfills the *result preservation* and *witness list preservation* properties.

Result Preservation:

From the semantics of left outer join we know that the left join never produces an empty result for a non empty left hand input. Therefore, $\Pi^S{}_{\mathbf{Q}}(q^+) = \Pi^S{}_{\mathbf{Q}}(q)$ holds.

Witness List Preservation:

w.l.o.g. we assume that none of the sublink queries contains sublinks themselves. Since each rewritten sublink algebra expression $q_{sub_i}{}^+$ is an algebra expression without sublinks it remains to show that J_{sub_i} guarantees that witness list representations are filtered out from $Q_{sub_i}{}^+$ if they do not belong to the provenance of q^+. For *EXISTS*- and scalar sublinks all tuples from $Q_{sub_i}{}^+$ belong to the provenance. In this case $J_{sub_i} = true$ and none of these tuples is excluded. For *ANY*- and *ALL*- sublinks we have shown in the proof of theorem 4.2, that J_{sub_i} only filters out tuples that do not belong to the provenance. □

Rewrite Rules

$$q^+ = (\sigma_C(q_1))^+ = \Pi^B{}_{\mathbf{Q},\mathscr{P}(q^+)}(\sigma_{C'}(\Pi^B{}_{\mathbf{Q},\mathscr{P}(q_1{}^+),C_{sub_1}\rightarrow C_1,\dots,C_{sub_m}\rightarrow C_m}(q_1{}^+)\text{⟕}_{J_{sub_1}{}'}\, q_{sub_1}{}^+ \dots \text{⟕}_{J_{sub_n}{}'}\, q_{sub_n}{}^+)) \quad \textbf{(M1)}$$

$$q^+ = (\Pi^{S/B}{}_A(q_1))^+ = \Pi^B{}_{A'',\mathscr{P}(q^+)}(\Pi^{S/B}{}_{A'}(q_1{}^+)\text{⟕}_{J_{sub_1}{}'}\, q_{sub_1}{}^+ \dots \text{⟕}_{J_{sub_n}{}'}\, q_{sub_n}{}^+) \quad \textbf{(M2)}$$

Preconditions

1. All sublinks in C/A are uncorrelated

Figure 4.8: *Move* Strategy Rewrite Rules and Preconditions

4.3.2.2 Move Strategy

The rewrite rules of the *Left* strategy have the disadvantage that the sublink C_{sub} is duplicated in the condition J_{sub}. This is unproblematic if the query optimizer is aware of the duplication and computes C_{sub} only once. If the duplication is not recognized, query performance will suffer dramatically. To circumvent this potential problem, we introduce the *Move* strategy that uses modified versions of the *Left* rewrite rules. These rewrite rules have been modified to move selection sublinks into a projection. Thus, we are able to use the results of a sublink in both the selection and the join condition J_{sub}. Rewrite rule **(M1)** uses this strategy for sublinks in selection. Selection condition C' is a modified version of selection condition C where all sublinks are replaced by the new projection attributes $C_1,\dots,C_m$. Each C_i stores the result of one sublink expression C_{sub_i}. Similar $J_{sub}{}'$ is the join condition J_{sub} with the sublink expressions replaced by the new projection attributes.

Rewrite rule **(M2)** is the modified rewrite rule for projection sublinks. A new inner projection on A' is used to project on all expressions from A that do not contain a sublink, on the sublinks used in A as new attributes $C_1,\dots,C_m$, and on the provenance attributes of $q_1{}^+$. The modified outer projection on A'' includes all expressions from A' that do not contain a sublink and all expressions from A that contain sublinks with the sublinks replaced by the new attributes from A'. For each projection expression containing sublinks, the sublinks are replaced with the new $C_1,\dots,C_m$ attributes.

Example 4.6. *As an example consider the queries q_a and q_b, and their rewritten versions ($q_a{}^+$ and $q_b{}^+$) presented below.*

$$\begin{aligned}
q_a &= \sigma_{a\,=\,ALL\,(S)}(R)\\
q_a{}^+ &= \Pi^B{}_{a,b,\mathscr{N}(a),\mathscr{N}(b),\mathscr{N}(c)}(\sigma_{C_1}(\Pi^B{}_{a,b,a\rightarrow\mathscr{N}(a),b\rightarrow\mathscr{N}(b),(a\,=\,ALL\,(S))\rightarrow C_1}(R)\text{⟕}_{C_1\vee\neg(a=c)}\Pi^B{}_{c,c\rightarrow\mathscr{N}(c)}(S)))\\
q_b &= \Pi^B{}_{a,S}(R)\\
q_b{}^+ &= \Pi^B{}_{a,C_1,\mathscr{N}(a),\mathscr{N}(b),\mathscr{N}(c)}(\Pi^B{}_{a,S\rightarrow C_1}(R)\text{⟕}_{true}(S))
\end{aligned}$$

Theorem 4.4 (Correctness and Completeness of the Move Strategy)**.** *For a query q with sublinks, the provenance computed by the rewritten query q^+ according to rules (M1) and (M2) is the provenance derived according to definition 3.12:*

$$Q^+ = Q^{PI}$$

Proof.
For an algebra expression q with sublinks let q^M be the result of applying the *Move* strategy rewrite rules to q and q^L be the result of applying the *Left* strategy rewrite rules. q^M can be transformed into q^L using algebraic equivalence rules (Factoring out a sub-condition in a selection predicate into a projection expression). For q^L we have already proven that $Q^L = Q^{PI}$ holds. Hence, $Q^M = Q^{PI}$ holds. □

Rewrite Rules

$$q^+ = (\sigma_C(q_1))^+ = \Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^+)}(\sigma_{C^+}({q_1}^+ \bowtie_{{C_1}^+} {q_{sub_1}}^+ \dots \bowtie_{{C_m}^+} {q_{sub_m}}^+)) \qquad \textbf{(U1)}$$

Preconditions

1. All sublinks in C are uncorrelated.
2. All sublinks are *ANY*- or *EXISTS*-sublinks.
3. Each sublink C_{sub} is either the only expression in selection condition C or if C is represented as an expression tree, all ancestors of C_{sub} are logical conjunctions ($\wedge$).

Figure 4.9: *Unn* Strategy Rewrite Rules and Preconditions

4.3.2.3 Unn Strategy

Provenance computation can benefit from the de-correlation and un-nesting techniques developed for query optimization [CB07, EGLGJ07, AB03, Cha98, Mur92, Day83, Kim82]. Besides the fact that these approaches are normally only suitable for specific types of sublink queries, the performance gain can be significant. We will demonstrate in chapter 6 that the performance gain is even higher for provenance queries, because most techniques transform sublinks into joins for which the provenance rewrite rules are very efficient. In addition, the complex join conditions containing sublinks used in the *Left* and *Move* strategies can be omitted.

The first strategy using un-nesting we introduce is the *Unn* strategy. The *Unn* strategy completely un-nests sublinks by transforming them into standard join operations. In the rewritten result of the *Unn* rewrite rule **(U1)** the provenance computation for the sublink query and the evaluation of the sublink itself are performed by a single join. This is similar to one of the un-nesting rule described in [Kim82] (the authors call this type of sublink *type N*). In contrast to this approach we do not need to apply a duplicate removing projection in the provenance computation, because the produced duplicates of the original result tuples are required to fit in all witness list representations.

Rewrite rule *(U1)* transforms a selection with uncorrelated sublinks $C_{sub_1}, \dots, C_{sub_n}$ in selection condition C into joins between the rewritten regular input and the rewritten sublink queries. Here each ${C_i}^+$ denotes a modified version of C where C_{sub_i} is replaced with ${C_{sub_i}}'$ as defined for the *Gen* strategy and all other sublinks expressions are substituted with *true*. The selection condition C^+ is derived from the original selection condition C by replacing each sublink expression with *true*. We replace the sublinks with *true*, because the preconditions of the strategy guaranty that all sublinks evaluate to *true* if C is fulfilled.

The *Unn* strategy has several preconditions. First, all sublinks in C have to be uncorrelated *ANY*-, *EXISTS*-, or scalar sublinks (**1** and **2**). Second, each sublink expression C_{sub_i} is either the only expression in C or all ancestors of C_{sub} in the expression tree for C are boolean conjunctions (**3**). For instance, $C_1 = (a = 3 \vee EXISTS\ (S))$ does not fulfill this precondition, but $C_2 = (a = 3 \wedge EXISTS\ (S))$ does. All this preconditions are needed to guaranty that the rewrite does not change the semantics of the sublink expressions. E.g., disjunctions may lead to false positives in the provenance and *ALL*-sublinks cannot be expressed directly as joins.

Example 4.7. *As an example application of rewrite rule (U1) consider query q_a presented below. The sublink expression is checked by the join condition $a = c$. This condition also guarantees that all tuples from ${Q_{sub}}^+$ that fulfill the join condition are extended versions of tuples from ${Q_{sub}}^{true}$. These are exactly the tuples that contain relational representations of witness lists that are in the provenance of q_a according to the compositional semantics of ANY-sublinks.*

$$q_a = \sigma_{a\,=\,ANY\ (S)}(R)$$
$${q_a}^+ = \Pi^B{}_{a,b,\mathscr{N}(a),\mathscr{N}(b),\mathscr{N}(c)}(\Pi^B{}_{a,b,a \rightarrow \mathscr{N}(a),b \rightarrow \mathscr{N}(b)}(R) \bowtie_{(a=c)} \Pi^B{}_{c,c \rightarrow \mathscr{N}(c)}(S))$$

Theorem 4.5 (Correctness and Completeness of the Unn Strategy)**.** *For a query q with sublinks, the provenance computed by the rewritten query q^+ according to rewrite rule (U1) is the provenance derived according to definition 3.12:*

$$Q^+ = Q^{PI}$$

Proof.

Result Preservation:

The result preservation property is fulfilled if for all regular input tuples t for which C is fulfilled, each join condition C_i evaluate to true for at least one tuple from the right hand side input, because the preconditions of the *Unn* strategy guaranty that C cannot be true if one of the sublink expressions evaluates to *false*. If C_{sub_i} is an *EXISTS*-sublink then it evaluates to true, iff Q_{sub_i} contains at least one tuple. C_i is defined as $true$ and, thus a tuple (t,u) from the rewritten regular input is joined successfully iff Q_{sub_i} is not the empty set. If C_{sub_i} is an *ANY*-sublink then C_{sub_i} evaluates to true if ${Q_{sub_i}}^{true}(t)$ contains at least one tuple. For all tuples from ${Q_{sub_i}}^{true}(t)$ the join condition $C_i = e_i \; op_i \; t$ is fulfilled. It follows that q^+ generated by the *Unn* strategy fulfills the result preservation property.

Witness List Preservation: We have to show that only witness list representations from ${q_{sub_i}}^+$ that belong to the provenance are propagated. If C_{sub_i} is an *EXISTS*-sublink all witness list representations from ${q_{sub_i}}^+$ belong to the provenance and the join condition $C_i = true$ guarantees that all these witness list representations appear in the result of q^+. If C_{sub_i} is an *ANY*-sublink then only witness list representations attached to tuples in ${Q_{sub_i}}^{true}(t)$ belong to the provenance (Recall that the preconditions of the *Unn* strategy require all sublink conditions to evaluate to true). In this case the join condition $C_i = e_i \; op_i \; t$ filters out tuples that do not belong to ${Q_{sub_i}}^{true}(t)$. □

Rewrite Rules

$$(\sigma_C(q_1))^+ = \Pi^B{}_{\mathbf{Q},\mathscr{P}(q^+)}(\sigma_{C^+,}(q_1{}^+ \text{⟕}_{C_1{}^+} \Pi^B{}_{\mathbf{Q_{sub_1}},1\rightarrow dummy_1}(q_{sub_1}) \text{⟕}_{true} \Pi^B{}_{\mathscr{P}(q_{sub_1}{}^+)}(q_{sub_1}{}^+) \ldots \text{⟕}_{C_m{}^+} \Pi^B{}_{\mathbf{Q_{sub_m}},1\rightarrow dummy_m}(q_{sub_m}) \text{⟕}_{true} \Pi^B{}_{\mathscr{P}(q_{sub_m}{}^+)}(q_{sub_m}{}^+))) \qquad \textbf{(N1)}$$

$$(\sigma_C(q_1))^+ = \Pi^B{}_{\mathbf{Q},\mathscr{P}(q^+)}(\sigma_{C^+,}(q_1{}^+ \text{⟕}_{C_1{}^+} \Pi^B{}_{\mathbf{Q_{sub_1}},1\rightarrow dummy_1}(q_{sub_1}) \times null(\mathscr{P}(q_{sub_1}{}^+)) \ldots \text{⟕}_{C_m{}^+} \Pi^B{}_{\mathbf{Q_{sub_m}},1\rightarrow dummy_m}(q_{sub_m}) \times null(\mathscr{P}(q_{sub_m}{}^+)))) \qquad \textbf{(N2)}$$

Preconditions

1. All sublinks in C are uncorrelated
2. Each sublink C_{sub} is negated and if C is represented as an expression tree, all ancestors of C_{sub} except for its direct parent are logical conjunctions ($\wedge$).
3. **(N1)**: All sublinks are *ANY*-sublinks
4. **(N2)**: All sublinks are *EXISTS*-sublinks

Figure 4.10: *Unn-Not* Strategy Rewrite Rules and Preconditions

4.3.2.4 Unn-Not Strategy

The basic idea of the *Unn-Not* strategy is to transform the provenance computation of a sublink into two joins; one that simulates the sublink and a second one that propagates provenance information. This strategy rewrites negated *ANY*- and *EXISTS*-sublinks by using the left outer join operator to simulate negation. The original sublink query is joined with each sublink C_{sub_i} on condition $C_i{}^+ = C_{sub_i}{}'$ (see section 4.3.1 for the definition of $C_{sub_i}{}'$). If for a regular input tuple t none of the tuples from Q_{sub} fulfill this condition, then the left join will produce null values for the attributes from $\mathbf{Q_{sub_i}}$ which is checked by an additional selection condition C^+ that is applied to the result of the join. In this case the negated sublink expression would evaluate to true. C^+ is derived from C by replacing each sublink C_{sub_i} in C with $dummy_i$ is ε. The additional constant valued attributes $dummy_i$ are needed to cope with tuples from the sublink query that contain ε values. Such tuples are problematic, because we cannot distinguish between them and a ε produced by the left outer join. The left join of $q_1{}^+$ with q_{sub_i} only checks the negated sublink condition. To propagate provenance information an additional join with $q_{sub_i}{}^+$ is added to the rewritten query. Figure 4.10 presents **(N1)**, the *Unn-Not* strategy rewrite rule for *ANY*-sublinks. *EXISTS*-sublinks evaluate to false, iff the sublink query returns the empty set. Therefore, the provenance propagation is implemented in the rewrite rules for *EXISTS*-sublinks as a cross product with *null* values (rule **(N2)**). The *Unn-Not* strategy cannot be applied to correlated sublinks, *ALL*-sublinks, or sublinks that are not guaranteed to be *reqfalse* for all regular input tuples.

Example 4.8. *As an example for the application of this rewrite strategy consider query q_a presented below. In the rewritten query $q_a{}^+$ the selection condition dummy is ε guarantees that there are no tuples in S that fulfill the condition $a = c$ of the ANY-sublink. For regular input tuples for which this condition is fulfilled all witness list representations from $q_{sub}{}^+$ are joined as required by the compositional semantics of an ANY-sublink.*

$$q_a = \sigma_{\neg(a\,=\,ANY\ (S))}(R)$$

$$q_a{}^+ = \Pi^B{}_{a,b,\mathscr{N}(a),\mathscr{N}(b),\mathscr{N}(c)}(\sigma_{dummy\ is\ \varepsilon}(\Pi^B{}_{a,b,a\rightarrow\mathscr{N}(a),b\rightarrow\mathscr{N}(b)}(R) \text{⟕}_{a=c} \Pi^B{}_{c,1\rightarrow dummy}(S) \text{⟕}_{true} \Pi^B{}_{\mathscr{N}(c)}(\Pi^B{}_{c,c\rightarrow\mathscr{N}(c)}(S))))$$

Theorem 4.6 (Correctness and Completeness of the Unn-Not Strategy)**.** *For a query q with sublinks, the provenance computed by the rewritten query q^+ according to rewrite rule (N1) is the provenance derived according to definition 3.12:*

$$Q^+ = Q^{PI}$$

Proof.

Result Preservation: All tuples from ${q_1}^+$ are propagated to the result of the joins in the rewritten query, because all sublink queries and rewritten sublink queries are joined with the rewritten regular input. The preconditions of the *Unn-Not* strategy guarantee that the selection predicate C can only evaluate to true iff all sublink expressions evaluates to false, because all sublink expressions are negated and used in conjunctions. If C_{sub_i} is an *EXISTS*-sublink then Q_{sub_i} has to be the empty set for C_{sub_i} to evaluate to false. In this case $dummy_i$ is ε and the modified selection predicate C^+ evaluates to the same result as C in the original query. If C_{sub_i} is an *ANY*-sublink ${C_i}^+ = e_i \; op_i \; t$ evaluates to false for each tuple from Q_{sub_i} and, thus, $dummy_i$ is ε is fulfilled.

Witness List Preservation:

According to the compositional semantics of *ANY*- and *EXISTS*-sublinks, all witness list representations from ${Q_{sub_i}}^+$ belong to the provenance if the sublink expression evaluates to false. In the rewritten query all tuples from ${Q_{sub_i}}^+$ are propagated to the result, because they are joined on condition $true$ using a left outer join.

□

Rewrite Rules

$$(\sigma_C(q_1))^+ = \Pi^B{}_{\mathbf{Q},\mathscr{P}(q^+)}(\sigma_{C^+}(q_1{}^+ \mathbin{⟕}_{C_1} x_{sub_1}{}^+ \ldots \mathbin{⟕}_{C_m} x_{sub_m}{}^+)) \qquad \textbf{(J1)}$$

Preconditions

1. Each sublink query has as outermost operator an aggregation without group-by attributes. I.e., it is of the form $\alpha_{agg}(q')$.

2. All correlations used in sublink queries are used in selection conditions C and are of form $reg = corr$ where reg is a correlated attribute from the regular input and $corr$ is an attribute from the sublink query. $reg = corr$ is either the only expression in C or all of its ancestors are logical conjunctions. Furthermore, the selection condition C has to be applied by a selection that is not below any set operations or aggregations (except the outermost one).

3. For *ANY* sublinks the expression e is not allowed to contain other sublinks.

4. Each sublink C_{sub} is either the only expression in selection condition C or all ancestors of C_{sub} in C are logical conjunctions.

5. All sublinks are correlated *ANY*- or *scalar* sublinks.

Figure 4.11: *JA* Strategy Rewrite Rules and Preconditions

4.3.2.5 JA Strategy

The *JA strategy* is a modified version of the rewrite for so called *JA* queries from [Kim82]. The rewrite rule of this strategy is applicable to queries with correlated sublink expressions ($C_{sub_1}, \ldots, C_{sub_m}$) that have an aggregation without group-by attributes as their outermost operator. The rationale behind the strategy is that the correlation can be simulated as a group-by and join. Thus, de-correlating the sublink expression. For this strategy to be applicable all correlations have to be of form $reg = corr$ for an regular input attribute reg and a sublink query attribute $corr$ and each of this correlations has to be used in a selection condition C (for further preconditions see Figure 4.11). The rewrite rule **(J1)** joins a modified version x_{sub_i} of each sublink query q_{sub_i} with the rewritten regular input. For a sublink query $q_{sub_i} = \alpha_{agg}(q_x)$, the modified version x_{sub_i} is derived from q_{sub_i} by removing all correlation expressions (replacing them with $true$) and for each correlation expression adding $corr$ to the list of group-by attributes. Furthermore, we add a new outermost projection that adds a constant valued attribute $dummy_i$. x_{sub} is joined with the rewritten regular input on a condition C_i. C_i contains all the correlation expressions from q_{sub} with the attributes from q_{sub} replaced by the new group-by attributes from x_{sub}. The output of the left joins is then filtered by a selection condition C^+ that is equal to the original selection condition C but each sublink C_{sub_i} in C is replaced by an expression $C_i{}'$. The definition of $C_i{}'$ depends on the type of the sublink and the aggregation function that is applied in q_{sub_i}. If $C_{sub_i} = e_i\ op_i\ ANY\ q_{sub_i}$ is an *ANY*-sublink that uses the aggregation function *count*, then

$$C_i{}' = e_i\ op_i\ agg_i \vee (dummy_i \text{ is } \varepsilon \wedge e_i\ op_i\ 0)$$

If an aggregation function different from *count* is used, then

$$C_i{}' = e_i\ op_i\ agg_i \vee (dummy_i \text{ is } \varepsilon \wedge e_i\ op_i\ \varepsilon)$$

If $C_{sub_i} = q_{sub_i}$ is a scalar sublink that uses aggregation function *count*, then

$$C_i{}' =\ if\ (dummy_i \text{ is } \varepsilon)\ then\ (0)\ else\ (agg_i)$$

If $C_{sub_i} = q_{sub_i}$ is a scalar sublink that uses an aggregation function other than *count*, then

$$C_i{}' = \mathit{if}\ (\mathit{dummy}_i \text{ is } \varepsilon)\ \mathit{then}\ (\varepsilon)\ \mathit{else}\ (\mathit{agg}_i)$$

The constant valued attribute *dummy* is needed to prevent the so-called *count* bug (see, e.g., [Mur92]). The *count* bug is an error in the original de-correlation rewrite rule for *JA* queries presented in [Kim82]. If the input to the aggregation used in a sublink is the empty set, the aggregation will result in a single tuple containing either value 0 (if the aggregation function is *count*) or ε (for all other aggregation functions). In the de-correlated version of the sublink query no tuple will be returned, because aggregation only outputs tuples for existing groups. The *dummy* attribute is used to circumvent this problem. If the de-correlated sublink query returns no tuples for a specific group-by attribute then the *dummy* attribute will be ε, and the second part of $C_i{}'$ simulates the sublink expression with the aggregation function value that would have been generated by the original sublink query.

Example 4.9. *As an example for the application of rule (J1) consider query q_a presented below. The sublink query q_{sub} contains a single correlation expression $d = b$. This expression is transformed into the group-by on d and the join predicate $d = b$.*

$$\begin{aligned}
q_a &= \sigma_{a\,=\,ANY\ (q_{sub})}(R)\\
q_{sub} &= \alpha_{sum(c)}(\sigma_{d=b}(S))\\
q_a{}^{+} &= \Pi^{B}{}_{a,b,\mathcal{N}(a),\mathcal{N}(b),\mathcal{N}(c)}(\sigma_{a=sum(c)\vee(dummy\ is\ \varepsilon\wedge a=\varepsilon)}(\Pi^{B}{}_{a,b,a\rightarrow\mathcal{N}(a),b\rightarrow\mathcal{N}(b)}(R) \mathbin{⟕}_{d=b} x_{sub}{}^{+}))\\
x_{sub} &= \Pi^{B}{}_{d,sum(c),1\rightarrow dummy}(\alpha_{d,sum(c)}(S))\\
x_{sub}{}^{+} &= \Pi^{B}{}_{d,sum(c),1\rightarrow dummy,\mathcal{N}(c),\mathcal{N}(d)}(\alpha_{d,sum(c)}(S) \mathbin{⟕}_{d=_{n}x} \Pi^{B}{}_{d\rightarrow x,c\rightarrow\mathcal{N}(c),d\rightarrow\mathcal{N}(d)}(S))
\end{aligned}$$

Theorem 4.7 (Correctness and Completeness of the JA Strategy)**.** *For a query q with sublinks, the provenance computed by the rewritten query q^{+} according to rules (J1) is the provenance derived according to definition 3.12:*

$$Q^{+} = Q^{PI}$$

Proof.

Result Preservation:

The result preservation property is fulfilled for a regular input tuple t if the join conditions C_i are only fulfilled for tuples from $Q_{sub_i}(t)$ and if the modified selection predicate C^{+} evaluates to the same result over t as the original selection predicate C. For a tuple $t \in Q$ the group-by used in x_{sub_i} in combination with the join predicate on the correlation expressions filters out tuples from Q_{sub_i} that do not belong to $Q_{sub_i}(t)$. This has been demonstrated in [Kim82]. For *scalar* sublinks the expression C_i' applied in C^{+} is bound to evaluate to the same result as C_{sub_i}, because it just references agg_i, the result of C_{sub_i} and handles the count bug using the constant valued attribute $dummy_i$. $Q_{sub_i}(t)$ is an aggregation without group-by, which means it returns exactly one result tuple. Therefore, if C_{sub_i} is an *ANY*-sublink then C_{sub_i} is equivalent to the expression $e_i\ op_i\ agg_i$ which is used in C_i' (in addition to the condition that handles the count bug).

Witness List Preservation:

For *scalar*-sublinks all witness list representations from $Q_{sub_i}{}^{+}(t)$ are propagated to the result because the join condition C_i only filters out tuples that do not belong to $Q_{sub_i}{}^{+}(t)$ by applying the correlation expressions on the group-by attributes. For *ANY*-sublink expressions, the additional condition $e_i\ op_i\ t$ in C_i' filters out witness list representations that are not attached to tuples from $Q_{sub_i}{}^{true}(t)$. This is correct, because the preconditions of the *JA* strategy guarantee that all *ANY*-sublinks are required to return true.

□

Rewrite Rules

$$q^+ = (\sigma_C(q_1))^+ = \Pi^B{}_{\mathbf{Q},\mathscr{P}(q^+)}(\sigma_{C^+}(q_1{}^+ \bowtie_{C_1} q_{sub_1}{}^E \dots \bowtie_{C_m} q_{sub_m}{}^E)) \qquad \textbf{(E1)}$$

Preconditions

1. All sublinks are correlated *EXISTS*- or scalar sublinks.
2. The outermost operator of each sublink query is not an aggregation or set operation.
3. Each correlation used in a sublink query is used in a selection conditions C and is of form $reg = corr$ where reg is an correlated attribute from the regular input and $corr$ is an attribute from the sublink query. $reg = corr$ is either the only expression in C or all of its ancestors are logical conjunctions. Furthermore, the selection condition C has to be applied by a selection that is not below any set operations or aggregations.
4. Each sublink C_{sub} is either the only expression in selection condition C or all ancestors of C_{sub} in C are logical conjunctions.

Figure 4.12: *Exists* Strategy Rewrite Rules and Preconditions

4.3.2.6 Exists Strategy

The *Exists* strategy un-nests and de-correlates correlated *EXISTS*-sublinks by transforming the sublink expression into a simple join. The rationale behind this strategy is that the *EXISTS*-sublink expression evaluates to true if the sublink query returns at least one tuple and that all witness list representations generated by the rewritten sublink query belong to the provenance. This can be simulated by replacing the sublink expression with a join over the correlated attributes. In rewrite rule **(E1)** (Figure 4.12) the expression $q_{sub_i}{}^E$ is the rewritten sublink query with $true$ substituted for the correlation expressions and each attribute a that is used in a correlation expression added to an additional outermost projection. The join condition C_i is a conjunction of expressions for each correlation expression in q_{sub_i}. Each of the transformed correlation expressions is of the form $reg_j = x_j$ where reg_j is the regular input attribute used in the correlation expression and x_j is the correlated attributed from $\mathbf{Q_{sub}}$ used in the correlation expression. The modified selection condition C^+ is derived from C be replacing the sublink expressions with $true$.

Example 4.10. *As an example for the exists strategy consider query q_a presented below. The modified selection condition is just the constant true, because the sublink is the only expression in C.*

$$q_a = \sigma_{EXISTS\,(\sigma_{c=a}(S))}(R)$$
$$q_a{}^+ = \Pi^B{}_{a,b,\mathcal{N}(a),\mathcal{N}(b),\mathcal{N}(c)}(\sigma_{true}(\Pi^B{}_{a,b,a\to\mathcal{N}(a),b\to\mathcal{N}(b)}(R) \bowtie_{a=x} \Pi^B{}_{c,c\to x,c\to\mathcal{N}(c)}(S)))$$

Theorem 4.8 (Correctness and Completeness of the Count Strategy)**.** *For a query q with sublinks, the provenance computed by the rewritten query q^+ according to rewrite rule (E1) is the provenance derived according to definition 3.12:*

$$Q^+ = Q^{PI}$$

Proof.

<u>**Result Preservation**</u>:

All sublinks expressions in q are *EXISTS*-sublinks and because of the precondition of the *Exists* strategy have to evaluate to true for all tuples t in the result of q. In the modified selection predicate C^+ all sublink expressions are replaced with true. The sublink expressions are simulated by using joins on the correlation expressions. A tuple from ${q_1}^+$ has at least one join partner in ${q_{sub_1}}^E$ iff $Q_{sub_i}(t)$ contains at least one tuple. Therefore, the result preservation property is fulfilled.

Witness List Preservation:

All witness list representations that are attached to a tuple t' from $Q_{sub_i}(t)$ belong to the provenance of t. The correlation expressions in the join condition filter out all witness list representations that are not attached to these tuples. Hence, the witness list preservation property holds. □

4.4 Rewrite Rules for Copy-Contribution-Semantics

In this section we present rewrite rules for the *C-CS* types used in *Perm*. Recall that the *C-CS* types were defined as restrictions of *PI-CS* modeled by the so-called *copy maps*. Therefore, we can compute provenance according to this *CS* types by using the rewrite rules for *PI-CS* and adding filter conditions that implement these restrictions. We first present the rewrite rules that implement this approach. Afterwards, we demonstrate that according to the copy map definitions the *C-CS* provenance of some types of algebra expressions is independent of the database instance. It follows, that for these algebra expressions it is not necessary to apply the possibly complex *PI-CS* rewrites, but instead the provenance computation can be implemented as additional projections. The relational representation of provenance according to one of the *C-CS* types is defined as for *PI-CS*, because these *CS* types also use witness lists to represent provenance information:

Definition 4.3 (Relational Representation of *C-CS* Provenance)**.**

$$Q^{\mathscr{CD}/\mathscr{CT}/\mathscr{PD}/\mathscr{PT}} = \{(t, w[1]', \ldots, w[n]')^m \mid t^p \in Q \wedge w^m \in \mathscr{CD}(q,t)/\mathscr{CT}(q,t)/\mathscr{PD}(q,t)/\mathscr{PT}(q,t)\}$$

To integrate the restrictions imposed by the copy-maps into the rewrites we add additional attributes that store intermediate versions of copy maps for parts of a rewritten query. $\mathscr{C}(q)$ denotes this list of copy map attributes for a query q. Let $\mathscr{B}(q)$ be the list of all attributes from the base relations accessed by q. $\mathscr{C}(q)$ contains one attribute for each attribute a in $\mathscr{B}(q)$ that is used to store the part of the copy map of q that corresponds to that attribute ($\mathscr{CM}(q,a,w,t)$). The names of the attributes in $\mathscr{C}(q)$ are derived using a function $\mathscr{C}$ that generates unique attribute names for each attribute in $\mathscr{C}(q)$ (similar to the provenance attribute naming function $\mathscr{N}$). We do not discuss the concrete naming scheme here, but instead use $\mathscr{C}(a)$ to indicate the new name generated for attribute a. Each attribute $\mathscr{C}(a)$ stores a set of attribute identifiers. Therefore, we assume the existence of a data type for these sets and standard set operations for this data type. We define ε as an alternative representation of the empty set, because this will simplify the definition of the rewrite rules.

Example 4.11. *For example, if q is $R \bowtie_C S$ over relations R and S with schemas $\mathbf{R} = (a)$ and $\mathbf{S} = (b,c)$ then:*

$$\mathscr{C}(q) = \mathscr{C}(a), \mathscr{C}(b), \mathscr{C}(c)$$

Using $\mathscr{C}(q)$ we define expressions $CM(q)$ (called copy expressions) that are used to simulate an incremental computation of the copy maps. E.g., for projection the result of the copy map may depend on the attribute values of an tuple, thus, attributes have to be included or excluded from $\mathscr{C}(q)$ if the currently processed tuple fulfills certain conditions. Figure 4.13 presents the definition of CM. Like the rewrite rules CM is defined recursively over the structure of an algebra expression. We first discuss CM in detail before presenting the *C-CS* meta-operators and the rewrite rules that implement them. Two version of CM are introduced. One that implements the *direct-copy* copy map and one that implements the *transitive-copy* copy map (see 3.3.1).

For the *direct-copy* version of CM the copy expressions of a base relation access initialize each $\mathscr{C}(a)$ attribute with a singleton set containing a. The copy expression for selection, join operators, union, and intersection simply pass on the copy map attributes from their input algebra expression(s). For projection the conditional inclusion of attributes is modeled as a union $\bigcup_{x \in A} \mathscr{C}^*(a_i, x)$ of conditional attribute inclusions $\mathscr{C}^*(a_i, x)$. E.g., attribute z is included in the copy map of an attribute a_i, if an expression $y \rightarrow z$ is one of the projection expressions in A and the copy map of a_i for the input of the projection includes y.

This is modeled as $if\ (y \in \mathscr{C}(a_i))\ then\ (z)\ else\ (\emptyset)$. The CM for aggregation intersects each $\mathscr{C}(a_i)$ set with the set G of group-by attributes. The *transitivity-copy* version of CM agrees with the *direct-copy* version on all algebra operators except for the join operators and selection. For these operators the inclusion of an attributes due to a sub-condition $x = y$ in C is modeled as $if\ ((x = y) \wedge x \in \mathscr{C}(a))\ then\ (y)\ else\ (\emptyset)$.

The inclusion of a part of a *PI-CS* witness list into the *C-CS* provenance of a query q is determined by a condition over the copy map of q. This check is implemented in the rewrites as an outermost projection

Direct-Copy-CS Copy Expressions

$$
\begin{aligned}
CM(R) &= \{a_1\} \rightarrow \mathscr{C}(a_1), \ldots, \{a_n\} \rightarrow \mathscr{C}(a_n) \text{ for } \mathbf{R} = (a_1, \ldots, a_n)\\
CM(\sigma_C(q_1)) &= CM(q_1)\\
CM(q_1 \diamond_C q_2) &= CM(q_1), CM(q_2)\\
CM(\alpha_{G,agg}(q_1)) &= (\mathscr{C}(a_1) \cap G) \rightarrow \mathscr{C}(a_1), \ldots, (\mathscr{C}(a_n) \cap G) \rightarrow \mathscr{C}(a_n) \text{ for } \mathscr{B}(q_1) = (a_1, \ldots, a_n)\\
CM(\Pi_A(q_1)) &= \bigcup_{x \in A} \mathscr{C}^*(a_1, x) \rightarrow \mathscr{C}(a_1), \ldots, \bigcup_{x \in A} \mathscr{C}^*(a_n, x) \rightarrow \mathscr{C}(a_n) \text{ for } \mathscr{B}(q_1) = (a_1, \ldots, a_n)\\
\mathscr{C}^*(a_i, x) &= \begin{cases} \{x\} \cap \mathscr{C}(a_i) & \text{for } x \in \mathbf{Q_1}\\ if\ (C)\ then\ (\{y\} \cap \mathscr{C}(a_i))\ else\ (\emptyset) & \text{for } x = \ if\ (C)\ then\ (y)\ else\ (e)\\ if\ (C)\ then\ (\emptyset)\ else\ (\{y\} \cap \mathscr{C}(a_i)) & \text{for } x = \ if\ (C)\ then\ (e)\ else\ (y)\\ if\ (y \in \mathscr{C}(a_i))\ then\ (\{z\})\ else\ (\emptyset) & \text{for } x = (y \rightarrow z)\\ \emptyset & \text{else} \end{cases}\\
CM(q_1 \cup q_2) &= CM(q_1), CM(q_2)\\
CM(q_1 \cap q_2) &= CM(q_1), CM(q_2)\\
CM(q_1 - q_2) &= CM(q_1), null(\mathscr{C}(q_2)) \rightarrow \mathscr{C}(q_2)
\end{aligned}
$$

Transitive-Copy-CS Copy Expressions

$$
\begin{aligned}
CM(R) =& \{a_1\} \rightarrow \mathscr{C}(a_1), \ldots, \{a_n\} \rightarrow \mathscr{C}(a_n)\\
& \text{for } \mathbf{R} = (a_1, \ldots, a_n)\\
CM(\sigma_C(q_1)) =& (\mathscr{C}^*(q_1, a_1) \cup \mathscr{C}(a_1)) \rightarrow \mathscr{C}(a_1) \ldots, (\mathscr{C}^*(q_1, a_n) \cup \mathscr{C}(a_n)) \rightarrow \mathscr{C}(a_n) \text{ for } \mathscr{B}(q_1) = (a_1, \ldots, a_n)\\
CM(q_1 \diamond_C q_2) =& (\mathscr{C}^*(q_1, a_1) \cup \mathscr{C}(a_1)) \rightarrow \mathscr{C}(a_1) \ldots, (\mathscr{C}^*(q_1, a_n) \cup \mathscr{C}(a_n)) \rightarrow \mathscr{C}(a_n),\\
& (\mathscr{C}^*(q_2, b_1) \cup \mathscr{C}(b_1)) \rightarrow \mathscr{C}(b_1), \ldots, (\mathscr{C}^*(q_2, b_m) \cup \mathscr{C}(b_m)) \rightarrow \mathscr{C}(b_m)\\
& \text{for } \mathscr{B}(q_1) = (a_1, \ldots, a_n) \text{ and } \mathscr{B}(q_2) = (b_1, \ldots, b_m)\\
\mathscr{C}^*(q_i, a) =& \bigcup_{((x=y) \in C \wedge x \in \mathbf{Q_i}) \vee ((y=x) \in C \wedge x \in \mathbf{Q_i})} if\ ((x = y) \wedge x \in \mathscr{C}(a))\ then\ (\{y\})\ else\ (\emptyset)\\
CM(\alpha_{G,agg}(q_1)) =& (\mathscr{C}(a_1) \cap G) \rightarrow \mathscr{C}(a_1), \ldots, (\mathscr{C}(a_n) \cap G) \rightarrow \mathscr{C}(a_n) \text{ for } \mathscr{B}(q_1) = (a_1, \ldots, a_n)\\
CM(\Pi_A(q_1)) =& \bigcup_{x \in A} \mathscr{C}^*(a_1, x) \rightarrow \mathscr{C}(a_1), \ldots, \bigcup_{x \in A} \mathscr{C}^*(a_n, x) \rightarrow \mathscr{C}(a_n) \text{ for } \mathscr{B}(q_1) = (a_1, \ldots, a_n)\\
\mathscr{C}^*(a_i, x) =& \begin{cases} \{x\} \cap \mathscr{C}(a_i) & \text{for } x \in \mathbf{Q_1}\\ if\ (C)\ then\ (\{y\} \cap \mathscr{C}(a_i))\ else\ (\emptyset) & \text{for } x = \ if\ (C)\ then\ (y)\ else\ (e)\\ if\ (C)\ then\ (\emptyset)\ else\ (\{y\} \cap \mathscr{C}(a_i)) & \text{for } x = \ if\ (C)\ then\ (e)\ else\ (y)\\ if\ (y \in \mathscr{C}(a_i))\ then\ (\{z\})\ else\ (\emptyset) & \text{for } x = (y \rightarrow z)\\ \emptyset & \text{else} \end{cases}\\
CM(q_1 \cup^{S/B} q_2) =& CM(q_1), CM(q_2)\\
CM(q_1 \cap^{S/B} q_2) =& CM(q_1), CM(q_2)\\
CM(q_1 -^{S/B} q_2) =& CM(q_1), null(\mathscr{C}(q_2)) \rightarrow \mathscr{C}(q_2)
\end{aligned}
$$

Figure 4.13: C-CS Copy Expressions Definition

that includes parts of a generated witness list representation depending of the content of the sets stored in the $\mathscr{C}(q)$ attributes.

Example 4.12. *For instance, assume the CDC-CS provenance is computed for a query $q = R ⟕_{a=b} S$ over the relations presented in example 4.11 (R and S with schemas $\mathbf{R} = (a)$ and $\mathbf{S} = (b,c)$). For each witness list w in $\mathscr{DD}(q,t)$ for some t, the part of w that stores a tuple from S is included in the corresponding witness list for CDC-CS of t if $\mathscr{CM}(q,b,t,w) \neq \emptyset \wedge \mathscr{CM}(q,c,t,w) \neq \emptyset$.*

Our approach to generate the *PI-CS* witness lists and check the inclusion of parts of the witness lists in the *C-CS* provenance is as follows:

- Extend the *PI-CS* rewrite rules to propagate the copy attributes in addition to the provenance attributes using the *CM* copy expressions. These extended rewrites are modeled as a meta-operator *C*.
- Apply an outermost projection to the rewritten algebra expression to check the inclusion of witness list parts according to the copy sets stored in the $\mathscr{C}$ attributes. The projection expressions that implement this check are denoted as $\mathscr{P}^*(q)$.

In the inclusion conditions over the *CM* copy expressions these conditions are modeled as projections of the form $if\ (C^*(R_i))\ then\ (\mathscr{N}(a))\ else\ (\emptyset) \rightarrow \mathscr{N}(a)$. Here R_i denotes one of the base relations accessed by q and a is an attribute from $\mathbf{R_i}$. For *C-CS* types with direct copying, the condition $C^*(R_i)$ for $\mathbf{R_i} = (a_1,\ldots,a_n)$ is defined as:

$$\mathscr{C}(a_1) \neq \emptyset \wedge \ldots \wedge \mathscr{C}(a_n) \neq \emptyset$$

For the *transitive copy C-CS* types $C^*(R_i)$ is defined as:

$$\mathscr{C}(a_1) \neq \emptyset \vee \ldots \vee \mathscr{C}(a_n) \neq \emptyset$$

Example 4.13. *The inclusion conditions for q from example 4.12 are modeled as the following projection expressions:*

$$if\ (\mathscr{C}(b) \neq \emptyset \wedge \mathscr{C}(c) \neq \emptyset)\ then\ (\mathscr{N}(b))\ else\ (\emptyset) \rightarrow \mathscr{N}(b),$$
$$if\ (\mathscr{C}(b) \neq \emptyset \wedge \mathscr{C}(c) \neq \emptyset)\ then\ (\mathscr{N}(c))\ else\ (\emptyset) \rightarrow \mathscr{N}(c)$$

With the copy expressions and the inclusion conditions in place the *C-CS* rewrites are modeled as a meta-operator for each *C-CS* type: *CD* for *CDC-CS*, *CT* for *CTC-CS*, *PD* for *PDC-CS*, and *PT* for *PTC-CS*.

Definition 4.4 (C-CS Provenance Rewrite Meta-Operators)**.** *The provenance rewrite meta algebra operators $CD/CT/PD/PT : \mathscr{E} \rightarrow \mathscr{E}$ map an algebra expression q to a rewritten algebra expression $q^{CD/CT/PD/PT}$. $CD/CT/PD/PT$ are defined over the structure of q, the inclusion expressions $\mathscr{P}^*(q^C)$ (Figure 4.14), and an additional meta-operator C that is defined as rewrite rules for each algebra operator which are shown in Figure 4.14:*

$$q^{CD/CT/PD/PT} = \Pi^B{}_{\mathbf{Q},\mathscr{P}^*(q^C)}(q^C)$$

The rewrite rules that implement C are simple extensions of the *PI-CS* rewrite rules. The only difference is that in addition to the provenance attributes $\mathscr{P}$ also the copy attributes $\mathscr{C}$ are propagated by the rewritten algebra expressions. Note that we define only *PI-CS* semantics rewrites for union and set difference. In the definition of $\mathscr{P}^*(q)$ we use R_j to denote a base relation accessed by q.

Structural Rewrite

Unary Operators

$$q^C = R^C = \Pi^B{}_{\mathbf{R},\mathbf{R}\rightarrow\mathscr{N}(\mathbf{R}),CM(q)}(R) \tag{C1}$$

$$q^C = (\sigma_C(q_1))^C = \Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^C),CM(q)}(\sigma_C({q_1}^C)) \tag{C2}$$

$$q^C = (\Pi^{S/B}{}_A(q_1))^C = \Pi^B{}_{A,\mathscr{P}(q^C),CM(q)}({q_1}^C) \tag{C3}$$

$$q^C = (\alpha_{G,agg}(q_1))^C = \Pi^B{}_{G,agg,\mathscr{P}(q^C),CM(q)}(\alpha_{G,agg}(q_1) \mathbin{\text{⟕}}_{G=_nX} \Pi^B{}_{G\rightarrow X,\mathscr{P}({q_1}^C),\mathscr{C}(q_1)}({q_1}^C)) \tag{C4}$$

Join Operators

$$q^C = (q_1 \times q_2)^C = \Pi^B{}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^C),CM(q)}({q_1}^C \times {q_2}^C) \tag{C5.a}$$

$$q^C = (q_1 \bowtie_C q_2)^C = \Pi^B{}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^C),CM(q)}({q_1}^C \bowtie_C {q_2}^C) \tag{C5.b}$$

$$q^C = (q_1 \mathbin{\text{⟕}}_C q_2)^C = \Pi^B{}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^C),CM(q)}({q_1}^C \mathbin{\text{⟕}}_C {q_2}^C) \tag{C5.c}$$

$$q^C = (q_1 \mathbin{\text{⟖}}_C q_2)^C = \Pi^B{}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^C),CM(q)}({q_1}^C \mathbin{\text{⟖}}_C {q_2}^C) \tag{C5.d}$$

$$q^C = (q_1 \mathbin{\text{⟗}}_C q_2)^C = \Pi^B{}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{P}(q^C),CM(q)}({q_1}^C \mathbin{\text{⟗}}_C {q_2}^C) \tag{C5.e}$$

Set Operations

$$\begin{aligned} q^C = (q_1 \cup^{S/B} q_2)^C =& (\Pi^B{}_{\mathbf{Q_1},\mathscr{P}({q_1}^C),\mathscr{P}({q_2}^C),\mathscr{C}(q_1),\mathscr{C}(q_2)}({q_1}^C \times null(\mathscr{P}({q_2}^C)) \times null(\mathscr{C}(q_2)))) \\ & \cup^B (\Pi^B{}_{\mathbf{Q_2},\mathscr{P}({q_1}^C),\mathscr{P}({q_2}^C),\mathscr{C}(q_1),\mathscr{C}(q_2)}({q_2}^C \times null(\mathscr{P}({q_1}^C)) \times null(\mathscr{C}(q_1)))) \end{aligned} \tag{C6}$$

$$\begin{aligned} q^C = (q_1 \cap^{S/B} q_2)^C =& \Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^C),CM(q)}(q_1 \cap^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_nX} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}({q_1}^C),CM(q_1)}({q_1}^C) \\ & \bowtie_{\mathbf{Q_1}=_nY} \Pi^B{}_{\mathbf{Q_2}\rightarrow Y,\mathscr{P}({q_2}^C),CM(q_2)}({q_2}^C)) \end{aligned} \tag{C7}$$

$$\begin{aligned} q^C = (q_1 -^{S/B} q_2)^C =& \Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^C),CM(q)}(q_1 -^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_nX} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}({q_1}^C),CM(q_1)}({q_1}^C) \\ & \times null(\mathscr{P}({q_2}^C)) \times null(CM(q_2))) \end{aligned} \tag{C8}$$

Provenance Attribute Inclusions Expressions

$$\mathscr{P}^*(q^C) = if\ (\mathscr{C}^*(a_1))\ then\ (\mathscr{N}(a_1))\ else\ (\varepsilon) \rightarrow \mathscr{N}(a_1), \ldots,\ if\ (\mathscr{C}^*(a_n))\ then\ (\mathscr{N}(a_n))\ else\ (\varepsilon) \rightarrow \mathscr{N}(a_n)$$

$\mathscr{C}^*(a_i) = (\mathscr{C}(b_1) \neq \emptyset \wedge \ldots \wedge \mathscr{C}(b_x) \neq \emptyset)$ for $a \in \mathbf{R_j} = (b_1, \ldots, b_x)$ (CDC-CS / CTC-CS)

$\mathscr{C}^*(a_i) = (\mathscr{C}(b_1) \neq \emptyset \vee \ldots \vee \mathscr{C}(b_x) \neq \emptyset)$ for $a \in \mathbf{R_j} = (b_1, \ldots, b_x)$ (PDC-CS / PTC-CS)

Figure 4.14: *C-CS* Algebraic Rewrite Rules

4.4.1 Rewrite Example

Figures 4.15 and 4.16 presented example rewrites for *CDC-CS* and *PTC-CS* for the example queries q_a and q_b. The *CDC-CS* meta-operator rewrites the algebra expression q_a into an algebra expression ${q_a}^{CD}$. The base relation accesses of q (R and S) are rewritten (rewrite rule **(C1)**) by duplicating the attributes of the base relation to generate $\mathscr{P}(R^+)$ and $\mathscr{P}(S^+)$. In addition the copy attributes are initialized with singleton sets. E.g., $\{a\} \rightarrow \mathscr{C}(a)$. Rewrite rule **(C5.b)** is used to rewrite the join in q_a by applying an additional projection that produces the correct ordering of the original result attributes of the join, the provenance attributes, and the copy attributes. The projection on $a \rightarrow x$ is transformed into a modified projection **(C3)**. Additional projection expressions are added to the projection for attributes from $\mathscr{P}$ and conditional expressions for attributes from $\mathscr{C}$. Applying the definition of *CM* for projection leads to the following projection expressions:

$$if\ (a \in \mathscr{C}(a))\ then\ (\{x\})\ else\ (\emptyset) \rightarrow \mathscr{C}(a),$$
$$if\ (a \in \mathscr{C}(b))\ then\ (\{x\})\ else\ (\emptyset) \rightarrow \mathscr{C}(b)$$

Each of these expressions checks if attribute a is included in the set stored in a $\mathscr{C}$ attribute and if so adds x to this set. This is valid behavior because if a is present in a set $\mathscr{C}(y)$ this means that the projection is copying the value of attribute y to x. Finally, an outermost projection is used to evaluate the inclusion conditions for attribute values from $\mathscr{P}$. E.g., $if\ (\mathscr{C}(a))\ then\ (\mathscr{N}(a))\ else\ (\varepsilon) \rightarrow \mathscr{N}(a)$.

Figure 4.15 also presents the rewritten algebra expression for *PTC-CS* (${q_a}^{PT}$). The only difference between ${q_a}^{PT}$ and ${q_a}^{CD}$ are the *CM* expressions used for the join rewrite. For *PTC-CS* these expressions check for copy operations induced by the join condition. For instance:

$$\begin{aligned} & if\ (a = b \wedge b \in \mathscr{C}(a))\ then\ (\{a\})\ else\ (\emptyset) \\ & \cup\ if\ (a = b \wedge a \in \mathscr{C}(a))\ then\ (\{b\})\ else\ (\emptyset) \\ & \cup \mathscr{C}(a) \\ \rightarrow & \mathscr{C}(a) \end{aligned}$$

This expression includes a in the copy set for attribute b if the join condition $a = b$ is fulfilled. For q_a the condition $a = b \wedge b \in \mathscr{N}(b)$ is fulfilled for all input tuples, but in the general case the condition is required. E.g., for join conditions that use disjunction. For instance, if the join condition is $a = b \vee a = 3$ then $a = b$ is not necessarily fulfilled by all result tuples of the join.

Figure 4.16 presents the *C-CS* rewrites for a second example query q_b. The join and base relations accesses in q_b are rewritten using rewrite rules (C1) and (C3). This is similar to the rewrite of q_a except that the base relations have two attributes. For *CDC-CS* the projection on attribute c is rewritten by adding copy expressions that intersect the individual $\mathscr{C}$ sets with the singleton set containing solely attribute c. Each condition used by the $\mathscr{P}^*$ projection expressions applied in the outermost selection of the rewritten query references all copy sets for a base relation. For instance, the schema of base relation U is $\mathbf{U} = (c,d)$. Therefore, the inclusion expressions for attributes c and d both use the following condition:

$$\mathscr{C}(c) \neq \emptyset \wedge \mathscr{C}(d) \neq \emptyset$$

For *PTC-CS* the rewrite is similar except that the generated join *CM* and inclusion expressions are different. The *CM* expressions take the transitive copying of the join condition into account by merging each input $\mathscr{C}(x)$ attributes with the following conditional expressions:

$$if\ (c = e \wedge \{c\} \in \mathscr{C}(x))\ then\ (\{e\})\ else\ (\emptyset)$$
$$if\ (c = e \wedge \{e\} \in \mathscr{C}(x))\ then\ (\{c\})\ else\ (\emptyset)$$

If the selection condition $c = e$ is fulfilled the input $\mathscr{C}$ sets are unioned with the singleton set $\{c\}$ and $\{e\}$. In contrast to ${q_b}^{CD}$ the $\mathscr{P}^*$ expressions in ${q_b}^{PT}$ use disjunctions instead of conjunctions.

R

a
1
3

S

b
1
5

$\mathbf{Q_a}$

x
1

$$q_a = \Pi^S{}_{a \rightarrow x}(R ⟕_{a=b} S)$$

$$\mathscr{C}\mathscr{D}(q_a, (1)) = \{<(1), \perp>\} \qquad \mathscr{P}\mathscr{T}(q_a, (1)) = \{<(1), (1)>\}$$

$$\begin{aligned}
{q_a}^{CD} =& \Pi^B{}_{x, \mathscr{P}^*(q_a)}(\\
& \Pi^B{}_{a \rightarrow x, \mathscr{N}(a), \mathscr{N}(b), if\ (a \in \mathscr{C}(a))\ then\ (\{x\})\ else\ (\emptyset) \rightarrow \mathscr{C}(a), if\ (a \in \mathscr{C}(b))\ then\ (\{x\})\ else\ (\emptyset) \rightarrow \mathscr{C}(b)}(\\
& \Pi^B{}_{a,b,\mathscr{N}(a),\mathscr{N}(b),\mathscr{C}(a),\mathscr{C}(b)}(\Pi^B{}_{a,a \rightarrow \mathscr{N}(a), \{a\} \rightarrow \mathscr{C}(a)}(R) ⟕_{a=b} \Pi^B{}_{b,b \rightarrow \mathscr{N}(b), \{b\} \rightarrow \mathscr{C}(b)}(S)))) \\
\mathscr{P}^*(q_a) =& if\ (\mathscr{C}(a) \neq \emptyset)\ then\ (\mathscr{N}(a))\ else\ (\varepsilon) \rightarrow \mathscr{N}(a), \\
& if\ (\mathscr{C}(b) \neq \emptyset)\ then\ (\mathscr{N}(b))\ else\ (\varepsilon) \rightarrow \mathscr{N}(b)
\end{aligned}$$

$$\begin{aligned}
{q_a}^{PT} =& \Pi^B{}_{x, \mathscr{P}^*(q_a)}(\\
& \Pi^B{}_{a \rightarrow x, \mathscr{N}(a), \mathscr{N}(b), if\ (a \in \mathscr{C}(a))\ then\ (\{x\})\ else\ (\emptyset) \rightarrow \mathscr{C}(a), if\ (\mathscr{C}(b) \neq \emptyset)\ then\ (\mathscr{N}(b))\ else\ (\varepsilon) \rightarrow \mathscr{N}(b)}(\\
& \Pi^B{}_{a,b,\mathscr{N}(a),\mathscr{N}(b),CM(R ⟕_{a=b} S)}(\\
& \Pi^B{}_{a,a \rightarrow \mathscr{N}(a), \{a\} \rightarrow \mathscr{C}(a)}(R) ⟕_{a=b} \Pi^B{}_{b,b \rightarrow \mathscr{N}(b), \{b\} \rightarrow \mathscr{C}(b)}(S)))) \\
CM(R ⟕_{a=b} S) =& if\ (a = b \wedge a \in \mathscr{C}(a))\ then\ (\{b\})\ else\ (\emptyset) \\
& \cup\ if\ (a = b \wedge b \in \mathscr{C}(a))\ then\ (\{a\})\ else\ (\emptyset) \\
& \cup \mathscr{C}(a) \rightarrow \mathscr{C}(a), \\
& if\ (a = b \wedge b \in \mathscr{C}(b))\ then\ (\{a\})\ else\ (\emptyset) \\
& \cup\ if\ (a = b \wedge a \in \mathscr{C}(b))\ then\ (\{b\})\ else\ (\emptyset) \\
& \cup \mathscr{C}(b) \rightarrow \mathscr{C}(b) \\
\mathscr{P}^*(q_a) =& if\ (\mathscr{C}(a) \neq \emptyset)\ then\ (\mathscr{N}(a))\ else\ (\varepsilon) \rightarrow \mathscr{N}(a), \\
& if\ (\mathscr{C}(b) \neq \emptyset)\ then\ (\mathscr{N}(b))\ else\ (\varepsilon) \rightarrow \mathscr{N}(b)
\end{aligned}$$

$\mathbf{Q_a}^{\mathbf{CD}}$

a	$\mathscr{N}(a)$	$\mathscr{N}(b)$
1	1	ε

$\mathbf{Q_a}^{\mathbf{PT}}$

a	$\mathscr{N}(a)$	$\mathscr{N}(b)$
1	1	1

Figure 4.15: *C-CS* Rewrite Example

U

c	d
3	2
3	6

V

e	f
3	3
5	4

$\mathbf{Q_b}$

c
3

$$q_b = \Pi^S{}_c(U \bowtie_{c=e} V)$$

$$\mathscr{CD}(q_b,(3)) = \{<\perp,\perp>\} \qquad \mathscr{PT}(q_b,(3)) = \{<(3,2),(3,3)>,<(3,6),(3,3)>\}$$

$$\begin{aligned}
{q_b}^{CD} =&\Pi^B{}_{c,\mathscr{N}(c),\mathscr{N}(d),\mathscr{N}(e),\mathscr{N}(f),\mathscr{P}^*(q_b)}(\\
&\Pi^B{}_{c,\mathscr{N}(c),\mathscr{N}(d),\mathscr{N}(e),\mathscr{N}(f),\{c\}\cap\mathscr{C}(c)\rightarrow\mathscr{C}(c),\{c\}\cap\mathscr{C}(d)\rightarrow\mathscr{C}(d),\{c\}\cap\mathscr{C}(e)\rightarrow\mathscr{C}(e),\{c\}\cap\mathscr{C}(f)\rightarrow\mathscr{C}(f)}(\\
&\Pi^B{}_{c,d,e,f,\mathscr{N}(c),\mathscr{N}(d),\mathscr{N}(e),\mathscr{N}(f),\mathscr{C}(c),\mathscr{C}(d),\mathscr{C}(e),\mathscr{C}(f)}(\\
&\Pi^B{}_{c,d,c\rightarrow\mathscr{N}(c),d\rightarrow\mathscr{N}(d),\{c\}\rightarrow\mathscr{C}(c),\{d\}\rightarrow\mathscr{C}(d)}(U) \bowtie_{c=e} \Pi^B{}_{e,f,e\rightarrow\mathscr{N}(e),f\rightarrow\mathscr{N}(f),\{e\}\rightarrow\mathscr{C}(e),\{f\}\rightarrow\mathscr{C}(f)}(V))))\\
\mathscr{P}^*(q_b) =& if\ (\mathscr{C}(c) \neq \emptyset \wedge \mathscr{C}(d) \neq \emptyset)\ then\ (\mathscr{N}(c))\ else\ (\varepsilon) \rightarrow \mathscr{N}(c),\\
& if\ (\mathscr{C}(c) \neq \emptyset \wedge \mathscr{C}(d) \neq \emptyset)\ then\ (\mathscr{N}(d))\ else\ (\varepsilon) \rightarrow \mathscr{N}(d),\\
& if\ (\mathscr{C}(e) \neq \emptyset \wedge \mathscr{C}(f) \neq \emptyset)\ then\ (\mathscr{N}(e))\ else\ (\varepsilon) \rightarrow \mathscr{N}(e),\\
& if\ (\mathscr{C}(e) \neq \emptyset \wedge \mathscr{C}(f) \neq \emptyset)\ then\ (\mathscr{N}(f))\ else\ (\varepsilon) \rightarrow \mathscr{N}(f)
\end{aligned}$$

$$\begin{aligned}
{q_b}^{PT} =&\Pi^B{}_{c,\mathscr{N}(c),\mathscr{N}(d),\mathscr{N}(e),\mathscr{N}(f),\mathscr{P}^*(q_b)}(\\
&\Pi^B{}_{c,\mathscr{N}(c),\mathscr{N}(d),\mathscr{N}(e),\mathscr{N}(f),\{c\}\cap\mathscr{C}(c)\rightarrow\mathscr{C}(c),\{c\}\cap\mathscr{C}(d)\rightarrow\mathscr{C}(d),\{c\}\cap\mathscr{C}(e)\rightarrow\mathscr{C}(e),\{c\}\cap\mathscr{C}(f)\rightarrow\mathscr{C}(f)}(\\
&\Pi^B{}_{c,d,e,f,\mathscr{N}(c),\mathscr{N}(d),\mathscr{N}(e),\mathscr{N}(f),CM(U\bowtie_{c=e}V)}(\\
&\Pi^B{}_{c,d,c\rightarrow\mathscr{N}(c),d\rightarrow\mathscr{N}(d),\{c\}\rightarrow\mathscr{C}(c),\{d\}\rightarrow\mathscr{C}(d)}(U)\\
&\bowtie_{c=e} \Pi^B{}_{e,f,e\rightarrow\mathscr{N}(e),f\rightarrow\mathscr{N}(f),\{e\}\rightarrow\mathscr{C}(e),\{f\}\rightarrow\mathscr{C}(f)}(V))))\\
CM(U \bowtie_{c=e} V) =& if\ (c = e \wedge \{c\} \in \mathscr{C}(c))\ then\ (\{e\})\ else\ (\emptyset)\\
&\cup\ if\ (c = e \wedge \{e\} \in \mathscr{C}(c))\ then\ (\{c\})\ else\ (\emptyset)\\
&\cup \mathscr{C}(c) \rightarrow \mathscr{C}(c),\\
& if\ (c = e \wedge \{c\} \in \mathscr{C}(d))\ then\ (\{e\})\ else\ (\emptyset)\\
&\cup\ if\ (c = e \wedge \{e\} \in \mathscr{C}(d))\ then\ (\{c\})\ else\ (\emptyset)\\
&\cup \mathscr{C}(d) \rightarrow \mathscr{C}(d),\\
& if\ (c = e \wedge \{c\} \in \mathscr{C}(e))\ then\ (\{e\})\ else\ (\emptyset)\\
&\cup\ if\ (c = e \wedge \{e\} \in \mathscr{C}(e))\ then\ (\{c\})\ else\ (\emptyset)\\
&\cup \mathscr{C}(e) \rightarrow \mathscr{C}(e),\\
& if\ (c = e \wedge \{c\} \in \mathscr{C}(f))\ then\ (\{e\})\ else\ (\emptyset)\\
&\cup\ if\ (c = e \wedge \{e\} \in \mathscr{C}(f))\ then\ (\{c\})\ else\ (\emptyset)\\
&\cup \mathscr{C}(f) \rightarrow \mathscr{C}(f)\\
\mathscr{P}^*(q_b) =& if\ (\mathscr{C}(c) \neq \emptyset \vee \mathscr{C}(d) \neq \emptyset)\ then\ (\mathscr{N}(c))\ else\ (\varepsilon) \rightarrow \mathscr{N}(c),\\
& if\ (\mathscr{C}(c) \neq \emptyset \vee \mathscr{C}(d) \neq \emptyset)\ then\ (\mathscr{N}(d))\ else\ (\varepsilon) \rightarrow \mathscr{N}(d),\\
& if\ (\mathscr{C}(e) \neq \emptyset \vee \mathscr{C}(f) \neq \emptyset)\ then\ (\mathscr{N}(e))\ else\ (\varepsilon) \rightarrow \mathscr{N}(e),\\
& if\ (\mathscr{C}(e) \neq \emptyset \vee \mathscr{C}(f) \neq \emptyset)\ then\ (\mathscr{N}(f))\ else\ (\varepsilon) \rightarrow \mathscr{N}(f)
\end{aligned}$$

$\mathbf{Q_b}^{\mathbf{CD}}$

c	$\mathscr{N}(c)$	$\mathscr{N}(d)$	$\mathscr{N}(e)$	$\mathscr{N}(f)$
3	ε	ε	ε	ε

$\mathbf{Q_b}^{\mathbf{PT}}$

c	$\mathscr{N}(c)$	$\mathscr{N}(d)$	$\mathscr{N}(e)$	$\mathscr{N}(f)$
3	3	2	3	3
3	3	6	3	3

Figure 4.16: *C-CS* Rewrite Example Query q_b

4.4.2 Proof of Correctness and Completeness

Similar to the proofs for *PI-CS* we have to show that the *C-CS* meta-operators generate the relational representation of *C-CS*.

Theorem 4.9 (Correctness and Completeness of the C-CS Meta-Operators)**.** *Let q be an algebra statement. The rewritten queries produced by the meta-operators CD, CT, PD, and PT produce the relational representation of provenance according to CDC-CS, CTC-CS, PDC-CS, and PTC-CS:*

$$Q^{CD} = Q^{\mathscr{C}\mathscr{D}}$$
$$Q^{CT} = Q^{\mathscr{C}\mathscr{T}}$$
$$Q^{PD} = Q^{\mathscr{P}\mathscr{D}}$$
$$Q^{PT} = Q^{\mathscr{P}\mathscr{T}}$$

Proof.

Result Preservation

The only difference between a rewritten query $q^{CD/CT/PD/PT}$ and the query q^{+} generated by the meta-operator for *PI-CS* is the existence of additional projection expressions and duplicate preserving projections. The additional projections do not project out original result attributes. From the definition of projection we know that a duplicate preserving projection generates an output tuple for each of its input tuples. Since q^{+} fulfills the result preservation property we conclude that this property is fulfilled for $q^{CD/CT/PD/PT}$ too.

Witness List Preservation

The *C-CS* provenance of a query q contains the same witness lists as the *PI-CS* provenance of q except that tuples in these witness lists may be replaced by $\perp$ depending on the *copy map* of q. Since the rewrite rules for *C-CS* produce the same $\mathscr{P}$ attribute values as the *PI-CS* rewrite rules, it remains to show that the copy map expressions *CM* of the *C-CS* rules model the copy map correctly and that the provenance attribute inclusion expressions $\mathscr{P}^{*}$ exclude parts of a witness list representation according to the definition of the *C-CS* types.

Correctness of the Copy Expressions:

We first prove that the copy map of q is modeled correctly by the $CM(q)$ expressions. This assumption is proven by induction over the structure of an algebra expression. We have to prove that for each attribute a from a base relation accessed by q, a result tuple t of q, and a witness list w from $\mathscr{D}\mathscr{D}(q,t)$ the following holds:

$$\Pi^{S}{}_{\mathscr{C}(a)}(\sigma_{\mathbf{Q}=t \wedge \mathscr{P}(q^{+})=w'}(q^{+})) = \mathscr{C}\mathscr{M}(q,a,w,t)$$

This property has to be proven for both the direct copy and the transitive copy definition of $CM(q)$.

Induction Start:

For $q = R$ both the direct copy map and the transitive copy map of q is independent of the t and w. This means the equality we have to prove can be simplified to the following equality that trivially holds:

$$\mathscr{C}\mathscr{M}(q,a,w,t) = \{a\} = \Pi^{S}{}_{\mathscr{C}(a)}(\Pi^{B}{}_{\mathbf{R},\mathbf{R} \rightarrow \mathscr{N}(\mathbf{R}),\{a_1\} \rightarrow \mathscr{C}(a_1),\ldots}(R))$$

Induction Step:

Given that for an algebra expression with nesting depth n the copy expressions model the copy map correctly we have to prove that this is also the case for algebra expressions with nesting depth $n+1$.

Case $q = \sigma_C(q_1)$ (Direct Copy):
The direct-copy copy map for selection is equal to the copy map of the input to the selection. The same holds for the direct-copy copy expressions for selection.
Case $q = \sigma_C(q_1)$ (Transitive Copy):
The transitive copy version of CM for selection unions each $\mathscr{C}(a)$ from $\mathscr{C}(q_1)$ with

$$if\ ((x = y) \wedge y \in \mathscr{C}(a))\ then\ (\{x\})\ else\ (\emptyset)$$

for each condition $(x = y)$ or $(y = x)$ in the selection predicate C. In $\mathscr{CM}(q,a,t,w)$ the input copy map is unioned with

$$\{x \mid \exists y : (x = y) \in C \wedge w \models (x = y) \wedge y \in \mathscr{CM}(q_1,a,w,t)\}$$

If attribute y is in $\mathscr{CM}(q_1,a,w,t)$ then we know from the induction hypothesis that $\mathscr{C}(a)$ contains y. The condition $w \models (x = y)$ is equivalent to that the tuple $t' \in Q_1$ representing w fulfills $(x = y)$. Since ${Q_1}^+$ contains a tuple $(t', w', CM(q_1))$ if t' is derived from w, the copy expressions model the copy map correctly.
Case $q = \Pi^{S/B}{}_A(q_1)$:
Both the copy map of q and the copy expressions for q include attributes from the copy map respective copy attributes of q_1 if they appear in the projection list A. For other constructs in A both the copy map and copy expressions of projection union the input attribute sets with additional attributes depending on certain preconditions. For a renaming $y \rightarrow z$ the copy map of the projection for an attribute a contains z, if y is contained in the copy map of a for q_1. This is modeled in the copy expressions as $if\ (y \in \mathscr{C}(a))\ then\ (\{z\})\ else\ (\emptyset)$ which is equivalent to the formulation used in the definition of the copy map. If one the projection expressions is a conditional expressions of the form $if\ (C)\ then\ (y)\ else\ (e)$, then the copy map of the projection for some attribute a includes y if C is fulfilled and y is also included in the copy map of C_1. The copy expressions use the equivalent formulation $if\ (C)\ then\ (\emptyset)\ else\ (\{y\} \cap \mathscr{C}(a))$. The case of a projection expression of form $if\ (C)\ then\ (e)\ else\ (y)$ is symmetric to the case we have just presented. If follows that the copy map and copy expressions for projection are equivalent.
Case $q = \alpha_{G,agg}(q_1)$:
The copy map for aggregation removes attributes from the copy map of the input if they do not belong to the group-by attributes. This is modeled in the copy expression by intersecting the input copy maps with the set of group-by attributes.
Case $q = q_1 \times q_2$:
Case $q = q_1 \bowtie_C q_2$ (Direct Copy):
The direct-copy copy map for a join simply unions the copy maps of the join's inputs. The copy expressions model this behaviour correctly by concatenating the copy expressions of the inputs. The same applies for the direct and transitive copy maps for cross product.
Case $q = q_1 \bowtie_C q_2$ (Transitive Copy): The transitive copy map for join contains the attributes from the direct copy map unioned with additional attributes based on selection conditions of the form $(x = y)$. The mechanisms applied by the copy map and the copy expressions for join to evaluate the selection conditions are the same as for selections. Therefore, using the same argument as for selection we conclude that the copy map and copy expressions for join are equivalent.
Case $q = q_1 ⟕_C q_2$:
The copy map for a left outer join contains the same attributes as the copy map for a join expect that attributes from the right hand side input of the join are only included for witness lists w that fulfill C. The copy expressions do not check this condition explicitly, but if w does not fulfill C, then in the result tuple of the rewritten left outer join that represents w the attributes from the rewritten right hand side will contain only null values. Recall that we use ε as an equivalent representation of the empty set. Therefore, the copy map and copy expressions for left outer join are equivalent.
Case $q = q_1 ⟖_C q_2$:
Case $q = q_1 ⟗_C q_2$:
The proofs for right outer and full outer join are analog to the proof for left outer join.
Case $q = q_1 \cup^{S/B}{}_C q_2$:
For both direct copy and transitive copy the copy map of union is the union of the input copy maps except that attributes from one input are omitted if the witness list $w[q_i]$ is $\perp [q_i]$ for this input. The copy

expressions for union are the concatenation of the copy expressions of the inputs of the union. If $w[q_i]$ is $\perp$ $[q_i]$ then q^+ will produce an output tuple with the $\mathscr{P}({q_1}^+)$ and $\mathscr{C}(q_1)$ attributes set to ε. Recall that ε is an alternative representation of $\emptyset$ for the data type used to represent the $\mathscr{C}$ sets. Therefore, each attribute $\mathscr{C}(a)$ will contain the same set of attribute names as the copy map of a.

Case $q = q_1 \cap^{S/B} C q_2$:

The copy maps for intersection is the union of the copy maps of the inputs of the intersection. The copy expressions for intersection are the concatenation of the copy expressions of its inputs. Therefore, each $\mathscr{C}(a)$ attributes is guaranteed to represent the same set as the copy map of a.

Case $q = q_1 -^{S/B} C q_2$: The copy map of a set difference q is the copy map of the left input q_1. The copy expressions for a set difference are the concatenation of the copy attributes from q_1 with null values which represent empty sets. From the induction hypothesis we know that the copy expressions for q_1 model the copy map of q_1 correctly. Therefore, the copy map and copy expressions are equivalent for this operation.

Correctness of the Inclusion Conditions:

We have proven that the *CM* expressions model the copy maps correctly. It remains to show that the inclusion conditions applied in the outermost projection in the *C-CS* rewrites simulate the behaviour of the inclusion conditions of the *C-CS* definitions. In the definition of the *C-CS* types with direct copying, a part $w[i]$ of a witness list is replaced with $\emptyset$, if the copy map is empty for at least one attribute a from the schema $\mathbf{R_i}$ the part of the witness list corresponds to $(\mathscr{CM}(q,a,w,t) = \emptyset)$. w.l.o.g. $\mathbf{R_i} = (\mathbf{a_1}, \dots, \mathbf{a_n})$. In the copy expressions the copy map of each attribute a is stored in one copy attribute $\mathscr{C}(a)$. The inclusion expressions for attribute a are defined as $if\ (\mathscr{C}^*(a))\ then\ (\mathscr{N}(a))\ else\ (\varepsilon)$ with $\mathscr{C}^*(a) = (\mathscr{C}(a_1) \neq \emptyset \wedge \ldots \wedge \mathscr{C}(a_n) \neq \emptyset)$. The condition $\mathscr{C}^*(a)$ is not fulfilled if at least one attribute $\mathscr{C}(a_i)$ stores the empty set which is exactly the condition applied in the definition of the *C-CS* types with direct copying. The proof for transitive copy is analog.

□

4.4.3 Rewrite Rules Simplifications

Surprisingly, in spite of the fact that the provenance of all *C-CS* types is a subset of the provenance according to *PI-CS*, the computation of the *C-CS* types is more complex than the computation of *PI-CS*. However, for a wide range of algebra expressions the rewritten query for *C-CS* can be simplified to a great extend. The potential simplifications are derived from the following observations:

1. **Copy Expressions with Fixed Results**: The projection copy expressions were designed to cope with arbitrary rewritten inputs. In a wide range of cases we can deduce from the input expression that some conditional clauses will always return the empty set and, therefore, can be omitted. Furthermore, if an input copy attribute is guaranteed to store the empty set, the empty set can just be passed on omitting conditional expressions and intersections completely for this attribute.

2. **Static Copy Map**: For many algebra operators the *copy-map* is independent of the instance data (t) and *PI-CS* witness list (w) parameters. For these algebra expressions the costly computation of the *copy-map* during run-time can be omitted, because it is known beforehand if tuples from a base relation will be included in the *C-CS* provenance. Hence, in these cases we can simply apply the *PI-CS* rewrite rules and replace provenance attributes with ε values where necessary.

3. **Omit Rewrite Simplification**: Based on the first observation we can deduce that if the provenance attributes for a base relation are guaranteed to be ε, it is not necessary to compute this part of the provenance at all. This means the sub-expression that produces this part of the relational representation of a witness is not rewritten at all.

4. **Combine Projections**: Adjacent projections added by the *C-CS* rewrites can often be combined into a single projection.

4.4.3.1 Copy Expression with Fixed Results

The copy expressions as defined *CM* were defined to be applicable for all possible input algebra expressions. Depending on the input they are applied to, some of the conditional expressions or set operations will always generate the same result. E.g., in query ${q_a}^{CD}$ (Figure 4.15) the expression

$$if\ (a \in \mathscr{C}(a))\ then\ (\{x\})\ else\ (\emptyset)$$

will evaluate to $\{x\}$ for all input tuples. Hence, it can be simplified to $\{x\}$. Another example are the intersections $\{c\} \cap \mathscr{C}(e/f/g)$ in ${q_b}^{PT}$ (Figure 4.16). A set stored in $\mathscr{C}(e/f/g)$ is guaranteed to contain only attribute $e/f/g$. Therefore, these intersections will evaluate to $\emptyset$ for all input tuples. One approach to identify fixed copy expressions is to use a bottom up traversal starting at the rewritten base relations that use only fixed copy expressions. In each step the copy expressions have to be examined to determine if they produce fixed results. As an example of this approach reconsider query ${q_a}^{CD}$. The *CM* expressions for the join applied in q_a are fixed, because they simply pass on the copy sets of their inputs. The copy expressions for the rewritten projection are conditional expressions. If we substitute the fixed input copy sets in the conditions of these expressions we get:

$$if\ (a \in \{a\})\ then\ (\{x\})\ else\ (\emptyset) = \{x\}$$
$$if\ (a \in \{b\})\ then\ (\{x\})\ else\ (\emptyset) = \emptyset$$

This means the conditional expressions are fixed too and can be substituted with their fixed evaluation result.

4.4.3.2 Static Copy Map Simplification

A static copy map is an extreme case of copy expressions with fixed results. As an example for a static copy map consider query q_a from Figure 4.15. Applying the definition of the direct-copy copy-map generates the following copy map that is independent of the t and w parameters:

$$\mathscr{C}\mathscr{M}(q_a, a, w, t) = \{x\} \qquad\qquad \mathscr{C}\mathscr{M}(q_a, b, w, t) = \emptyset$$

Therefore, the complex original rewrite is not necessary for this query. The *CDC-CS* provenance can be computed by simply using the outermost projection in ${q_a}^{+}$ to replace provenance attributes with ε if the copy map indicates that the relation they are derived from should not be included in the provenance:

$${q_a}^{CD'} = \Pi^{B}{}_{a \rightarrow x, \mathscr{N}(a), \varepsilon \rightarrow \mathscr{N}(b)}(\Pi^{B}{}_{a, a \rightarrow \mathscr{N}(a)}(R) \bowtie_{a=b} \Pi^{B}{}_{b, b \rightarrow \mathscr{N}(b)}(S))$$

In general, if a query q contains a sub-query *sub* for which the copy map is static, this sub-query can be rewritten by appending the static copy expressions to the outermost projection in *sub* (if the outermost operator is not a projection, a new projection is introduced).

Definition 4.5 (Static Copy Map)**.** *For an algebra expression q the copy map of q is called static if it fulfills the following condition:*

$$\forall t, t' \in Q, w \in \mathscr{D}\mathscr{D}(q,t), w' \in \mathscr{D}\mathscr{D}(q,t'), a : \mathscr{C}\mathscr{M}(q, a, w, t) = \mathscr{C}\mathscr{M}(q, a, w', t')$$

4.4.3.3 Omit Rewrite Simplification

Reconsider ${q_a}^{CD'}$, the simplified version of query ${q_a}^{CD}$ presented above. In this query the rewrite of base relation access S is superficial, because the generated provenance information is projected out by the outermost projection. This means we can simplify ${q_a}^{CD'}$ even further:

$$ {q_a}^{CD''} = {\Pi^B}_{a \rightarrow x, \mathscr{N}(a), \varepsilon \rightarrow \mathscr{N}(b)} ({\Pi^B}_{a, a \rightarrow \mathscr{N}(a)} (R) \rightouterjoin_{a=b} S) $$

This simplification can be applied to a sub-expression *sub* of an algebra expression q if all the base relations accessed by *sub* are guaranteed to be excluded from the provenance according to the copy map.

4.4.3.4 Combining Projections

Using standard relational algebra equivalences, two adjacent projections can be combined into one or a projection can be pulled up trough a join. For instance, in ${q_a}^{CD}$ the projections introduced by the base relation access rewrite can be pulled trough the join and combined with the projection introduced by the rewrite of the join.

Example 4.14. *The simplified versions of the C-CS queries from the example in Figure 4.15 are presented below.*

$$ \begin{aligned} {q_a}^{CD} &= {\Pi^B}_{a \rightarrow x, a \rightarrow \mathscr{N}(a), \varepsilon \rightarrow \mathscr{N}(b)} (R \rightouterjoin_{a=b} S) \\ {q_b}^{CD} &= {\Pi^B}_{c, \varepsilon \rightarrow \mathscr{N}(c), \varepsilon \rightarrow \mathscr{N}(d), \varepsilon \rightarrow \mathscr{N}(e), \varepsilon \rightarrow \mathscr{N}(f)} (U \rightouterjoin_{c=e} V) \\ {q_a}^{PT} &= {\Pi^B}_{a \rightarrow x, a \rightarrow \mathscr{N}(a), b \rightarrow \mathscr{N}(b)} (R \rightouterjoin_{a=b} S) \\ {q_b}^{PT} &= {\Pi^B}_{c, c \rightarrow \mathscr{N}(c), d \rightarrow \mathscr{N}(d), e \rightarrow \mathscr{N}(e), f \rightarrow \mathscr{N}(f)} (U \rightouterjoin_{c=e} V) \end{aligned} $$

Note that none of the simplified queries make use of the copy map construction. Even more, some of these queries are simpler than the rewritten queries for PI-CS. To simplify query ${q_a}^{CD}$ we use the fact that the conditional copy expressions introduced by the projection rewrite are fixed. Thus, they are replaced with their constant result ($\{x\} \rightarrow \mathscr{C}(a), \emptyset \rightarrow \mathscr{C}(b)$). The copy map is static for both base relations and for S is guaranteed to be the empty set. This means, the base relation access to S does not have to be rewritten at all, and for R we can omit the copy expression construction. In the resulting algebra expression, the projection that implements the rewrite of R can be pulled through the join and combined with the outermost projection. In the simplified version of ${q_a}^{CD}$ presented above, the provenance computation is implemented by a few additional projection expressions. As a second example consider the simplified version of ${q_b}^{PT}$. The copy map for this query is constant, because the conditions in the conditional expressions used in the join rewrite are always fulfilled. All tuples in the result of the join trivially fulfill the condition $c = e$ and the copy expressions of the inputs of the join are fixed. Therefore, also the intersections used by the projection rewrite generate a fixed result. Note that for the general case the conditional expression for join rewrites are necessary to guaranty correctness for, e.g., disjunctive join expressions. Because of the static copy map the copy expressions can be omitted in the rewritten query. ${q_a}^{PT}$ is further simplified by omitting the rewrite for V (the copy map result for attributes from V is guaranteed to be the empty set) and combining projections. In summary, applying the presented simplifications may lead to rewritten algebra expressions that are much more efficient and easier to comprehend. Note that the applicability of the projection combination is not limited to C-CS queries, but can be applied for PI-CS queries too.

R

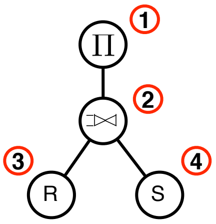
a	b
1	2
1	3
2	3
2	5

S

c
2
3

Q

a
1
2



$$q_a = \Pi^S{}_a(R ⟕_{b=c} S)$$

$$\mathscr{T}(q_c,(1)) : \theta_{<(1,2),(2)>}(op) = 1 \qquad \theta_{<(1,3),(3)>}(op) = 1$$

$$\mathscr{T}(q_c,(2)) : \theta_{<(2,3),(3)>}(op) = 1 \qquad \theta_{<(2,5),\perp>}(op) = \begin{cases} 0 \text{ if } op = \mathbf{S} \\ 1 \text{ else} \end{cases}$$

$$\Theta_{<(1,2),(2)>} = \{1,2,3,4\} \qquad \Theta_{<(1,3),(3)>} = \{1,2,3,4\}$$

$$\Theta_{<(2,3),(3)>} = \{1,2,3,4\} \qquad \Theta_{<(2,5),\perp>} = \{1,2,3\}$$

$\mathbf{Q_a}^{\mathbf{Trans}}$

a	$\mathscr{T}$
1	$\{1,2,3,4\}$
1	$\{1,2,3,4\}$
2	$\{1,2,3,4\}$
2	$\{1,2,3\}$

Figure 4.17: Transformation Provenance Representation Example

4.5 Relational Representation of Transformation Provenance Information

In this section we introduce a simplistic relational representation of *transformation* provenance. More user-friendly representations that are based on the SQL representation of queries will be presented in chapter 5. Recall that the *transformation* provenance of an algebra expression q contains one annotated algebra tree for each witness list in the *PI-CS data* provenance of q. These annotated trees all have the same nodes and edges; they only differ in their annotations functions θ_w. Therefore, we factor out the static part (that is the tree) in the relational representation of *transformation* provenance and only represent the annotation functions. Each annotation function θ_w is represented as the set of nodes from the algebra tree for which θ_w evaluates to 1. To simplify this representation identifiers for the nodes in an algebra tree are created by a pre-order traversal of the tree. We call the set representation of an annotation function θ_w the *annotation set* Θ_w.

Example 4.15. *In Figure 4.17 we reconsider the transformation provenance example from chapter 3. The algebra tree presented on the top right of this figure shows the generated node identifiers. These node identifiers are used in the set representations of the θ_w annotations functions for the transformation provenance of example query q_a. For instance, $\Theta_{<(2,5),\perp>}$ contains the identifiers for all nodes except the one for the base relation access S, because $\theta_{<(2,5),\perp} >$ evaluates to 0 for this node.*

Similar to the relational representation of *data* provenance we represent *transformation* provenance and the original result data in a single relation Q^{trans}. The annotation sets for the witness lists of a original result tuple are stored in a single additional attribute $\mathscr{T}$. Each tuple in Q^{trans} stores one original result tuple and one annotation set Θ_w for a $w \in \mathscr{DD}(q,t)$.

Definition 4.6 (Relational Transformation Provenance Representation)**.** *Let q be an algebra expression. The relational representation Q^{Trans} of the provenance of q according to the transformation provenance CS is defined as:*

$$Q^{Trans} = \{(t, \Theta_w)^m \mid t^p \in Q \wedge w^m \in \mathscr{DD}(q,t)\}$$

Example 4.16. *The relational representation ${Q_a}^{trans}$ of the transformation provenance of example query q_a is shown at the bottom of Figure 4.17. For instance, the last tuple in this relation represents the original result tuple (2) and the annotation set $\Theta_{<(2,5),\perp>}$ and, therefore, the set stored in the $\mathscr{T}$ attribute of this tuple includes all node identifiers except the one for the access to base relation S.*

Structural Rewrite

$$q^T = (R)^T = \Pi^B{}_{\mathbf{R},\mathscr{T}(q^T)\rightarrow\mathscr{T}}(R) \tag{T1}$$

$$q^T = (\sigma_C(q_1))^T = \Pi^B{}_{\mathbf{Q_1},\mathscr{T}(q^T)\rightarrow\mathscr{T}}(\sigma_C({q_1}^T)) \tag{T2}$$

$$q^T = (\Pi^{S/B}{}_A(q_1))^T = \Pi^B{}_{A,\mathscr{T}(q^T)\rightarrow\mathscr{T}}({q_1}^T) \tag{T3}$$

$$q^T = (\alpha_{G,agg}(q_1))^T = \Pi^B{}_{G,agg,\mathscr{T}(q^T)\rightarrow\mathscr{T}}(\alpha_{G,agg}(q_1) ⟕_{G=_n X} \Pi^B{}_{G\rightarrow X,\mathscr{T}}({q_1}^T)) \tag{T4}$$

$$q^T = (q_1 \diamond_C q_2)^T = \Pi^B{}_{\mathbf{Q_1},\mathbf{Q_2},\mathscr{T}(q^T)\rightarrow\mathscr{T}}({q_1}^T \diamond_C {q_2}^T) \tag{T5}$$

$$q^T = (q_1 \cup^{S/B} q_2)^T = \Pi^B{}_{\mathbf{Q_1},\mathscr{T}(q^T)\rightarrow\mathscr{T}}({q_1}^T \cup^{S/B} {q_2}^T) \tag{T6}$$

$$q^T = (q_1 \cap^{S/B} q_2)^T = \Pi^B{}_{\mathbf{Q_1},\mathscr{T}(q^T)\rightarrow\mathscr{T}}(q_1 \cap^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{T}}({q_1}^T) \bowtie_{\mathbf{Q_1}=_n\mathbf{Q_2}} {q_2}^T) \tag{T7}$$

$$q^T = (q_1 -^{S/B} q_2)^T = \Pi^B{}_{\mathbf{Q_1},\mathscr{T}(q^T)\rightarrow\mathscr{T}}(q_1 -^{S/B} q_2 \bowtie_{\mathbf{Q_1}=_n X} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{T}}({q_1}^T)) \tag{T8}$$

Transformation Provenance Attribute Rewrite

$$\begin{aligned}
\mathscr{T}(R^T) &= \{R\}\\
\mathscr{T}((\sigma_C(q_1))^T) &= \{\sigma_c(q_1)\} \cup \mathbf{Q_1}.\mathscr{T}\\
\mathscr{T}((\Pi^{S/B}{}_A(q_1))^T) &= \{\Pi^{S/B}{}_A(q_1)\} \cup \mathbf{Q_1}.\mathscr{T}\\
\mathscr{T}((\alpha_{G,agg}(q_1))^T) &= \{\alpha_{G,agg}(q_1)\} \cup \mathbf{Q_1}.\mathscr{T}\\
\mathscr{T}((q_1 \diamond_C q_2)^T) &= \{q_1 \diamond_C q_2\} \cup \mathbf{Q_1}.\mathscr{T} \cup \mathbf{Q_2}.\mathscr{T}\\
\mathscr{T}((q_1 \cup^{S/B} q_2)^T) &= \{q_1 \cup^{S/B} q_2\} \cup \mathbf{Q_1}.\mathscr{T}\\
\mathscr{T}((q_1 \cap^{S/B} q_2)^T) &= \{q_1 \cap^{S/B} q_2\} \cup \mathbf{Q_1}.\mathscr{T} \cup \mathbf{Q_2}.\mathscr{T}\\
\mathscr{T}((q_1 -^{S/B} q_2)^T) &= \{q_1 -^{S/B} q_2\} \cup \mathbf{Q_1}.\mathscr{T}
\end{aligned}$$

Figure 4.18: Transformation Provenance Rewrite Rules

4.6 Rewrite Rules for Transformation Provenance

We now present a meta-operator T for *transformation* provenance that transforms an algebra expression q into an algebra expression q^T that computes the relational representation Q^{Trans} of the *transformation* provenance of q. Like for *data* provenance this meta-operator is defined inductively over the structure of an algebra expression as rewrite rules for each algebra operator.

Definition 4.7 (Transformation Provenance Rewrite Meta-Operator)**.** *The transformation provenance rewrite meta-operator $T : \mathscr{E} \rightarrow \mathscr{E}$ is defined inductively over the structure of an input algebra expression q by applying the rewrite rules presented in Figure 4.18 to each operator in q.*

Fig. 4.18 presents the rewrite rules that implement T. For each rewrite rules the structural modification of the algebra expression and the computation of the new transformation provenance attribute $\mathscr{T}$ is presented separately. For the transformation provenance attribute rewrite we use the following notational conventions: $\{q\}$ denotes the node identifier of the top operator of q in the algebra tree of q. E.g., for the algebra expression $\sigma_C(R ⟕_{a=b} S)$ the expression $\{R ⟕_{a=b} S\}$ represents the node identifier 2. The $\cup$ used in the definition of the annotation sets is the normal set union operation except that we define $\mathscr{T} \cup \varepsilon = \mathscr{T}$.

The rewrite rule **(T1)** for a base relation access adds the singleton annotation set containing the node identifier of the base relation access $\{R\}$ as the value for attribute $\mathscr{T}$ to each generated result tuple. A selection is rewritten by rule **(T2)** by applying the unmodified selection to ${q_1}^T$ and adding an outermost projection that simply adds the node identifier of the selection operator to the annotation set of the rewritten input ${q_1}^T$. **(T3)**, the rewrite rule for projection, works analogously. An aggregation is rewritten **(T4)** by joining the rewritten input ${q_1}^T$ with the original aggregation and using a projection to add the node identifier for the aggregation to the annotation set of q^T.

$$q_a = \Pi^{S}{}_{a}(R \mathbin{\sqsupset\!\!\!\bowtie}_{b=c} S)$$

$$q_a{}^{T} = \Pi^{B}{}_{a,\{1\}\cup\mathscr{T}\rightarrow\mathscr{T}}(\Pi^{B}{}_{a,b,c,\{2\}\cup\mathbf{R}^{+}.\mathscr{T}\cup\mathbf{S}^{+}.\mathscr{T}\rightarrow\mathscr{T}}(\Pi^{B}{}_{a,b,\{3\}\rightarrow\mathscr{T}}(R) \mathbin{\sqsupset\!\!\!\bowtie}_{b=c} \Pi^{B}{}_{c,\{4\}\rightarrow\mathscr{T}}(S)))$$

$\mathbf{Q_a}^{\mathbf{T}}$

a	$\mathscr{T}$
1	{1,2,3,4}
1	{1,2,3,4}
2	{1,2,3,4}
2	{1,2,3}

Figure 4.19: Transformation Provenance Rewrite Example

The rewrite rule for join operators **(T5)** (here $\diamond$ denotes one of the algebra join operators) unions the annotation sets of the rewritten inputs and add the node identifier of the join to the result. Note that this is correct behavior for outer joins, because we have defined the union of a annotation set with ε as $\mathscr{T} \cup \varepsilon = \mathscr{T}$.

Rewrite rule **(T6)** for the union operator unions the rewritten inputs and uses a projection to union the annotation sets of both rewritten inputs with the node identifier of the union operator. In the transformation provenance attribute rewrite the $\mathscr{T}$ attribute of the input is referenced without using a qualification (e.g., $\mathbf{S}^{+}.\mathscr{T}$), because the result schema of the union operator is the schema of its the left hand input. **(T7)**, the rewrite rule for intersection works in a similar way as the *PI-CS* rewrite rule for this operator: the original intersection is joined with the rewritten left and right input on the original result attributes. The applied projection unions the annotation sets from both rewritten inputs with the node identifier of the intersection operator. A set difference is rewritten by **(T8)** using the same approach. For set difference the right input is not rewritten, because the *PI-CS* provenance of the right input is $\perp$. Hence, the annotation set of the right input is always the empty set.

Example 4.17. *Figure 4.19 shows the application of the T meta-operator to the example query q_a from Figure 4.17. In $q_a{}^{T}$ the node identifiers of the individual operators are added to the intermediate annotation sets produced by the rewritten inputs of the operator. Note that some node identifiers (1,2, and 3) are guaranteed to be contained in each annotation set produced by $q_a{}^{T}$. Thus, the rewritten query could be simplified to:*

$$\Pi^{B}{}_{a,\{1,2,3\}\cup\mathbf{S}^{+}.\mathscr{T}\rightarrow\mathscr{T}}(R \mathbin{\sqsupset\!\!\!\bowtie}_{b=c} \Pi^{B}{}_{c,\{4\}\rightarrow\mathscr{T}}(S))$$

Note that in the simplified query R is not rewritten at all. This kind of simplification is not specific to the example, but can be applied to a wide range of algebra expressions. We now prove the correctness and completeness of the rewrite rules and then discuss the simplification in detail.

Theorem 4.10 (Correctness and Completeness of the Transformation Provenance Rewrite Rules)**.** *For a algebra expression q the transformation provenance rewrite rules as presented in Figure 4.18 compute the relational representation of transformation provenance as defined in Definition 4.6:*

$$Q^{Trans} = Q^{T}$$

Proof.
To prove this theorem we use a modified version of the transformation provenance rewrite rules (denoted by q^{T+} that in addition to the annotation sets also propagate *PI-CS* witness list representations. The meta-operator implemented by the modified rewrite rules is called $T+$. The transformation provenance rewrite

rules use the same structural rewrites as the *PI-CS*. Therefore, the modified rewrites can be derived from the original $\mathscr{T}$ meta-operator rewrite rules by simply adding the provenance attribute list $\mathscr{P}$ to each outermost projection of an rewritten operator. E.g., the modified rule for projection is:

$$q^{T+} = (\Pi^{S/B}{}_{A}(q_1))^{T+} = \Pi^{B}{}_{A,\mathscr{T}(q^{T+})\rightarrow\mathscr{T},\mathscr{P}(q^{+})}({q_1}^{T+})$$

The modified versions allow us to reason about the witness list over which a $\mathscr{T}$ set is defined. Instead of the original equivalence $Q^{Trans} = Q^{T}$ we proof the equivalence $Q^{TransPI} = Q^{T+}$ where $Q^{TransPI}$ is defined as

$$Q^{TransPI} = \{(t, \Theta_w, w')^{m\times p} \mid t^p \in Q \wedge w^m \in \mathscr{DD}(q,t)\}$$

The only difference between Q^{Trans} and $Q^{TransPI}$ respective Q^{T} and Q^{T+} is that, in addition to the annotation sets, also the witness list is represented from which the annotation set is derived from. Therefore, the equivalence $Q^{Trans} = Q^{T}$ follows from $Q^{TransPI} = Q^{T+}$. Similar to the proof for *PI-CS*, the equivalence $Q^{TransPI} = Q^{T+}$ is proven in two steps. First the *result preservation* property of $T+$ is proven. Afterwards, we prove that for tuple $(t, w', x) \in Q^{T+}$, the set x is the annotation set Θ_w derived for witness list w (correctness), and that for every witness list $w \in \mathscr{DD}(q,t)$ the tuple (t, w', Θ_w) is in Q^{T+} (completeness)[4]. We refer to this property as *annotation set preservation*. From the proof of *PI-CS* we know that for each witness list $w \in \mathscr{DD}(q,t)$ there exists a tuple (t, w', x) in Q^{T+}. Hence, only the correctness part of the *annotation set preservation* has to be proven.

Result Preservation

The transformation provenance rewrite rules apply the same structural rewrites as the *PI-CS* rewrite rules. For the *PI-CS* rewrite rules we have proven that they fulfill the *result preservation* property. Therefore, this property is also fulfilled for the *transformation* rewrite rules.

Annotation Set Preservation

We prove the *annotation set preservation* by induction over the structure of an algebra expression q. We have to show that each result tuple in Q^{T+} is of the form (t, w', Θ_w).

Induction Start:

For $q = R$ each tuple in Q^{T+} is of form (t, w', T) with $w' = t$ and $T = \{R\}$. We have to show that $\{R\} = \Theta_w$. $\{R\}$ is contained in Θ_w if $[[R(w)]] = [[R(<t>)]] \neq \emptyset$ which is trivially fulfilled, because $[[R(<t>)]] = \{t^x\}$.

Induction Step: Given that the transformation provenance rewrite rules produce correct annotation sets for algebra expressions with a maximal nesting depth of n we have to prove that the same holds for algebra expressions with nesting depth $n+1$. Let op be an unary operator and q_1 be an algebra expression with maximal nesting depth $n+1$. We have to show that for (t, w', T) in $\left[\left[(op(q_1))^{T+}\right]\right]$ the following holds: $T = \Theta_w$.

Case $\sigma_C(q_1)$:

The rewrite rule for selection applies the unmodified selection and adds the annotation for the selection to the annotation set from ${Q_1}^{T+}$. Hence, each tuple from Q^{T+} is of form $(t, w', \{q\} \cup \Theta_v))$ where Θ_v is the annotation set of the witness list v in $\mathscr{DD}(q_1, t')$ from which w is derived from. To prove that $\{q\} \cup \Theta_v = \Theta_w$ we have to show that (1) Θ_w contains $\{q\}$ and that (2) $\Theta_w \cap OP(q_1)$ agrees with $\Theta_v \cap OP(q_1)$. Here $OP(q_1)$ denotes the node identifiers for all operators in q_1. If w is a witness list for q then $[[q(w)]] \neq \emptyset$. Thus, the first property is fulfilled. From the compositional semantics of *PI-CS* we know that $w = v$. Therefore, the property (2) is fulfilled, because each $[[sub_{op}(w)]] = [[sub_{op}(v)]]$ for op in q_1.

Case $\Pi^{S/B}{}_{A}(q_1)$:

Case $\alpha_{G,agg}(q_1)$:

[4] Recall that w' denotes the relational representation of witness list w (see section 4.1).

As for selection the rewrite rules for projection and aggregation produce tuples of the following form $(t, w', \{q\} \cup \Theta_v)$. Property (1) and (2) holds for the same reason as for selection.

For binary operators we can use the same reasoning as for unary operators to prove property (1) (the node identifier of op is included in the annotation set of op). For the proof of property (2) for binary operators the following cases have to be considered (based on the three cases in the definition of transitivity for *PI-CS*):

1. The witness list $w = v_1 \blacktriangleright v_2$ for v_1 and v_2 being the witness lists for q_1 respective q_2 from which w is derived trough transitivity.
2. The witness list $w = v_1 \blacktriangleright\!<\!\perp, \ldots, \perp\!>$ for v_1 being the witness lists for q_1 from which w is derived trough transitivity.
3. The witness list $w =\,<\!\perp, \ldots, \perp\!>\!\blacktriangleright v_2$ for v_2 being the witness lists for q_2 from which w is derived trough transitivity.

Case $q_1 \times q_2$:
Case $q_1 \bowtie_C q_2$:
Case $q_1 \cap^{S/B} q_2$:

For cross product, join, and intersection only the first case applies and the proof is analog to the proof for unary operators, because each of these operators combines the Θ_{v_1} and Θ_{v_2} sets.

Case $q_1 ⟕_C q_2$:

For left outer join the first and the second case apply. The first case applies for tuples which fulfill the join condition. Therefore, the annotation set construction $(\{q_1 ⟕ q_2\} \cup \mathbf{Q_1}.\mathscr{T} \cup \mathbf{Q_2}.\mathscr{T})$ applied by the rewrite rule generates the correct set. The second case applies for tuples t that do not fulfill the join condition. In this case all parts of w that correspond to q_2 are set to $\emptyset$. It follows that $[[q_2(w)]] = \emptyset$. This means in Θ_w does not include any node identifiers from $OP(q_2)$. This correctly modeled in the rewrite rule, because t does not fulfill the join condition and, therefore, attribute $\mathbf{Q_2}.\mathscr{T}$ is ε (Recall that we use ε is an alternative representation of the empty set).

Case $q_1 ⟖_C q_2$:
Case $q_1 ⟗_C q_2$:
Case $q_1 \cup^{S/B} q_2$:

The proof for right outer join, full outer join, and union are analog to the proof of left outer join.

Case $q_1 -^{S/B} q_2$:

For set difference only the second case applies. The rewrite rule discards the annotation set for q_2. Hence, no node identifiers from $OP(q_2)$ are included in the resulting annotation set.

□

4.6.1 Rewrite Rules Simplification

Recall that in example 4.17 we presented a simplification of the rewritten example query ${q_a}^T$ that uses the fact that some node identifiers are guaranteed to be in the annotation sets of ${Q_a}^T$. In general a node identifier for an operator op is guaranteed to be in the annotation set of a rewritten query if computing $[[sub_{op}(w)]]$ for a witness list $w \in \mathscr{DD}(q,t)$ for $t \in Q$ never returns the empty set. We use this fact to identify criteria for applying the presented simplification. According to the compositional semantics of *PI-CS* the only operators that include $\perp$ in witness lists are the outer joins, union, aggregation, and set difference. According to the transitivity of *PI-CS* the only possibility $\perp$ can occur in a witness list of a query q is that one of the aforementioned operators is used in q. From the definition of *PI-CS* (condition 2) and the definition of the algebra operators we can deduce that the evaluation of q over a witness list will never result in the empty set if none of the $\perp$ generating operators is used in q. Even though aggregation can have witness lists that contain $\perp$, evaluating this operator over $<\!\perp\!>$ does not produce the empty set. This means queries involving only operators that never include $\perp$ in their witness lists or aggregation (which we refer to as *transformation static*) are rewritten by simply adding a projection to the query that generates the static annotation set for this query. By static we mean that the annotation set is independent of the input data. Even more, if a query includes non-static operators, a sub-expression that contains only static operators can be rewritten by adding the annotation set trough a projection.

Proposition 4.1 (Transformation Provenance Rewrite Simplification)**.** *An algebra expression that contains solely static operators has static transformation provenance. For an algebra expression q, all subexpressions that are static and do not have a non-static operator as their ancestor in the algebra tree of q or are only in the left/right sub-tree of left/right outer joins have static transformation provenance.*

4.7 Summary

In this chapter we introduced relational representations for provenance according to the *contribution semantics* presented in chapter 3 and demonstrated how to generate these representations by evaluating rewritten algebra expressions. Several meta-operators were discussed that transform an algebra expression q into a rewritten form that computes the relational representation of a type of provenance for q. For each of the meta-operators we proved that the rewritten algebra expressions generated by this operator compute the corresponding relational representation of provenance. For *PI-CS* provenance of algebra expressions with sublinks we have introduced several rewrite strategies that utilize un-nesting and de-correlation techniques to rewrite these queries. Furthermore, for *C-CS* and *transformation* provenance we presented simplifications for the rewrites and demonstrated when they can be applied. In summary, we have theoretically sound algorithms for computing relational provenance representations according to several *CS* types. The relational representations allow us to store provenance in a relational database and query it using *SQL*. Which is a huge advantage over existing approaches that do not support querying of provenance information at all or supply only very limited query capabilities. Modeling provenance computation as algebraic rewrites has the intrinsic advantage that provenance computations can be seamlessly integrated into *SQL* which is not the case for other provenance approaches, because they usually develop a new language for provenance computation and querying. Note that even though these languages may be implemented as rewrites, in general it is not easily possible to integrate them into SQL. This is due to the fact that the applied provenance representation are not relational and are produced by post-processing the result of rewritten queries. In the next chapter we will demonstrate how to integrate the *Perm* provenance representation and computation into a relational *DBMS*.

Chapter 5

Implementation

In the previous chapters we introduced the theoretical background of our approach for provenance computation in relational databases. In this chapter we present the implementation of the *Perm* system as an extension of *PostgreSQL* [Mom01]. While the theoretical treatment already lays out some parts of the implementation, implementing a complete provenance system is far from trivial and, thus, is one of the major contributions of this thesis. As discussed in the previous chapters our goal is to add provenance support as an orthogonal extension of the relational model. Hence, we first present the additional constructs that we added to SQL to achieve this goal (section 5.1). To be able to implement the provenance rewrite rules in a relational system, they had to be translated from relational algebra into SQL. These translations are presented in section 5.2. Section 5.3 presents the architecture of *Perm*. The extension to *PostgreSQL* standard functionality that were necessary to seamlessly integrate provenance computation into *PostgreSQL* are discussed in section 5.4. Afterwards, we present the *Perm* module, the core of the system, that implements the translated rewrite rules and discuss optimizations which are applied to increase the performance of provenance computation (section 5.5). The *Perm-Browser*, a graphical user interface for *Perm* is presented in section 5.6.

shops

name	numEmpl
Merdies	3
Joba	14

sales

sName	itemId
Merdies	1
Merdies	2
Merdies	2
Joba	3
Joba	3

items

id	price
1	100
2	10
3	25

```
qa = SELECT s.sName, i.price FROM sales s JOIN items i ON s.itemId = i.id;
```

$$q_a = \Pi^B_{sName,price}(sales \bowtie_{itemId=id} items)$$

Figure 5.1: Provenance Attribute Naming Example

5.1 SQL Provenance Language Extensions

In chapter 4 we have introduced a relational representation of provenance information and demonstrated how this representation can be generated by evaluating rewritten relational algebra expressions. Recall that this approach has several advantages over alternative approaches. One major advantage is that this approach enables us to seamlessly integrate provenance computation in a relational query language. The goal we want to achieve with the *SQL-PLE* (*Provenance Language Extension*) is exactly that: An orthogonal language extension. E.g., it should be possible to combine standard SQL constructs with the SQL-PLE constructs in a single SQL statement. *SQL-PLE* contains language constructs that trigger provenance computation, limit the scope of a provenance computation, handle external provenance, and several additional constructs for debugging and investigating the inner workings of the system.

5.1.1 Provenance Attribute Naming Scheme

Recall that for the relational representation of provenance according to a *CS* type we introduced an attribute renaming function that generates unique names for provenance attributes. Until now we have not presented how these unique names are generated. To enable a user to query provenance data, a mechanism for addressing provenance attributes in a query is needed. Thus, the names generated for provenance attributes should not only be unique, but the naming scheme should produce predictable and self-explanatory names.

In the naming scheme applied by *Perm* each provenance attribute name consists of the fixed prefix *prov* followed by the name of the base relation, the attribute is derived from, and the original attribute name. Each part of a provenance attribute name is separated by an underline character. If a relation is referenced more than once in a query, an identifying number is attached to the relation name. To guarantee uniqueness, underline characters in original attribute names are escaped (replaced by two underscore characters).

For example, consider SQL query q_a (and its algebra representation) presented in Figure 5.1. This query accesses base relations *sales* and *items* (in this order). Applying the naming scheme generates the following schema for the relational representation of the provenance of q_a:

```
sName, price,
prov_sales_sName, prov_sales_itemId,
prov_items_id, prov_items_price
```

This naming scheme is predictable, because the attribute names can be derived from the structure of an SQL query. It is also self-explanatory, because the generated names identify the base relation and original attribute from which a provenance attribute is derived from. To keep the following examples concise we use the prefix *p* for provenance attribute names instead of the names generated by the naming scheme (e.g., *pName* instead of *prov_shops_name*).

5.1.2 Provenance Computation

A user can request provenance computation for an SQL query q in *SQL-PLE* by adding the keyword *PROVENANCE* to the select clause of q. A query q that is marked by this keyword is substituted with a query that computes the relational representation of the provenance of q. For example:

```
SELECT PROVENANCE name FROM shops;
```

The default is to compute provenance according to *PI-CS*. If another *data* provenance *CS* type is desired the optional *ON CONTRIBUTION* parameter can be used to indicate this. For instance, if the *CDC-CS* provenance of the example query presented above should be computed this is expressed in *SQL-PLE* as:

```
SELECT PROVENANCE ON CONTRIBUTION (COPY COMPLETE NONTRANSITIVE) name
FROM shops;
```

As mentioned before one main goal in the design of *SQL-PLE* was to fully integrate provenance computation into the SQL language. Therefore, a provenance computation can be applied in any place where the use of a standard *SELECT* statement is allowed. One important benefit of this integration is the ability to ask queries over provenance information by using a provenance query as a sub-query in a standard *SELECT* statement. For example, consider the following query q_a that computes the total revenue for each shop:

```
SELECT name, sum(price) AS sum
FROM shops, sales, items
WHERE name=sName AND itemId=id
GROUP BY name;
```

If a user wants to know which revenues were influenced by a sale of an item with a price greater than 50, this query can be expressed in *SQL-PLE* as:

```
SELECT name, sum, prov_items_id
FROM
    (SELECT PROVENANCE name, sum(price) AS sum
    FROM shops, sales, items
    WHERE name=sName AND itemId=id
    GROUP BY name) AS prov
WHERE prov_items_price > 50;
```

Note that the use of *SQL-PLE* constructs is not limited to *SELECT* queries. For example, the following *SQL-PLE* statement would store the result of a provenance computation as a view:

```
CREATE VIEW provview AS
    SELECT PROVENANCE sum(i.price)
    FROM items;
```

SQL-PLE contains three keywords for computing the transformation provenance of a query that are applied in the same way as the *PROVENANCE* keyword: *TRANSPROV*, *TRANSSQL*, and *TRANSXML*. All three keywords compute the same provenance information, but each uses a different result representation. These representation are discussed in detail in section 5.5.4.

5.1.3 Limited Provenance Scope

The huge amount of provenance information produced for complex queries over large database instances can be overwhelming for a user. Asking queries over the generated provenance information reduces the complexity and can be used to focus on interesting parts of the provenance or *transformation* provenance can be used to gain a better understanding of the query itself. However, this does not limit the complexity of the original query that has to be understood. In such cases it would be helpful to be able to compute provenance according to an intermediate result used in the query instead of tracing provenance to the base relations accessed by the query. This functionality is supported by *SQL-PLE* through the *BASERELATION* keyword. Appending the keyword *BASERELATION* to a *FROM* clause item instructs *Perm* to handle this *FROM* clause item as a base relation. For example, for the query presented below the result of the

Construct	Description
PROVENANCE	Marks a query for provenance computation.
ON CONTRIBUTION (cs_type)	Instructs *Perm* to use a certain *CS* type.
BASERELATION	Handle a *FROM* clause item as if it were a base relation.
PROVENANCE (attr_list)	Handle attributes from attr_list as provenance attributes.
TRANSPROV/TRANSSQL/TRANSXML	Mark a query for transformation provenance computation.
EXPLAIN SQLTEXT	Return the (rewritten) SQL text of a query.
EXPLAIN GRAPH	Return an algebra tree for a query (as a dot-language script).

Figure 5.2: SQL-PLE language constructs

provenance computation would include the tuples from the result of *FROM* clause item *sub* that contributed to a result tuple of the query instead of the tuples from the base relation *items*.

```
SELECT PROVENANCE total * 10 FROM
 (SELECT sum(price) AS total FROM items) BASERELATION AS sub;
```

5.1.4 Support for External Provenance

A user may store data in *Perm* that carries provenance information which has been created manually or by another provenance management system. We call such provenance information *external*. *SQL-PLE* contains language constructs that can be used to make *Perm* aware of external provenance and handle it as if it was created by *Perm* itself. External provenance may be either stored in the same relation as the data it refers too or in separate relation(s). To not impose any restrictions on the schema used to store external provenance information, *SQL-PLE* enables a user to generate a relation that contains the original data alongside with its provenance and then inform the system about which attributes store provenance information. A user can define that a subset of a *FROM* clause item's attributes are provenance attributes by appending PROVENANCE (*attrlist*) to a *FROM* clause item. *Perm* is instructed by the PROVENANCE clause to not rewrite the *FROM* clause item this keyword is appended to and handle the information stored in the provenance attributes identified by this clause as if they contain provenance information generated by *Perm*.

For example, assume a relation *provItem* stores information about the provenance of tuples in the *items* relation from the example database. An attribute *itemId* is used to store the association between an item and its provenance. In addition, *provItem* contains a *dbName* attribute that contains the name of the original database from which the information about an item is imported from. The example query below demonstrates how to make *Perm* aware of the provenance information stored in relation *provItem*:

```
SELECT PROVENANCE sum(price) AS total
FROM
    (SELECT *
    FROM items JOIN provItem ON (id = itemId)) PROVENANCE (dbName) AS provItem;
```

Note that implementing support for external provenance and limited scope in *Perm* is simplified to a great extend by the recursive nature of the query rewrite rules developed for *Perm*.

5.1.5 Debugging and Convenience Language Constructs

SQL-PLE contains language constructs for debugging *Perm* and inspecting the query rewrites applied by the system. *EXPLAIN SQLTEXT* returns the SQL text of an SQL query. This construct can be used to investigate the rewrites applied by the *Perm* system, because provenance constructs are rewritten before the SQL text is returned. Similar *EXPLAIN GRAPH* returns the algebra tree of a query[1]. As an example application of these constructs consider the following statement:

[1] Actually this construct returns a script in the dot language (see [GN00]), a graph description language. The graphviz tools can be used to visualize the graph.

```
EXPLAIN SQLTEXT SELECT PROVENANCE * FROM shops;
```

5.1.6 Overview of the SQL-PLE language constructs

Figure 5.2 presents an overview of the *SQL-PLE* language constructs introduced in this section. An grammar for an excerpt of the *SQL-PLE* language is given in appendix A. In contrast to other approaches for relational provenance computation, *SQL-PLE* has the following advantages:

- Provenance computation for a larger subset of SQL constructs (e.g., sublinks).
- Full integration in SQL.
- Support for multiple *CS* types and *transformation* provenance.
- Support for external provenance.
- Ability to limit the scope of provenance computations.

$$\begin{tikzcd}
\textbf{SQL Query} \arrow[r, "\text{SQL Rewrite}", "\mathbf{(2)}"'] \arrow[d, "\text{Translation to Algebra}"', "\mathbf{(1)}"] & \textbf{Rewritten SQL Query} \\
\textbf{Algebra Expression} \arrow[r, "\text{Algebraic Rewrites}", "\mathbf{(1)}"'] & \textbf{Rewritten Algebra Expression} \arrow[u, "\mathbf{(1)}", "\text{Translation to SQL}"']
\end{tikzcd}$$

Figure 5.3: Translation of Algebraic Rewrites into SQL Rewrites

5.2 Rewrite Rules Translation

A translation between SQL and relational algebra is needed to be able to implement the rewrite rules developed in chapter 4. Given such a translation, an SQL query can be rewritten by either **(1)** translating it into relational algebra, applying the algebraic rewrites, and translating it back to SQL, or by **(2)** using the mapping between relational algebra and SQL to translate the rewrite rules to SQL. Figure 5.3 illustrates this process. The translation of the rewrite rules to SQL is more complex, but it has the advantage that no translation between algebra and SQL has to be applied at run-time (The downward and upward arrows in Figure 5.3). Therefore, we have chosen to implement the second approach. We first introduce an canonical translation between SQL and relational algebra. Afterwards, types of queries are classified regarding their provenance behaviour. Finally, we present how each query type is rewritten.

5.2.1 A Canonical Translation between SQL and Relational Algebra

We present the canonical translation between SQL and relational algebra in both directions. First from SQL to algebra expressions and afterwards from algebra expression to SQL. Using common terminology we refer to each SELECT ... FROM ... WHERE ... GROUP BY ... HAVING part in an SQL statement as a query block. Sub-queries in the *FROM* clause of an SQL statement are separate query blocks. A query block represents either a tree of set operations or an *ASPJ*-query.

5.2.1.1 Translation from SQL to Algebra

We translate an *ASPJ* query into the following algebra expression, where some of the operators may be left out if their corresponding part is not present in the SQL-query.

$$\Pi^{S/B}{}_{A_1}(\sigma_H(\alpha_{G,agg}(\Pi_{A_2}(\sigma_C(q_1 \diamond_{C_1} \dots \diamond_{C_n} q_n)))))$$

The *FROM* clause of the query is translated into a list of joins ($\diamond$ represents one of the join operators of the algebra). Sub-queries in the *FROM* clause are recursively transformed using the same approach. The *WHERE* clause is represented by σ_C and the expressions of the *SELECT* clause are handled in $\Pi^{S/B}{}_{A_2}$. If the query block contains aggregation functions, then the aggregation is transformed into $\alpha_{G,agg}$. Projections on aggregation functions (e.g., $sum(a)+2$) are modeled as another projection $\Pi^{S/B}{}_{A_1}$ and σ_H represents the *HAVING* clause. For query blocks that use *DISTINCT* the duplicate removing version of projection is used as the outermost projection operator in the resulting algebra expression. Otherwise the duplicate preserving version is applied. *LIMIT*, *ORDER BY*, and *OFFSET* clauses do not have an algebraic counterpart. The algebra was deliberately defined to not include these operators, because they are either non-deterministic (*LIMIT* and *OFFSET*) or produce an output that cannot be modeled without extending the relational model with an order over the tuples in a relation (*ORDER BY*). We will demonstrate later in this section that it is nonetheless possible to generate useful provenance information for queries containing these clauses. As an example for the translation between *ASPJ* query blocks and relational algebra expressions consider the following SQL query:

```
SELECT sum(R.a) + 2 AS one, avg(T.c * 50) AS two
FROM R JOIN S ON (a = b), T
WHERE T.c = S.b
GROUP BY S.b
HAVING count(*) > 5;
```

This query is translated into the following algebra expression:

$$\Pi^{B}{}_{sum(a)+2\rightarrow one,avg(x)\rightarrow two}(\sigma_{count(*)>5}(\alpha_{b,sum(a),avg(x),count(*)}(\Pi^{B}{}_{a,b,c,c\times 50\rightarrow x}(\sigma_{c=b}((R \bowtie_{a=b} S) \times T)))))$$

A query block that represents a tree of set operations is translated into an algebra expression by replacing each SQL set operation with its algebraic counterpart. E.g., *UNION ALL* is translated into $\cup^{B}$ and *INTERSECT* is translated into $\cap^{S}$. For instance, the following query

```
(SELECT * FROM R
UNION
SELECT * FROM S)
INTERSECT ALL
SELECT * FROM T;
```

is translated into:

$$(R \cup^{S} S) \cap^{B} T$$

If a query block contains sublinks, these are translated into the corresponding sublink expression in the algebra. Recall that in the *Perm* algebra sublink expression are only allowed in selection predicates and projection expressions. In contrast the SQL standard allows sublinks to be used in any place where expressions are applicable. For instance, in group-by expressions, the *HAVING* clause, or join conditions. Sublink expressions in aggregation functions, *GROUP BY* clause, or *HAVING* clause are translated using the approach for aggregation presented above (details are given in section 5.2.7). E.g., the following query

```
SELECT sum(R.a) * (SELECT sum(b) FROM S) AS x
FROM R;
```

is translated into:

$$\Pi^{B}{}_{sum(a)\times(\alpha_{sum(b)}(S))\rightarrow x}(\alpha_{sum(a)}(R))$$

Sublinks in join conditions are translated into algebra by modeling the join as a cross product followed by a selection. For outer joins algebraic equivalences have to be applied to transform the outer join into an inner join.

5.2.1.2 Translation from Algebra to SQL

A simple translation from the *Perm* algebra into SQL is to create a query block for each operator in the algebra expression and add the SQL representation of the inputs of the operator to the *FROM* clause of the resulting query block. This approach is straightforward, but it creates an unnecessary large number of query blocks. Therefore, adjacent operators in the algebra expression are modeled as a single SQL query block if possible. Adjacent set operations are merged into a single query block. Furthermore, adjacent join operators are integrated into a single *FROM* clause. Projections and selections above a join operation are integrated into the query block of the join operation. The same applies for aggregation. Note that only a single aggregation operator per query block is allowed. For example, consider the algebra expression present below:

$$\alpha_{avg(suma)}(\Pi^{B}{}_{sum(a)\rightarrow suma,b}(\alpha_{b,sum(a)}(R \times S)))$$

This expression is translated into:

```
SELECT avg(suma)
FROM
    (SELECT sum(a) AS suma, b
    FROM R,S
    GROUP BY b) AS sub;
```

5.2.2 Query Block Types

We now classify types of query blocks in a query tree according to the algebra expression they represent and their provenance. Below we present each identified block type with an example in SQL.

SPJ (Select-Project-Join): A query block that does not contain aggregation, set operations or sublinks is equivalent to an algebra sequence of the format:

$$\Pi_A(\sigma_C(q_1 \diamond \ldots \diamond q_n))$$

The association of joins is defined by brackets (not included in the representation to keep the expression simple). As usual $\diamond$ represents one of the join operators of the algebra.

```
SELECT *
FROM R LEFT JOIN (S NATURAL JOIN T)
   ON (R.a = S.b);
```

ASPJ (Aggregate-Select-Project-Join): If a query block contains aggregation functions, a *GROUP BY* clause and/or a *HAVING* clause, this block is equivalent to an algebra expression of the following format:

$$\Pi_{A_1}(\sigma_H(\alpha_{G,agg}(\Pi_{A_2}(\sigma_C(q_1 \diamond \ldots \diamond q_n)))))$$

```
SELECT sum(R.a) FROM R GROUP BY R.b;
```

SET (Set operations): As mentioned before a list of set operations is represented as a single query block. A *SET* query block is equivalent to an algebra expression of the following format where $\bullet$ denotes one of the set operators of the algebra (brackets are left out to keep the expression simple):

$$q_1 \bullet \ldots \bullet q_n$$

```
SELECT * FROM R UNION
    (SELECT * FROM S INTERSECT SELECT * FROM T);
```

SPJ-sub (Sublinks in Projection or Selection): This type of query block is an *SPJ* query block that contains sublink queries in the *SELECT* clause, join predicates or *WHERE* clause expressions. Except for these sublinks the algebra expression for an *SPJ-sub* query block is the same as for an *SPJ* query block.

```
SELECT (SELECT sum(S.b) FROM S) AS sums, R.a
FROM R;
```

ASPJ-sub (Sublinks in Aggregation): An *ASPJ-sub* block is a query block with aggregation and sublink queries in the *SELECT*, *GROUP BY*, and/or *HAVING* clause.

```
SELECT avg(R.a) FROM R
GROUP BY R.a IN (SELECT S.b FROM S)
```

5.2.3 Rewrite of SPJ Query Blocks

The general approach we follow for translating the algebraic rewrite rules to SQL is to first transform a query block QB into an algebra expression q, apply the rewrite rules for one of the provenance meta-operators to generate, e.g., q^+, and finally transform q^+ back into SQL, thus, generating a rewritten SQL statement QB^+. In the implementation of the rewrite rules the direct translation from QB to QB^+ is applied (The horizontal arrow from the SQL query to the rewritten SQL query in Figure 5.3).

5.2.3.1 PI-CS Rewrites

The algebra expression q of an *SPJ* query block consists of a single projection and selection applied to the result of join operations over sub-queries and base relations. For *PI-CS* such algebra expression is rewritten by applying rewrite rules (R1), (R2), (R3), and (R5.a-R5.e). Except for adding projections and replacing duplicate removing projections with duplicate preserving projections, these rewrite rules do not alter the structure of q. The rewrite rules for joins (R5.a-R5.e) add an additional projection to the rewritten algebra expression to generate a correct order of result attributes. In q^+ this projection can be omitted because the rewritten projection operator in q^+ guarantees a correct attribute order. If one of the inputs q_i to q (the *FROM* clause items in the SQL query block) is a base relation access, rewrite rule (R1) transforms q_i into a projection that generates the provenance attributes for the base relation. This projection can be merged into the rewritten projection operator. We denote the modified provenance attribute list in this projections as $\mathscr{P}'$.

Example 5.1. *For instance, the result of applying the* $+$ *meta-operator to algebra expression* $q = \Pi_A(R \times S)$ *over relations* R *and* S *with schemas* $\mathbf{R} = (a)$ *and* $\mathbf{S} = (b)$ *is:*

$$q^+ = \Pi^B{}_{a,\mathcal{N}(a),\mathcal{N}(b)}(\Pi^B{}_{a,b,\mathcal{N}(a),\mathcal{N}(b)}(\Pi^B{}_{a,a\to\mathcal{N}(a)}(R) \times \Pi^B{}_{b,b\to\mathcal{N}(b)}(S)))$$

Merging the projections in q^+*, the query is transformed into:*

$$q^+ = \Pi^B{}_{a,a\to\mathcal{N}(a),b\to\mathcal{N}(b)}(R \times S)$$

For an arbitrary algebra version q of an *SPJ* query block the resulting simplified version of q^+ is as follows:

$$q^+ = \Pi_{A,\mathscr{P}'(q^+)}(\sigma_C(q_1{}^+ \diamond_{C_1} \dots \diamond_{C_n} q_n{}^+)$$

Using the translation between algebra and SQL, q^+ is translated into a single SQL query block QB^+. Of course, inputs q_i that are not base relation accesses are not merged into this query block. The result of translating q^+ into SQL is:

SELECT A, $\mathscr{P}'(q)$
FROM $q_1{}^+$ ... $q_n{}^+$
WHERE C;

Thus, an *SPJ* query block QB is rewritten as follows:

QB		QB^+

SELECT A
FROM q_1 ... q_n
WHERE C
	$\longrightarrow$	SELECT A, $\mathscr{P}'(q)$
		FROM $q_1{}^+$ ... $q_n{}^+$
		WHERE C;

5.2.3.2 C-CS Rewrites

The *C-CS* and *transformation* provenance rewrite rules use a data-type that stores a set of integer number or sets of attribute names. *PostgreSQL* does not natively support set data-types, but sets of integers can be modeled using the build-in variable length *bit-vector* data-type. A bit-vector of length n is used to represent a subset of the set $\{1,\dots,n\}$ by using a 0 respective 1 in the ith position to indicate if i is included in the set. The union and intersection operations applied in the rewrite rules are modeled as the build-in *bit-wise-or* and *bit-wise-and* operations of the bit-vector data-type. The *C-CS* rewrite rules also use the $if\ (e_1)\ then\ (e_2)\ else\ (e_3)$ expression. This expression is represented in SQL using the *CASE* construct.

For the *C-CS* types the algebra expression q for an *SPJ* query block *QB* is rewritten into q^C using rewrite rules (C1), (C2), (C3), and (C5.a-C5.e). As for the *PI-CS* rewrite rules the additional projections added by these rules can be merged into a single outermost projection (CM' denotes the merged copy expressions). The resulting algebra expression is presented below.

$$q^C = \Pi_{A,\mathscr{P}'(q^C),CM'(q)}(\sigma_C(q_1{}^C \diamond_{C_1} \dots \diamond_{C_n} q_n{}^C)$$

Recall that an algebra expressions is rewritten according to a *C-CS* type by applying the $\mathscr{P}^*$ inclusion expressions to q^C to generate the final witness list representations from the information stored in the $\mathscr{C}$ attributes. Thus, if *QB* is the outermost query block in the SQL query, then the inclusion expressions have to be applied to *QB* to create the rewritten version $q^{CD/CT/PD/PT}$. This final projection can be merged with the outermost projection of q^C too. Similar to the rewrite for *PI-CS* the rewritten algebra statement q^C can be translated into a single query block QB^C. The resulting SQL rewrite is as follows:

QB		QB^C
SELECT A		SELECT A, $\mathscr{P}'(q^C)$, $CM'(q)$
FROM q_1 ... q_n	$\longrightarrow$	FROM $q_1{}^+$ ... $q_n{}^+$
WHERE C		WHERE C;

5.2.3.3 Transformation Provenance Rewrites

The *transformation* provenance rewrite rules for the operators applied in the algebraic version q of an *SPJ* query block *QB* are (T1), (T2), (T3), and (T5). These rewrite rules only modify q by adding additional projections that manipulate the $\mathscr{T}$ attribute that is used to store the *transformation* provenance information. As for *PI-CS*, these projections can be merged into a single outermost projection. We denote the merged expressions for $\mathscr{T}(q^T)$ as $\mathscr{T}'(q^T)$. The resulting simplified version of q^T is:

$$q^T = \Pi_{A,\mathscr{T}'}(\sigma_C(q_1{}^T \diamond_{C_1} \dots \diamond_{C_n} q_n{}^T)$$

As for the *data* provenance *CS* types, the rewritten algebra expression can be translated into a single query block. The SQL rewrite rule that is derived from this translation is presented below.

QB		QB^T
SELECT A		SELECT A, $\mathscr{T}'$
FROM q_1 ... q_n	$\longrightarrow$	FROM $q_1{}^+$ ... $q_n{}^+$
WHERE C		WHERE C;

Example 5.2. *As an example for the PI-CS SQL rewrite consider the following query over base relations R and S with schemas* $\mathbf{R} = (a,b)$ *and* $\mathbf{S} = (c)$*:*

```
SELECT R.a * 2 AS two
FROM R, S
WHERE R.a = S.c
```

Applying the SQL rewrite for PI-CS this query is rewritten into:

```
SELECT R.a * 2 AS two, R.a AS prov_R_a, R.b. AS prov_R_b, S.c AS prov_S_c
FROM R, S
WHERE R.a = S.c
```

In the rewritten query the projections of the rewritten base relation accesses are merged into the SELECT clause.

5.2.4 Rewrite of ASPJ Query Blocks

An *ASPJ* query block is equivalent to an algebra expression q that applies a projection and selection (the *HAVING* clause) to the result of an aggregation. Recall that all rewrite rules for aggregation use a join between the original aggregation and its rewritten input, because it is not possible to propagate provenance directly trough an aggregation.

5.2.4.1 PI-CS Rewrites

The algebraic translation q of an *ASPJ* query block *QB* is rewritten using rewrite rules (R1), (R2), (R3), (R4), and (R5.a-e). Let q_1 denote the input of the aggregation operator applied in q. q_1 is the relational representation of an *SPJ* query and, thus, rewritten as presented in the discussion of *SPJ* query blocks. In q^+ the projections added by the rewrite rules for projection (R3) for the outer projection $\Pi^{S/B}{}_{A_1}$ and aggregation (R4) can be combined into a single projection. The simplified version of q^+ is as follows:

$$q^+ = \Pi^B{}_{A,\mathscr{P}(q^+)}(q ⟕_{G=G'} \Pi^B{}_{G\to G',\mathscr{P}'(q_1{}^+)}(q_1{}^+))$$

q^+ cannot be translated into a single query block, because it is not possible to compute an aggregation function and join its result with another query in one query block. Therefore, q^+ is translated into three query blocks. Query block *QB*, a query block that contains the translated version of ${q_1}^+$, and a query block that models the outermost projection in q^+ and the join between ${q_1}^+$ and q. In SQL the group-by attributes used in an aggregation are not necessarily present in the result schema of the query. Therefore, in the translation of q^+ to SQL we add omitted group by attributes to the *SELECT* clause in *QB*. This modified version of *QB* is denoted as *QB'*. The projection over ${q_1}^+$ used in q^+ to rename the group by attributes can be omitted in SQL, because using the same attribute names in difference *FROM* clause items is unproblematic. The resulting SQL rewrite is as follows. Note that we use notations like "A = B" and "A AS B" as abbreviations for comparisons or re-naming of attribute lists.

```
                                         QB+
            QB                    SELECT orig.A, sub.𝒫'(q)
                                  FROM
  SELECT A                               QB' AS orig
  FROM q1 ... qn                    LEFT JOIN
  WHERE C                ⟶              (SELECT G, 𝒫'(q)
  GROUP BY G                             FROM q1+ ... qn+
  HAVING H                               WHERE C) AS sub
                                    ON orig.G = sub.G;
```

5.2.4.2 C-CS and Transformation Provenance Rewrites

The *C-CS* and *transformation* provenance rewrites rules for aggregation use the same approach to propagate provenance information by joining the original aggregation with its rewritten input as in the *PI-CS* aggregation rewrite rule. They only differ from the *PI-CS* rule in the type of provenance information they propagate. Therefore, the same simplifications can be applied to the rewritten algebra statement as for *PI-CS*. The resulting simplified versions of q^C and q^T are:

$$q^C = \Pi^B{}_{A,\mathscr{P}'(q^C),CM'(q)}(q \,⟕_{G=G'} \Pi^B{}_{G\rightarrow G',\mathscr{P}'({q_1}^C),\mathscr{C}(q_1)}({q_1}^C))$$
$$q^T = \Pi^B{}_{A,\mathscr{T}'(q^T)}(q \,⟕_{G=G'} \Pi^B{}_{G\rightarrow G',\mathscr{T}'({q_1}^T)}({q_1}^T))$$

Using the same approach as for *PI-CS* we translate the rewritten algebra expression into three query blocks:

QB

```
SELECT A
FROM q1 ... qn
WHERE C
GROUP BY G
HAVING H
```

$\longrightarrow$

$QB^{C/T}$

```
SELECT orig.A, sub.X
FROM
        QB' AS orig
    LEFT JOIN
        (SELECT G, X
        FROM q1^C/T ... qn^C/T
        WHERE C) AS sub
    ON orig.G = sub.G;
```

In the SQL rewrite X denotes $\mathscr{P}'(q),CM'(q)$ for *C-CS* and $\mathscr{T}'(q^T)$ for *transformation* provenance rewrites.

Example 5.3. *As an example of the PI-CS SQL rewrite of an ASPJ query block consider the following query:*

```
SELECT sum(a) AS suma
FROM R JOIN S ON (R.b = S.c)
GROUP BY R.b
```

The SQL rewrite for this query generates a new top level query block that joins the original query with the rewritten input of the aggregation on the group-by attributes. The group-by attribute R.b has to be added to the SELECT clause of the original query to be able to perform the join. The rewritten SQL query is presented below:

```
SELECT suma, prov_R_a, prov_R_b, prov_S_c
FROM
            (SELECT sum(a) AS suma, R.b
            FROM R JOIN S ON (R.b = S.c)
            GROUP BY R.b) AS orig
      LEFT JOIN
            (SELECT R.b, R.a AS prov_R_a, R.b AS prov_R_b, S.c AS prov_S_c
            FROM R JOIN S ON (R.b = S.c)) AS sub
      ON (orig.b = sub.b)
```

5.2.5 Rewrite of SET Query Blocks

A *SET* query block *QB* contains a tree of set operations and is translated into an algebraic expression q that contains a sequence of set operators that are associated with brackets to preserve the tree structure. The rewrite rules for set operators are quite diverse. In contrast to the rewrites of *SPJ* and *ASPJ* query blocks it is not possible to present a single rewritten form q^+ for an arbitrary *SET* query block. Therefore, we first present how such a query block can be rewritten by processing one set operation at a time. Afterwards, we present simplifications that are are applicable for *SET* blocks that have a certain structure.

5.2.5.1 PI-CS Rewrite

The algebraic version q of a query block QB can be represented as $q_1 \bullet q_2$ where $\bullet$ denotes one of the set operators of the algebra. Such an algebra expression is rewritten using rewrite rule (R6.b), (R7), or (R8.b). We do not present the translations for rules (R6.a) and (R8.a) to simplify the discussion and because the translation of these rules is similar to the other translations. q is transformed by the rewrite rules into one of the algebra expression presented below:

$$q^+ = (q_1 \cup^{S/B} q_2)^+ = ({q_1}^+ \times null(\mathscr{P}({q_2}^+))) \cup^B (\Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^+)}({q_2}^+ \times null(\mathscr{P}({q_1}^+))))$$
$$q^+ = (q_1 \cap^{S/B} q_2)^+ = \Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^+)}(q_1 \cap^S q_2 \bowtie_{\mathbf{Q_1}=X} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}({q_1}^+)}({q_1}^+) \bowtie_{\mathbf{Q_1}=Y} \Pi^B{}_{\mathbf{Q_2}\rightarrow Y,\mathscr{P}({q_2}^+)}({q_2}^+))$$
$$q^+ = (q_1 -^{S/B} q_2)^+ = \Pi^B{}_{\mathbf{Q_1},\mathscr{P}(q^+)}(\Pi^S{}_{\mathbf{Q_1}}(q_1 -^{S/B} q_2) \bowtie_{\mathbf{Q_1}=X} \Pi^B{}_{\mathbf{Q_1}\rightarrow X,\mathscr{P}({q_1}^+)}({q_1}^+) \times null(\mathscr{P}({q_2}^+)))$$

Union: If q has a union as top level set operator, then the rewritten query q^+ is translated into three query blocks. One query block for the union operation, a query block for the cross product between ${q_1}^+$ with $null(\mathscr{P}({q_2}^+))$, and one query block for the cross product between ${q_2}^+$ with $null(\mathscr{P}({q_1}^+))$. Note that instead of a cross product between a rewritten input to q and null values, a projection can be used that adds the null values to the result schema. This approach is taken in the SQL rewrite for union:

QB		QB^+

q_1
UNION [ALL]
q_2;
	$\longrightarrow$	SELECT $\mathbf{Q_1}$, $\mathscr{P}({q_1}^+)$, NULL AS $\mathscr{P}({q_2}^+)$
		FROM ${q_1}^+$
		UNION ALL
		SELECT $\mathbf{Q_2}$, NULL AS $\mathscr{P}({q_1}^+)$, $\mathscr{P}({q_2}^+)$
		FROM ${q_2}^+$;

Intersection: If the top level set operation in q is an intersection, then q^+ is translated into four query blocks: A new top query block that joins the original query with the rewritten inputs of the top level intersection, a query block containing the original query QB, and a query block for each rewritten input. The resulting SQL rewrite is as follows:

QB		QB^+

q_1
INTERSECT [ALL]
q_2;
	$\longrightarrow$	SELECT orig.∗, $\mathscr{P}(q^+)$
		FROM
		    QB' AS orig,
		    ${q_1}^+$ AS sub1,
		    ${q_2}^+$ AS sub2
		WHERE
		    orig.∗ = sub1.$\mathbf{Q_1}$ AND
		    orig.∗ = sub2.$\mathbf{Q_2}$;

Set Difference: For algebra expressions q with a top level set difference, the rewritten algebra expression q^+ is translated into SQL as three query blocks. Similar to intersection a query block is introduced that joins the original query with the rewritten left hand side input. As for union, the cross product with ε constants is modeled as a projection. In q^+ a duplicate removing projection is applied to ${q_1}^+$. In the translation an additional query block is added that uses *DISTINCT* to model the duplicate removing projection on the original query. If possible, this query block is merged into the original query QB. Below we present the SQL rewrite:

QB

```
q1
EXCEPT [ALL]
q2;
```

$\longrightarrow$

QB^+

```
SELECT orig.*, 𝒫(q1+), NULL AS 𝒫(q2+)
FROM
    (SELECT DISTINCT * FROM QB) AS orig,
    q1+ AS sub1
WHERE
    orig.* = sub1.Q1
```

If a *SET* query block *QB* contains more than one set operation, then the rewrite for one set operation presented above is applied recursively to the set operations in *QB*. The recursive application of these rewrites can result in a large number of additional query blocks. The number of query blocks can be reduced if *QB* uses only unions or only intersections. In this case the rewritten algebra statement q^+ can be transformed into the following algebra statements using algebraic equivalences:

$$\begin{aligned}
q^+ = (q_1 \cup^{S/B} \dots \cup^{S/B} q_n)^+ &= \Pi^B{}_{\mathbf{Q_1}, \mathscr{P}(q_1{}^+) \dots \varepsilon \rightarrow \mathscr{P}(q_n{}^+)}(q_1{}^+) \cup^B \dots \cup^B \Pi^B{}_{\mathbf{Q_n}, \varepsilon \rightarrow \mathscr{P}(q_1{}^+) \dots \mathscr{P}(q_n{}^+)}(q_n{}^+) \\
q^+ = (q_1 \cap^{S/B} \dots \cap^{S/B} q_n)^+ &= \Pi^B{}_{\mathbf{Q_1}, \mathscr{P}(q^+)}((q_1 \cap^S \dots \cap^S q_n) \bowtie_{\mathbf{Q_1} = X_1} \\
&\quad \Pi^B{}_{\mathbf{Q_1} \rightarrow X_1, \mathscr{P}(q_1{}^+)}(q_1{}^+) \dots \bowtie_{\mathbf{Q_1} = X_n} \Pi^B{}_{\mathbf{Q_n} \rightarrow X_n, \mathscr{P}(q_n{}^+)}(q_n{}^+))
\end{aligned}$$

This simplified versions of q^+ can be translated into SQL as a top level query block that handles the union respective join operations, a query block for the modified original set operation, and a query block for each rewritten input q_i. We only present the resulting SQL rewrite rule for union. The SQL rewrite for intersection is analog.

QB

```
q1
UNION [ALL]
...
UNION [ALL]
qn;
```

$\longrightarrow$

QB^+

```
SELECT Q1, 𝒫(q1+), ... , NULL AS 𝒫(qn+)
FROM q1+
UNION ALL
...
UNION ALL
SELECT Qn, NULL AS 𝒫(q1+), ... , 𝒫(qn+)
FROM qn+;
```

The recursive definition of the SQL rewrites for set operations implies that this optimization can also be applied to a sub-tree in a *SET* query block, if this sub-tree only contains unions or intersections.

5.2.5.2 C-CS and Transformation Provenance Rewrites

The *C-CS* and *transformation* provenance rewrite rules for set operations use the same structural rewrites as the *PI-CS* rules. The translation of these rewrites into SQL is analog to the translation for *PI-CS* presented above. Therefore, we do not present the corresponding SQL rewrites.

Example 5.4. *Consider the following query over base relations R, S, and T with schemas $\mathbf{R} = (a)$, $\mathbf{S} = (b)$, and $\mathbf{T} = (c)$ as an example of the simplified PI-CS SQL rewrite for SET query blocks that contain only union set operations.*

```
( SELECT * FROM R
UNION
SELECT * FROM S )
UNION
SELECT * FROM T )
```

Applying the SQL rewrite for union the query is rewritten into:

```
( SELECT a , a AS prov_R_a , NULL AS prov_S_b , NULL AS prov_T_c FROM R )
UNION ALL
SELECT b , NULL AS prov_R_a , b AS prov_S_b , NULL AS prov_T_c FROM S )
UNION ALL
SELECT c , NULL AS prov_R_a , NULL AS prov_S_b , c AS prov_T_c FROM T;
```

5.2.6 Rewrite of SPJ-sub Query Blocks

The algebraic representation q of an *SPJ-sub* query block *QB* is generated like for an *SPJ* query block except that each sublink in *QB* is translated into an algebraic sublink expression (see section 5.2.1 on how sublinks in join conditions are translated into algebra). The result of this translation is an algebra expression of the following format:

$$q = \Pi_A(\sigma_C(q_1 \diamond \ldots q_n))$$

q may contain sublink expressions in A and/or C. This algebra expression is rewritten using the same rewrite rules applied for *SPJ* query blocks except for the sublink expressions in q that are rewritten using one of the sublink rewrite strategies. To translate these rewrite strategies into SQL the rewrite rules have to be translated into SQL and the preconditions of the strategies have to be adapted for SQL queries.

5.2.6.1 Gen Strategy

The rewrite rules of the *Gen* strategy rewrite sublinks in $q = \sigma_C(q_1)$ or $q = \Pi^{S/B}{}_A(q_1)$ by adding additional selection predicates that contain modified versions of the sublinks in q and using a cross product between the rewritten input of q and the *CrossBase* of each sublink. The *Gen strategy* rewrite rules are quite complex. Therefore, we discuss each part of the translation in detail for rewrite rule (G1), the rewrite rule for sublinks in selection predicates. The translation of (G2) is omitted, because it is analog. The *CrossBase* of a sublink contains the cross product of all base relations accessed by the sublink query with attributes renamed into the provenance attribute names unioned with a tuple of ε constants. For a sublink query q_{sub} over base relations R_1 to R_n the *CrossBase* is expressed in relational algebra as:

$$CrossBase(q_{sub}) = \Pi^B{}_{\mathbf{R_1} \to \mathcal{N}(R_1)}(R_1 \cup^B null(\mathbf{R_1})) \times \ldots \Pi^B{}_{\mathbf{R_n} \to \mathcal{N}(R_n)}(R_n \cup^B null(\mathbf{R_n}))$$

The translation of a *CrossBase* into SQL is straightforward:

```
          (SELECT R_1 AS $\mathscr{P}(R_1{}^+)$
          FROM $R_1$
          UNION ALL
          SELECT NULL AS $\mathscr{P}(R_1{}^+)$)
     CROSS JOIN
     ...
     CROSS JOIN
          (SELECT R_n AS $\mathscr{P}(R_n{}^+)$
          FROM $R_n$
          UNION ALL
          SELECT NULL AS $\mathscr{P}(R_n{}^+)$)
```

The selection predicate $C_{sub_i}{}^+$ added for each sublink expression C_{sub_i} by the *Gen* strategy contains the original sublink expression, the rewritten sublink query $q_{sub_i}{}^+$, the predicate J_{sub_i}, and correlation expressions between the $CrossBase(q_{sub_i})$ and attributes from $\mathbf{Q_{sub_i}}{}^+$:

$$C_{sub_i}{}^+ = EXISTS\ (\sigma_{J_{sub_i} \wedge \mathscr{P}(q_{sub_i}{}^+) =_n X}(\Pi^B{}_{\mathscr{P}(q_{sub_i}{}^+) \rightarrow X}(q_{sub_i}{}^+)) \vee (\neg\ EXISTS\ (q_{sub_i}) \wedge \mathscr{P}(q_{sub_i}{}^+) \text{ is } \varepsilon)$$

The elements of $C_{sub_i}{}^+$ are translated into SQL as follows. $q_{sub_i}{}^+$ is translated into SQL using the translation for the query block types used in the original SQL sublink from which q_{sub_i} is derived from. Depending on the type of sublink J_{sub_i} contains the original sublink expression C_{sub_i} and/or an equality predicate. C_{sub_i} is translated into its SQL counterpart (The SQL sublink from which it is derived from). The same applies for the correlation expressions $\mathscr{P}(q_{sub_i}{}^+) =_n X$. Recall that an algebra expression $\sigma_C(q_1)$ is rewritten by *Gen* strategy rewrite rule (G1) into:

$$(\sigma_C(q_1))^+ = \sigma_{C \wedge C_{sub_1}{}^+ \ldots \wedge C_{sub_n}{}^+}(q_1{}^+ \times CrossBase(q_{sub_1}) \ldots \times CrossBase(q_{sub_n}))$$

The algebraic representation q of an *SPJ-sub* query block QB may contain joins and a projection in addition to the selection that contains the sublinks. Assume that C_{sub_1} to C_{sub_m} are the sublink expressions in q. Applying rewrite rules (R1), (R3), and (R5.a-R5.e) in combination with (G1), q is rewritten into:

$$q^+ = \Pi^B{}_{A,\mathscr{P}'(q^+)}(\sigma_{C \wedge C_{sub_1}{}^+ \ldots \wedge C_{sub_m}{}^+}((q_1{}^+ \diamond\ \ldots\ \diamond q_n{}^+) \times CrossBase(q_{sub_1}) \ldots \times CrossBase(q_{sub_m}))$$

The rewritten algebra expression q^+ is translated into a query block for the rewritten *SPJ* part and one query block for each *CrossBase* using the SQL versions of C_{sub_i} and the $CrossBase(q_{sub_i})$:

QB

```
SELECT A
FROM $q_1$ ... $q_n$
WHERE $C$;
```

$\longrightarrow$

QB^+

```
SELECT A, $\mathscr{P}(q^+)$
FROM
     ($q_1{}^+$ ... $q_n{}^+$),
     $CrossBase(q_{sub_1})$,
     ...
     $CrossBase(q_{sub_m})$
WHERE
     $C$ AND $C_{sub_1}$ AND ... AND $C_{sub_m}$;
```

5.2.6.2 Specialized Strategies

The translation of the specialized rewrite strategies for sublinks is to a large extended analog to the translation of the *Gen* strategy. Therefore, we do not present the translated rewrites for each strategy, but instead discuss only the differences between these translations and the translation for the *Gen* strategy.

Left Strategy The *Left* strategy has the single precondition that it is only applicable for uncorrelated sublink expressions. This precondition can be checked for an SQL query in the same manner as for an algebra expression: By searching for references to correlated attributes in the expressions used in the sublink query. The rewrite rules of the *Left* strategy rewrite an algebra expression by joining the rewritten input to the selection respective projection that contains the sublinks with the rewritten sublink algebra expressions on the J_{sub_i} conditions. This rewrite is translated into SQL by adding query blocks for the rewritten sublink queries to the *FROM* clause of the rewritten query that uses sublinks. The J_{sub_i} expressions are rewritten as presented for the *Gen* strategy.

Move Strategy Recall that the *Move* strategy is derived from the *Left* strategy through the application of algebraic equivalence rules. The difference between a query q^M produced by the *Move* strategy rewrites and a query q^L produced by the *Left* strategy rewrites is that in q^M the result of sublink expressions is reused by outsourcing them into projection expressions. The same approach is applied in the SQL translation of the *Move* rewrite rules: Sublink queries are moved into the *SELECT* clause of the original *SPJ* query block and selection predicates and projection expressions that use the result of these sublinks are moved into a new top level query block. The joins with the rewritten sublink queries is processed in the *FROM* clause of this query block.

Unn Strategy The *Unn* strategy un-nests sublink expressions by transforming them into joins with the sublink query. In the SQL translation of the *Unn* rewrite rule this is implemented by moving sublink queries from the *WHERE* clause to the *FROM* clause. The preconditions of this strategy require that none of the sublink expressions used in the algebra expression is an *ALL* sublink and that the sublinks are used in logical conjunctions. This preconditions are checked for an SQL query in the same way as for an algebra expression (in SQL logical conjunctions are denoted as *AND*).

Unn-Not Strategy The *Unn-Not* strategy transforms each sublink in an algebra expressions into two joins. One join simulates the sublink expression and the other one is used to propagate the provenance of the sublink query. In the SQL translation two duplicates of the sublink query are added to the *FROM* clause. The only major difference to the translation of the *Unn* strategy is the projection on a constant value $1 \rightarrow dummy_i$ which is translated into `1 AS dummy_i` in SQL.

JA Strategy The rewrite rules of the *JA* strategy apply de-correlation and un-nesting to transform correlated sublink expressions into joins with the rewritten sublink queries. This rewrite strategy is only applicable to sublink queries that have an aggregation as their outermost operator. To de-correlate such sublinks group-by expressions are added to the aggregation (See, e.g., [Kim82]). In the SQL translation of this strategy these expressions are added to the *GROUP BY* clause of the rewritten sublink query. The correlation expressions are transformed into join predicates in both the algebraic and the SQL version of this strategy.

Exists Strategy Like the *JA* strategy, the *Exists* strategy uses de-correlation and un-nesting techniques. This strategy is only applicable for correlated *EXISTS* sublinks. In contrast to the *JA* strategy no grouping is required to de-correlated the sublinks for this strategy. Besides that, the translation into SQL is analog to the one for the *JA* rewrite rule.

Example 5.5. *We now present example rewrites for an SQL query with sublinks using SQL rewrites of the Gen and the Unn strategy. Consider the query presented below that accesses base relations R and S with schemas* $\mathbf{R} = (a)$ *and* $\mathbf{S} = (b)$.

```
SELECT *
FROM R
WHERE EXISTS (SELECT * FROM S);
```

The SQL translation of the Gen strategy rewrites transform this query into:

```
SELECT R.a, cbase.prov_S_b, R.a AS prov_R_a
FROM
     R,
     (SELECT S.b AS prov_S_b
     FROM S
     UNION ALL
     SELECT NULL AS b) AS cbase
WHERE
     EXISTS ( SELECT * FROM S)) AND
     (
          (EXISTS (SELECT S.b, S.b AS prov_S_b
                  FROM s
                  WHERE NOT S.b IS DISTINCT FROM cbase.prov_S_b))
          OR
          (NOT EXISTS (SELECT * FROM S) AND cbase.prov_S_b IS NULL)
     );
```

The rewritten query contains two additional sublinks - one of them with correlations to the CrossBase. The application of the Unn strategy SQL rewrite to this query generates the following query:

```
SELECT
          R.a, resub.prov_S_b, R.a AS prov_R_a
FROM
          R,
          (SELECT S.b, S.b AS prov_public_S_b FROM S) AS resub;
```

It is apparent from the example that the rewritten queries produced by different rewrite strategies for the same input query vary considerably. We discuss in section 5.5.1.1 how to determine which rewrite strategy minimizes the execution time of the provenance computation.

5.2.7 Rewrite of ASPJ-sub Query Blocks

As mentioned in the discussion of the translation between algebra expressions and SQL queries, the *Perm* algebra only supports sublinks in selection predicates and projection expressions. To translate the algebraic rewrites to *ASPJ-sub* query blocks, either the algebra and rewrite rules have to be extended to support sublink expressions in aggregations or the *ASPJ-sub* query blocks have to be transformed into query blocks for which equivalent expressions in the *Perm* algebra exist. We choose to use the second approach, because it requires no further extensions of the already quite complex algebra and rewrite rules. Even more, only selection, projection, and join sublink rewrites have to be realized which simplifies the implementation of sublink rewrites to a great extend.

Sublinks in aggregation can either be used in a group-by expression **(1)**, in the input of an aggregation function **(2)**, in an expression that contains an aggregation function **(3)**, or in the *HAVING* clause **(4)**. For example, the query presented below contains all types of aggregation sublinks.

```
SELECT sum(a1) IN (SELECT * FROM S) AS suma(3)
       sum(a1 * (SELECT count(*) FROM T)) AS sumb(2)
FROM R
GROUP BY a2 = ANY (SELECT * FROM U)(1)
HAVING avg(a1) = ALL (SELECT * FROM V);(4)
```

The four types of aggregation sublinks can be transformed into equivalent selection or projection sublinks. For sublinks of type **(1)** and **(2)** a new query block is added below the aggregation query block and the sublinks are moved to the new query block. To handle types **(3)** and **(4)** a new query block is added above the aggregation and the sublinks are moved to the target list or *WHERE* clause of the new query block. E.g., the query above would be transformed into the following query:

```
SELECT suma IN (SELECT * FROM S) AS suma(3),
       sumb
FROM
    (SELECT sum(a1) AS suma(3), sum(x) AS sumb(2), avg(a1) AS newagg(4)
    FROM
        (SELECT
                (a1 * (SELECT count(*) FROM T)) AS x(2),
                (a2 = ANY (SELECT * FROM U)) AS y(1)
        FROM R) AS sub
    GROUP BY y) AS agg(1)
WHERE newagg = ALL (SELECT * FROM V);(4)
```

5.2.8 Rewrite of Query Blocks with a LIMIT, ORDER BY, or DISTINCT ON Clause

The *LIMIT*, *ORDER BY*, and *DISTINCT ON* clauses of SQL do not have an algebraic counterpart. *LIMIT* and *DISTINCT ON* are not modeled in the *Perm* algebra, because their result is non-deterministic. E.g., the *LIMIT* clause returns only a specified number n of result tuples from a query, but it is not specified in the SQL standard which result tuples are left out if the query returns more than n tuples (unless a *ORDER BY* clause is used on an attribute with unique values). The *ORDER BY* clause causes a query to return its results in a specified order. This operation cannot be modeled in an algebra that is defined over bags, because there is no intrinsic order between the tuples in a bag. One approach to overcome this problem is to define an algebra that operates on ordered bags. However, the relational algebra operators (and their SQL counterpart) do not specify the order of results. Thus, the operators would be non-deterministic operators; an undesirable property. Therefore, the *Perm* algebra was deliberately defined without an ordering operator.

If the SQL rewrite rules of *Perm* are applied to a query with a *LIMIT* or *DISTINCT ON* clause this may result in the exclusion of original result tuples by the rewritten query. This is due to the fact that the provenance rewrites duplicates original result tuples to output their complete provenance. The following observation enables us to nonetheless generate meaningful provenance information for queries that use these clauses. Because of the non-deterministic nature of these clauses subsequent executions of a query that uses these clauses may return different results, but each execution generates a fixed, though unpredictable result. Since the provenance of an SQL query is generated by a single query, we can achieve consistent results for the provenance computation by preserving the original query in the provenance computation and matching its results with the provenance for all tuples that may be generated by the original query. I.e., a *LIMIT* or *DISTINCT ON* clause is rewritten by joining the original query with the rewritten query (without the *LIMIT* clause) on the original result attributes.

Example 5.6. *For example, the query presented below over base relation R ($\mathbf{R} = (a)$)*

```
SELECT * FROM R LIMIT 3;
```

is rewritten into

```
SELECT
    orig.a, prov.prov_R_a
FROM
    (SELECT * FROM R LIMIT 3) AS orig
    JOIN
    (SELECT R.a, R.a AS prov_R_a FROM R) as prov
    ON (org.a = prov.a)
```

The join between the original query and the rewritten query guarantees that the correct amount of original result tuples is produced. This additional join operation can be omitted if the *LIMIT* clause is used in a set operation or aggregation because the original query is used in the rewrite rule and, thus, the *LIMIT* clause can be applied to this query. We also do not need the join if the *LIMIT* clause is used in a query that is guaranteed to include at most one contributing tuples from each base relation for a single result tuple (e.g., an *SPJ* query without duplicate elimination).

For a query block *QB* with an *ORDER BY* clause we apply the SQL rewrite defined for the block type of *QB* and apply the unmodified *ORDER BY* clause in the top-level query block of the rewritten query. Thus, the query results will be returned in the same order as in the original query (except for the non-determinism of the order of tuples which have the same values in the order-by expressions intrinsic to the *ORDER BY* clause).

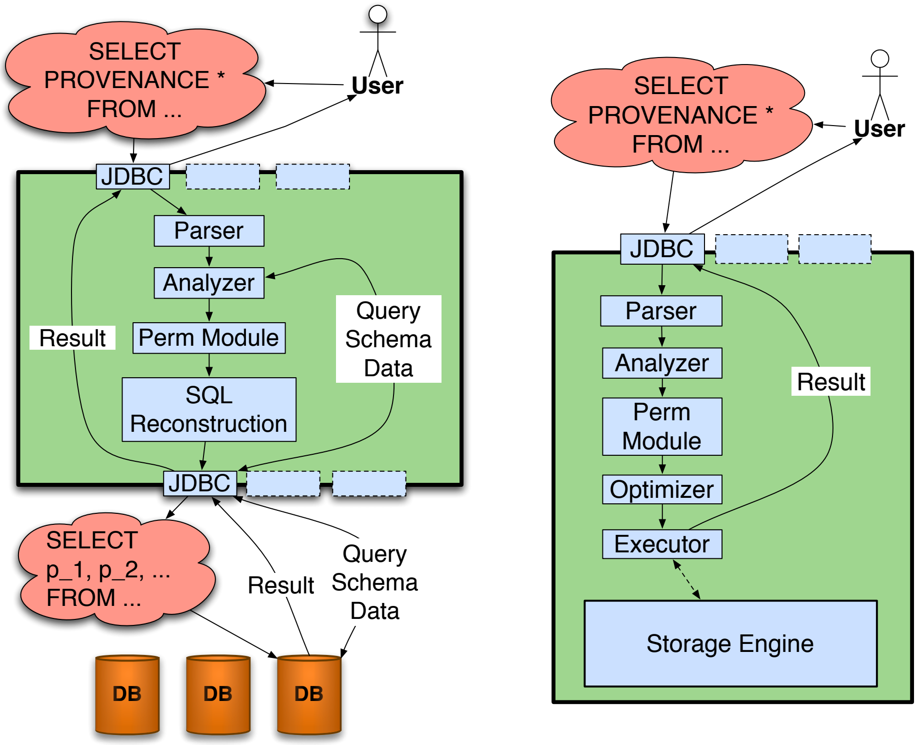


Figure 5.4: Approaches for Implementing *SQL-PLE*

5.3 Architecture

There are basically two approaches for implementing the *SQL-PLE* language extension. One approach is to implement a middle-ware solution. A user sends queries in *SQL-PLE* to this middle-ware, these queries are transformed into standard SQL by the middle-ware and send to a standard DBMS. The query results returned by the DBMS are passed on by the middle-ware to the user. The other approach is to integrate the *SQL-PLE* features into an existing DBMS.

A possible architecture of the middle-ware approach and how this approach would realize the execution of a *SQL-PLE* query is shown on the left of Figure 5.4. Like a DBMS the middle-ware has to provide one or more interfaces through which a user can connect to the system, send queries and retrieve results. For example, the middle-ware might implement the JDBC interface. To be able to transform incoming *SQL-PLE* queries into SQL and execute these queries using a DBMS, the system has to implement a full *parser* and *analyzer* for the *SQL-PLE* language. Under the term analyzer we understand the part of a query compiler that checks a query for semantic correctness. E.g., expands $*$ expressions or checks that a relation referenced by the query exists. Most of the checks applied by the analyzer need access to the database catalog of the underlying DBMS. While a valid approach would be to not do these checks at all and rely on the underlying DBMS to check the semantic correctness of a query, access to the database catalog is still necessary for rewriting a query. For instance, the rewrite rules need to be aware which attributes belong to a relation. The result of the analyzer would be an internal representation of the *SQL-PLE* query that contains all the necessary information to substitute provenance queries with their rewritten SQL counterpart. The resulting rewritten query would then be serialized into SQL text and send to the underlying DBMS. Finally, the query results returned by the DBMS are send to the user. The middle-ware approach has the advantage that it is possible to use various database systems as back-ends[2]. However, this approach also has several disadvantages. First, it is necessary to implement a full parser and analyzer. Second, incoming queries are parsed and analyzed twice; once by the middle-ware and once by the underlying DBMS. Third, the

[2] Given that queries are restricted to a subset of SQL that is understood by all these systems or that a translation between SQL dialects is implemented.

database catalog accesses applied by the analyzer (the *Query Schema Data* arrows in Figure 5.4) incur in the execution of additional SQL queries which results in additional run-time overhead.

As mentioned above an alternative approach is to extend an existing DBMS with *SQL-PLE* language constructs. To implement this approach the parser and analyzer of the underlying DBMS have to be extended to handle the *SQL-PLE* language constructs. Furthermore, the output of the analyzer has to be modified to implement the query rewrites before it is send to the optimizer of the DBMS. The optimizer, unaware that it is processing a provenance computation, creates an execution plan that in turn is executed by the DBMS and result are send back to the user. This approach has several advantages. First, less implementation effort has to be spend on the parser and analyzer component which allows to focus on the core functionality - the provenance computation. Second, the extended DBMS provides the same interfaces as the original one. This means application programs that access the database do not have to be modified, but can benefit from the *SQL-PLE* extensions. Third, it is possible to extend the system with performance optimizations that are not possible in SQL. E.g., implement specialized physical operators for computations that are common in provenance queries. The only disadvantage of this approach is that the provenance support is limited to the extended DBMS and porting it to another database system would require the reimplementation of large parts of the system.

Note that in comparison with middle-ware solutions that use a non-relational data model like, e.g., *DBNotes*, both approaches for *Perm* presented here have the advantage that they use the full potential of the query optimizer of the DBMS, because of the complete integration of provenance computation in SQL. Furthermore, approaches with a non-relational data model incur in additional overhead for transforming the relational data returned by the DBMS into this data model.

Besides the choice of architecture, we had to decide whether to use the *lazy* or *eager* approach for provenance computation. Recall that the *lazy* approach instantiates provenance information only if it is requested. The *eager* approach generates provenance information for every query send to the system and stores it for later use. We choose to use the *lazy* approach, because it has several advantages:

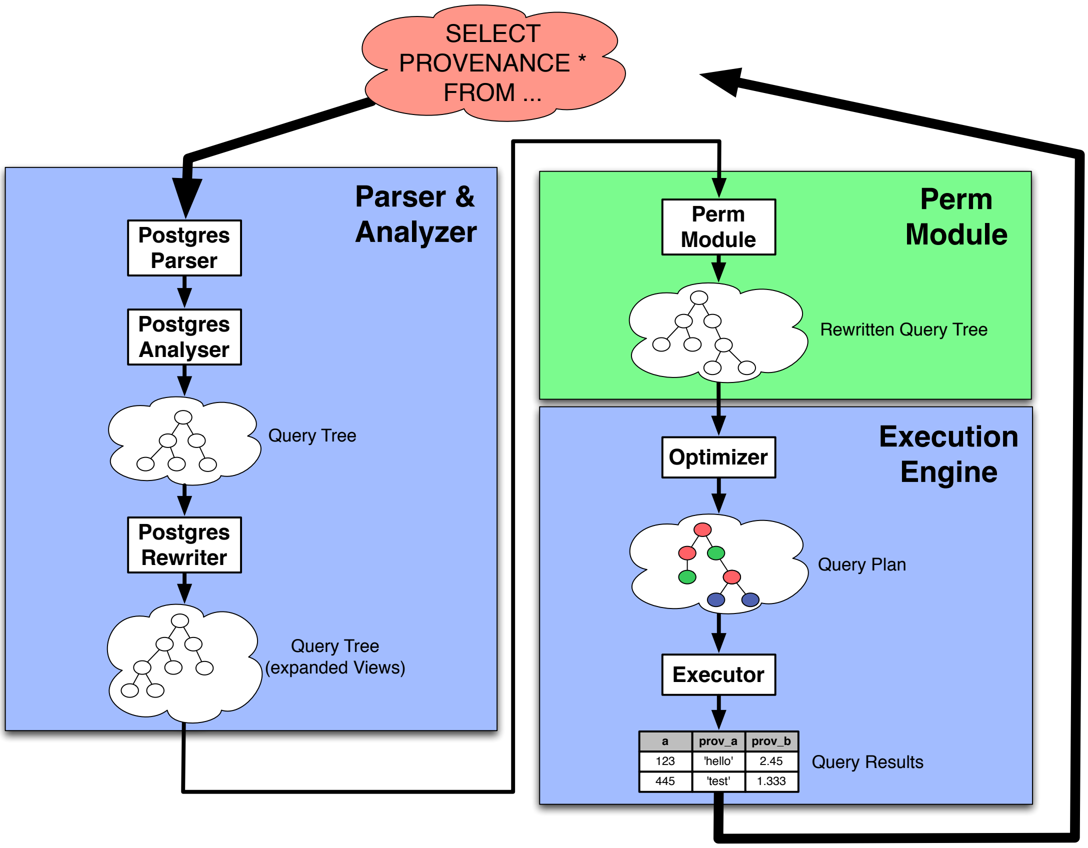
- **No Storage Overhead**: The provenance of a query can be several magnitudes larger than its result. Therefore, it is not feasible to store provenance information for all queries.
- **No Run-time Overhead for Normal Operations**: In most applications provenance information is only needed for a subset of the queries send to the systems. If the *lazy* approach is applied normal operations do not have to suffer from the overhead introduced by provenance computation.
- **Optimization of Queries**: If queries are asked over the result of an provenance computation, the *PostgreSQL* optimizer may be able to push selection predicates into the provenance query. This kind of optimization can increase the performance of such queries tremendously.

If provenance should be preserved for later use or performance benefits are expected by storing provenance instead of computing it on the fly, this is still possible using the *lazy* approach. The user can store provenance information permanently using, e.g., the *INSERT INTO* construct of *SQL*.

5.3.1 The Perm Approach

We choose to take the second approach and implement *Perm* as an extension of *PostgreSQL*, because in our opinion the advantages of this approach outweigh the advantages of the middle-ware approach. *PostgreSQL* was chosen because of the availability of well-documented source code and its extensive support of the SQL standard. The architecture of *Perm* is shown in Figure 5.5. The only major modification to *PostgreSQL* was adding the *Perm module* between the *Rewriter* and the *Optimizer*.

If a user sends a query to *Perm*, the query is first checked for syntactical correctness by the *Parser* and afterwards checked for semantic correctness by the *Analyser*. The output of the *Parser* module is a tree structure (*query tree*) that contains a node for each block of the parsed query. The *Analyser* traverses the query tree and possibly modifies the tree structure. If the query tree contains no semantic or syntactical errors it is passed to the *Postgres Rewriter*. This module is responsible for view expansion (replacing references to a view with the view definition). The output of the *Postgres Rewriter* is passed to the *Perm* module that rewrites any query block that uses *SQL-PLE* extensions using the SQL rewrite rules presented



a	prov_a	prov_b
123	'hello'	2.45
445	'test'	1.333

Figure 5.5: Perm Architecture

in 5.2. The modified query tree is then passed to the unmodified *PostgreSQL Optimizer* that creates a query plan that in turn is passed to the *Executor*. The query plan is executed by the *Executor* and results are send back to the user.

From a users point of view *Perm* appears to be a normal *PostgreSQL* server. Application programs connect to a *Perm* server through the same *APIs* as used for *PostgreSQL*. *SQL-PLE*, the *SQL* dialect of *Perm*, is a strict superset of the SQL dialect of *PostgreSQL*. Thus, existing *PostgreSQL* applications can be run without modifications on *Perm*, but may take advantage of the provenance language extensions implemented by this system.

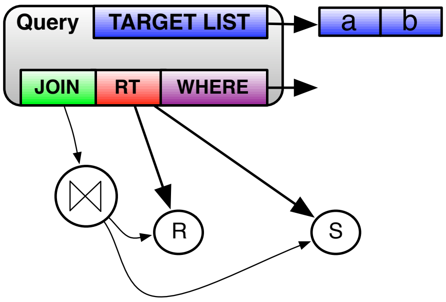


Figure 5.6: Example PQTM Representation of a Query

5.4 Modification of PostgreSQL Data Structures and Modules

In this section we present the realization of the *Perm module* that implements the *SQL-PLE* query rewrites in *Perm*. As mentioned in section 5.3 *Perm* uses the unmodified *Optimizer* module, *Executor* module, and *Storage Engine* of *PostgreSQL*. The *Parser* and *Analyzer* modules had to be adapted to cope with our the provenance language extension. In the following we first introduce the internal representation of an SQL query used in *PostgreSQL*. Afterwards, we discuss the modification of the *Parser* and *Analyzer* modules. Finally, in section 5.5 we present the implementation of the *Perm* module, the core of the *Perm* system, that implements the provenance computation and other query rewrites.

5.4.1 Postgres Query Tree Model

PostgreSQL internally represents an analyzed SQL query (the output of the *Analyzer* module) in a tree model. We refer to this model as *Postgres Query Tree Model* or short *PQTM*. Such a tree contains a *query node* for each query block of a query. *PostgreSQL* is implemented in C. Hence, the nodes used in the *PQTM* model are C-*structs* (see [KR88]). The projection expressions from the *SELECT* clause of a query block are stored in the *target list* field of a query node (The *target list* is a single-linked list). The *range table* contains the *FROM* clause items of a query stored as *RangeTableEntry* nodes. Joins between range table items are stored in the *join tree*. The join-tree contains a node for each join operation (this node also stores the join condition) of the query block and references to the range table items. For example, a join between two base relations would be represented as a top level join node with two children that are references to the range table entries of the two base relations. *WHERE* clause and *HAVING* clause predicates are stored as expression trees in the *qual* and *having* fields of a query node. The *group-by* list contains a list of grouping expressions with references to the target list. Several node types are defined for the expressions trees used in the target list, having, group by, qual fields and join conditions. For instance, nodes that represent boolean operators, function calls, or constants. Set operations are represented as a tree of set operators and range table references. If a query node represents a set operation, only the target list, range table, and set operation parts of this query node are used. In the following we use a graphical representation for query nodes that abstracts some detail where appropriate.

Example 5.7. *Figure 5.6 present the graphical notation for the example query presented below.*

```
SELECT *
FROM R JOIN S ON (a = b);
```

5.4.1.1 Perm Extensions of PQTM

Only minor modifications to *PQTM* were necessary to model the *SQL-PLE* language constructs. Before discussing these extensions we present some details about the implementation of *PQTM* in *PostgreSQL*. *PostgreSQL* uses pseudo-inheritance to handle node types which can have child nodes of different types. In *PQTM* each node type is implemented as a *C* `struct`. All node types are inherited from a base type

`Node` that contains only one field: an integer that stores a numerical identifier of the node's type. The C language does not support inheritance. Therefore, "all node types are inherited from `Node`" actually means that all other node type structs also have a node identifier as their first field[3]. Therefore, a pointer to a specific node type can be cast into a generic `Node*` without losing the ability to access its node type, and, thus, cast it into its concrete type to access its data. In a *PQTM* node type `struct`, fields that store pointers to child nodes are usually of type `Node*` enabling a node to have children with arbitrary node types. *PostgreSQL* implements node creation, deep copy of nodes, serialization and de-serialization, and deep equality comparison for all node types through functions that take as inputs `Node` pointers. These generic functions internally call specific functions defined for each node type. The pseudo-inheritance of node types and generic functions for standard operations on nodes simplify the extension of *PQTM* with new node types. To add a new node type, a new node struct is defined and the specific functions for, e.g., copying a struct of this type, are implemented. To extend an existing node type, new fields are added to the `struct` for this node type and the functions that implement the standard operations for this node type have to be adapted to the changes. In most cases extending a node type does require to alter code that uses this node type, because almost all accesses to nodes of this type that require information about the structure of the node type struct are handled by calling one of the generic functions.

We extended the *PQTM* node types to store markers for provenance computation. A new field of type *ProvInfo* is added to the *Query* node type that indicates if one of the *SQL-PLE* keywords that trigger provenance computation is used in a query block (E.g., the *TRANSSQL* keyword). The new query node field is ignored by all modules except for the *Perm* module. Besides being used to indicate which (if any) provenance rewrites should be applied to a query node, the *ProvInfo* node is also used to store auxiliary information that is produced and consumed by the *Perm* module rewrites. For instance, the implementation of the *C-CS* rewrites stores information about the copy map of a query block in the *ProvInfo* node. Some of the *SQL-PLE* language extensions alter the rewrite of *FROM* clause items (E.g., the keyword *BASERELATION*). To be able to represent this constructs, the *RangeTableEntry* node type is extended with a field that stores the necessary information for the *Perm* module to be able to handle these constructs.

5.4.2 Modifications of the Parser and Analyzer Modules

For the use in *Perm*, the parser of *PostgreSQL* had to be modified to recognize the *SQL-PLE* language extensions and output their *PQTM* representation. For instance, if the parser encounters the *PROVENANCE* keyword in a select clause, the query node for the current query block is marked for provenance rewrite using the *ProvInfo* field of the query node. If a provenance query is used as a sub-query and provenance attributes are used in selection or projection expressions, then the *Analyser* will throw an error, because it is not aware of the existence of provenance attributes. To circumvent this problem we modified the *Analyser* to make it aware of provenance attributes. I.e., if the *Analyzer* analyzes a query block it first checks if this block is marked for provenance rewrite. For query blocks that are marked for provenance rewrite the *Analyzer* extends the target list (*SELECT*) with the provenance attributes for this query block. This is feasible, because the provenance attributes of a query can be derived from its structure without access to any run-time information. The provenance attributes of a query depend only on the schemas of the base relations accessed by the query and the order of these accesses.

5.4.3 Extension of Standard SQL Commands

The implementation of some of the standard SQL commands of *PostgreSQL* have been extended to achieve a full integration of the *SQL-PLE* language extensions.

CREATE VIEW To enable a user to store the result of a provenance computation as a view the *CREATE VIEW* command needs to be expanded to deal with queries that are marked for provenance rewrite. Before the view description is stored in the database catalog tables it is passed to the *Perm* module to rewrite

[3] C compilers tend use a different memory layout for structs than the order of fields in the definition of the struct to achieve a better alignment for the struct. While this is desirable behaviour in general it has to be de-activated for compiling PostgreSQL, because it is only possible to cast node type pointers to Node* if these structs have the same layout in memory.

provenance queries. This means the standard SQL query produced by the *Perm* rewrites is actually stored in the view.

INSERT, UPDATE, and DELETE The three data manipulation statements of SQL allow SQL queries to be nested inside the manipulation statement.

Example 5.8. *For instance, a SELECT statement can be used instead of a VALUES clause in an INSERT statement:*

```
INSERT INTO table (SELECT * FROM R);
```

To enable the use of *SQL-PLE* constructs in these statements, their implementation had to be adapted to call the *Perm* module to rewrite the language extensions.

EXPLAIN The implementation of the *EXPLAIN* command of *PostgreSQL* has been extended to handle the *SQL-PLE* extensions of this command. The original implementation of *EXPLAIN* uses the *Parser*, *Analyzer* and *Optimizer* modules to generate a query plan for the query it is applied to, serializes this plan into text, and returns it as a query result. To rewrite provenance computations, a call to the *Perm* module is added to the implementation of *EXPLAIN*. Recall that *SQL-PLE* defines two extensions of the *EXPLAIN* command. The first extension (*EXPLAIN SQLTEXT*) returns the SQL text of the query passed to the command. This construct can be used to investigate the rewritten form of a provenance computation. The serialization of a query tree into SQL text utilizes build-in *PostgreSQL* functionality to perform the serialization (This functionality is used by *PostgreSQL* to present view definitions to a user). The second extension (*EXPLAIN GRAPH*) returns an algebra tree for the input query expressed in the dot language, a graph description language (see [GN00]). The translation into dot is performed by a traversal of the query tree. The structure of each query node is analyzed and an algebra tree fragment is generated for this node.

```
Input: QueryNode $q$
Output: modified QueryNode $q$

traverseQueryTree (QueryNode $q$) {
  if (IsMarkedForPIRewrite($q$))
    $q \leftarrow$ rewritePIQueryNode($q$);
  if (IsMarkedForCCSRewrite($q$))
    $q \leftarrow$ rewriteCCSQueryNode ($q$);
  if (IsMarkedForTransRewrite($q$))
    $q \leftarrow$ rewriteTransQueryNode($q$);
  else
    foreach $q'$ in $q$.rangetable
      if (IsA($q'$, QueryNode)
        $q' \leftarrow$ traverseQueryTree($q'$);
  return $q$;
}
```

Figure 5.7: traverseQueryTree Procedure

5.5 The Perm Module

The *Perm* module is the core of the *Perm* system. It operates on the extended *PQTM* representation of a query produced by the *Analyzer* module. The *SQL-PLE* language extensions are implemented by transforming any *SQL-PLE* extensions found in an input query into the standard *PQTM* representation understood by the *PostgreSQL* optimizer. Particularly, provenance computations are rewritten using the SQL version of the *Perm* rewrites.

The *PostgreSQL* analyzer was modified to pass a pointer to the root query node of the input query to the *Perm* module instead of directly handing it over to the *Optimizer* module. The *Perm* module rewrites this query tree and returns the root query node of the rewritten query tree to the *Analyzer* which in turn passes it on to the *Optimizer* module.

The pseudo code for the main procedure (*traverseQueryTree*) of the *Perm* module is shown in Figure 5.7. *traverseQueryTree* recursively traverses the input query tree one query node at a time and for each query node q checks if it is marked for provenance rewrite (or uses other *SQL-PLE* language extensions). If a query block uses one of the extensions, the appropriate rewrites are applied to transform the query block and replace it with the rewritten version. The *isMarkedForXRewrite* procedure checks if the *ProvInfo* field of the query node indicates that it is a provenance query. If this is the case the appropriate rewrite procedure is called. E.g., *rewritePIQueryNode* to rewrite a query node according to *PI-CS*. These rewrite procedures are discussed in the subsequent sections. The rewritten query tree is then returned to the *Analyzer*.

5.5.1 PI-CS Rewrite Algorithm

The procedure *rewritePIQueryNode* (Figure 5.8) of the *Perm* module implements the *PI-CS* SQL rewrites developed in section 5.2. Because of the recursive nature of the SQL rewrites, the obvious approach to implement them is a procedure that traverses the query tree and recursively rewrites each query node. Procedure *rewritePIQueryNode* computes the rewritten query q^+ top down by first rewriting the input query node q, leaving the sub-queries in its range table untouched. In a second step, all direct child query nodes are rewritten by recursive calls to *rewritePIQueryNode*. Afterwards, the list of provenance attributes ($\mathscr{P}$) of q is computed using the provenance attribute lists of its child nodes. Provenance attribute lists are stored on a global stack data structure (*pStack*). Before returning from *rewritePIQueryNode*, q^+'s $\mathscr{P}$-list is pushed on the stack. The first step distinguishes between the query block types presented in section 5.2 using the procedure *identifyBlockType*. Depending on the result of *identifyBlockType* the procedure that rewrites the identified block type is called. These procedures return a modified query node and push the list of provenance attributes of this query node on *pStack*. Afterwards, *rewritePIQueryNode* creates $\mathscr{P}$ for the current query node and returns q^+ the modified version of q. Note that we do not compute the rewrite

```
**Input**: QueryNode $q$
**Output**: QueryNode $q^+$
**Variables**: pStack ← $\emptyset$

**rewritePIQueryNode** (Querynode $q$) {
  $\mathscr{P} \leftarrow \emptyset$  // Provenance attrs of q
  switch identifyBlockType($q$)
  case NONDET
    $q^+$ ←rewriteBlockNONDET($q$)
    $\mathscr{P}$ ← pop(pStack)
  case BASE
    push(pStack, $\mathscr{N}(\mathbf{Q})$);
    return $q$
  case EXTERNAL
    push(pStack, external_attrs)
    return $q$
  case SPJ
    $q^+$ ←rewriteBlockSPJ($q$)
    $\mathscr{P}$ ← pop(pStack)
  case ASPJ
    $q^+$ ←rewriteBlockASPJ($q$)
    $\mathscr{P}$ ← pop(pStack)
  case SET
    $q^+$ ←rewriteBlockSET($q$)
    $\mathscr{P}$ ← pop(pStack)
  case SPJ−sub
    $q'$ = rewriteSublinks ($q$)
    $\mathscr{P}$ ← pop(pStack)
    $q^+$ = rewriteBlockSPJ($q'$)
    $\mathscr{P}$ ← pop(pStack) ▶ $\mathscr{P}$
  case ASPJ−sub
    $q$ = transformAggSublinks($q$)
    $q^+$ = rewritePIQueryNode($q$)
  push(pStack, $\mathscr{P}$)
  return $q^+$
}
```

Figure 5.8: PI-CS Rewrite Algorithm

bottom-up instead of top-down, because we would have to keep references to unmodified copies of sub-trees of the original query in memory. For example, for an aggregation over a projection, the bottom-up computation would rewrite the projection first. To rewrite the aggregation the original projection is needed. This means we have to keep a copy of the original projection. For complex query trees access to certain sub-trees of the original query would be cumbersome. Hence, we compute provenance top-down to avoid this additional complexity.

identifyBlockType accesses fields of the *Query* node to determine whether a query node represents a *SET* query block, an *ASPJ* query block, or contains sublinks. If the set operation field of query node q is not used then q is not a *SET* query block. The boolean field *hasAggs* is *true* if the *SELECT* clause of the query block contains aggregation functions. Together with inspection of the *groupClause* field that contains the *GROUP BY* clause expressions, this field is used to distinguish between *ASPJ* and *SPJ* query blocks. If sublinks are used in the query block is determined by examining the boolean *hasSublinks* field. To distinguish between *ASPJ-sub* and *SPJ-sub* query blocks, all expressions (projection expressions, selection and join predicates, group-by expressions, ...) of the query node have to be searched to find all sublinks used in that query node. This is necessary, because a query node may contain no aggregation functions, but contain a sublink in one of the group-by expressions, and, thus, be an *ASPJ-sub* query block.

identifyBlockType introduces three new block types: *BASE*, *EXTERNAL*, and *NONDET*. *BASE* is returned if the query node is a base relation access or the *SQL-PLE* keyword *BASERELATION* is used. *EXTERNAL* is used if the *FROM* clause item the query node represents uses the *PROVENANCE(attr_list)* construct. In this case the user provided list of provenance attributes (*attr_list*) is pushed on *pStack*. A query block is *NONDET* if it uses clauses that cause the query result to be non-deterministic or to be output in a certain order (*LIMIT*, *DISTINCT ON*, and *ORDER BY* clause).

We now discuss the procedures that implement the SQL rewrites of the individual query block types. We have already given an outline of the process of rewriting a query while presenting the SQL rewrites, but here we present a more precise and algorithmic representation that is closer to the actual implementation of these rewrites in the *Perm* system.

```
rewriteBlockSPJ (Querynode $q$) {
   $\mathscr{P} \leftarrow \emptyset$
   foreach $q'$ in $q$.rangetable
      $q'$ ←rewritePIQueryNode ($q'$) // rewrite sub-query
      $\mathscr{P} \leftarrow \mathscr{P}$ ▶ pop(pStack) // add sub-query provenance attribtutes to $\mathscr{P}$
   addAttrToTargetList($q$,$\mathscr{P}$)
   push(pStack, $\mathscr{P}$)
   return $q$
}
```

Figure 5.9: SPJ Block Rewrite Procedure

SPJ Rewrite Procedure The SQL rewrite for *SPJ* query blocks transforms a query block *QB* into a single query block QB^+. The procedure *rewriteBlockSQJ* (Figure 5.9) that implements this rewrite calls procedure *rewritePIQueryNode* for every entry in the rangetable (the *FROM* clause) of the input query node q. The provenance attributes of each rewritten rangetable entry are concatenated to build $\mathscr{P}(q^+)$. Afterwards, $\mathscr{P}(q^+)$ is pushed on *pStack* and appended to the target list (the *SELECT* clause) of q using procedure *addAttrToTargetList*.

```
rewriteBlockASPJ (Querynode $q$) {
   $\mathscr{P} \leftarrow \emptyset$
   top ← createQueryNode()
   sub ← copy($q$)
   sub ← removeAgg(sub) // remove aggregation from sub
   sub ← rewritePIQueryNode (sub)
   top.rangetable ← $q$ ▶sub
   createAggJoin(top,$q$,sub)
   $\mathscr{P}$ ← pop(pStack)
   addAttrToTargetList(top, $\mathscr{P}$)
   push(pStack, $\mathscr{P}$)
   return top
}
```

Figure 5.10: ASPJ Block Rewrite Procedure

ASPJ Rewrite Procedure Recall that an *ASPJ* query block is rewritten into a new top query block that implements the join between the original aggregation and its rewritten input. Procedure *rewriteBlockASPJ* (Figure 5.10) first creates a new query node `top` using procedure *createQueryNode*. Afterwards, a copy `sub` of the input query node q is created, the aggregation functions and *GROUP BY* and *HAVING* clauses are removed from `sub`, and `sub` is rewritten using *rewritePIQueryNode*. Then the rewritten query node `sub` and the original query node are added to the rangetable of `top`. Procedure *createAggJoin* is called to generate the join between q and `sub` on the group-by expressions. Finally, the provenance attributes of `sub` are added to the target list of top and pushed on *pStack*.

```
rewriteBlockSET (Querynode $q$) {
  if (isAllUnion($q$))
    $q^+$ ←rewriteAllUnion($q$)
    return $q^+$
  if (isAllIntersection($q$))
    $q^+$ ←rewriteAllIntersection($q$)
    return $q^+$
  $q^+$ ←rewriteSetOperator($q$, getSetTree($q$))
  return $q^+$
}
```

Figure 5.11: SET Block Rewrite Procedure

SET Rewrite Procedure The SQL rewrite of a *SET* query blocks depends on the set operations applied by this block. Therefore, *rewriteBlockSET* (Figure 5.11), the procedure that rewrites such blocks, first applies some checks to determine which rewrite should be applied. If the input query node q uses solely union or intersection, procedure *rewriteAllUnion* respective *rewriteAllIntersection* are used to rewrite q. Otherwise, procedure *rewriteSetOperator* is applied. This procedure takes two parameters - a query node and a tree of set operations. In *rewriteBlockSET*, the complete set operation tree of q is extracted using *getSetTree*.

Recall that in the general case a tree of set operations is rewritten by processing one set operation at a time starting at the root of the tree. Procedure *rewriteSetOperator* (Figure 5.12) retrieves the root set operation (*root*) from its set operation tree parameter *set*. Query nodes (q_1 and q_2) are extracted for the left and right sub-tree under *root* (*root.left* and *root.right*). *createQueryForSetOps* extracts a query node for a sub-tree of the set operation tree of q by creating a new query node, adding the sub-tree as the set operation tree for this query node, and adding all range table entries from q to the range table of the new query node that are referenced by the sub-tree. For instance, assume q is a query node representing the *SET* query block presented below.

```
(SELECT * FROM R INTERSECT SELECT * FROM S) UNION SELECT * FROM T)
```

In this case the q_1 and q_2 query nodes created by *createQueryForSetOps* would represent the following SQL queries:

```
q1 = (SELECT * FROM R INTERSECT SELECT * FROM S)
q2 = SELECT * FROM T
```

As apparent in the example, q_1 and q_2 may or may not contain set operations. To omit additional checks, these query nodes are rewritten using calls to *rewritePIQueryNode*. Following the generation of q_1 and q_2 a new query node *top* is created. Each set operator requires a different rewrite. Therefore, *identifySetType* is used to distinguish between the operators. If the *root* is a union, the *SELECT* clauses of ${q_1}^+$ and ${q_2}^+$ are extended with constant *null* values to make them union compatible (*adaptProvAttrs*). Both ${q_1}^+$ and ${q_2}^+$ are added to the rangetable of *top*. Afterwards the set operations for *top* is set to the bag union of ${q_1}^+$ with ${q_2}^+$ and $\mathscr{P}$ is generated by concatenating the provenance attribute lists of these queries. For intersections the range table of *top* includes the original query q and the rewritten children of the set operation (${q_1}^+$, and ${q_2}^+$) which are joined with q (*createSetJoin*). Similar to the algorithm for union the provenance attribute list is build as the concatenation of the provenance attribute lists of ${q_1}^+$ and ${q_2}^+$. Set differences are rewritten by adding q and ${q_1}^+$ to the range table of *top* and joining this two range table entries. $\mathscr{P}$ is generated by retrieving $\mathscr{P}$ of ${q_1}^+$ from *pStack* and appending constant *null* values for the provenance attributes of ${q_2}^+$. Finally, for all three set operation types, the provenance attribute list $\mathscr{P}$ is appended to the target list of *top* and pushed on *pStack*.

```
**rewriteSetOperator** (Querynode $q$, SetTree set) {
  root ← getRootNode(set)
  $q_1$ ←createQueryForSetOps(root.left)
  $q_2$ ←createQueryForSetOps(root.right)
  ${q_1}^+$ ←rewritePIQueryNode($q_1$)
  ${q_2}^+$ ←rewritePIQueryNode($q_2$)
  top ← createQueryNode()
  switch identifySetType($root$)
  case union
    adaptProvAttrs(${q_1}^+$)
    adaptProvAttrs(${q_2}^+$)
    top.rangetable ← $q_1^+ \blacktriangleright q_2^+$
    top.setoperations = ${q_1}^+ \cup^B {q_2}^+$
    $\mathscr{P}$ ← pop(pStack)
    $\mathscr{P}$ ← pop(pStack) $\blacktriangleright$ $\mathscr{P}$
  case intersection
    top.rangetable ← $q \blacktriangleright q_1^+ \blacktriangleright q_2^+$
    createSetJoin(top,${q_1}^+$)
    createSetJoin(top,${q_2}^+$)
    $\mathscr{P}$ ← pop(pStack)
    $\mathscr{P}$ ← pop(pStack) $\blacktriangleright$ $\mathscr{P}$
  case set difference
    top.rangetable ← $q \blacktriangleright q_1^+$
    createSetJoin(top,$q_1$)
    $\mathscr{P}$ ← pop(pStack) $\blacktriangleright$ $null(\mathscr{P}({q_2}^+))$
  addAttrsToTargetList(top,$\mathscr{P}$)
  push(pStack,$\mathscr{P}$)
  return top
}
```

Figure 5.12: Single Set Operation Rewrite Procedure

A *SET* query block q that uses solely union is rewritten by procedure *rewriteAllUnion* (Figure 5.13) extending the algorithm for single unions. All range table entries of q are rewritten with *rewritePIQueryNode* and *null* constants are added to their *SELECT* clauses to make them union compatible. For each range table entry q' *null* constants are added for the provenance attributes of all other range table entries in q. The provenance attribute list for q is build incrementally. After each range table entry rewrite the resulting provenance attribute list is appended to $\mathscr{P}$.

```
**rewriteAllUnion** (Querynode $q$) {
  $\mathscr{P}$ ← $\emptyset$
  foreach $q'$ in $q$.rangetable
    $q'$ ←rewritePIQueryNode($q'$)
    adaptProvAttrs($q'$)
    $\mathscr{P}$ ← $\mathscr{P}$ $\blacktriangleright$ pop(pStack)
  push(Pstack,$\mathscr{P}$)
  return $q$
}
```

Figure 5.13: Rewrite Procedure for Set Operations Containing only Union

Figure 5.14 presents *rewriteAllIntersection*, the rewrite procedure for a *SET* query block that contains only intersection. Similar to the rewrite of a single intersection, a new query node *top* is created that joins the original query q with all rewritten range table entries of q. The range table, provenance attribute list, and join operations are generated by iteratively extending these constructs with the information from the rewritten range table entries of q.

```
rewriteAllIntersection (Querynode $q$) {
  $\mathscr{P} \leftarrow \emptyset$
  top $\leftarrow$ createQueryNode()
  top.rangetable = copy($q$)
  foreach $q'$ in $q$.rangetable
    $q'$ $\leftarrow$rewritePIQueryNode($q'$)
    top.rangetable $\leftarrow$ top.rangetable $\blacktriangleright$ $q'$
    $\mathscr{P} \leftarrow \mathscr{P}$ $\blacktriangleright$ pop(pStack)
    createSetJoin(top,$q'$)
  addAttrsToTargetList(top,$\mathscr{P}$)
  push(pStack,$\mathscr{P}$)
  return top
}
```

Figure 5.14: Rewrite Procedure for Set Operations Containing only Intersection

5.5.1.1 SPJ-sub Rewrite Procedure

Sublinks can be rewritten using one of the rewrite strategies developed for these constructs (Though not all strategies may be applicable to a certain sublink). Thus, the *Perm* module has to (1) determine which rewrite strategies are applicable for a sublink and (2) choose which one should be used to rewrite the sublink. To determine which rewrite strategies can be applied for the sublinks of an *SPJ-sub* query node q, the preconditions of each rewrite strategy are checked for each sublink in q. To be able to evaluate these checks the following tasks have to be processed:

1. Find all sublinks in q by traversing the expressions of q (e.g., the selection predicate).

2. Build a data structure (called *SublinkInfo*) for each sublink to store information that is needed to determine the pre-conditions of the rewrite rules and is used later on to simplify the rewrite process.

    - Find all correlation expressions used in the sublink query and analyze the context they are used in.
    - Search for sublinks nested inside the sublink.
    - Analyze the context of the sublink. For instance, are all parent nodes in the expression tree, the sublink is used in, logical conjunctions.

Based on the *SublinkInfo* for each sublink in q, the preconditions of the rewrite strategies are evaluated. Afterwards, the *Perm* module has to decide which of the applicable rewrite strategies is used to rewrite each sublink. The general goal is to choose the rewrite strategy that results in the most efficient query. I.e., the query for which the optimizer will generate the plan that has the lowest execution time. But how can we determine which rewrite strategy will generate the query that results in the most efficient execution plan? One approach for choosing the rewrite strategy is to use a heuristic that based on, e.g, the structure of a query. Another approach is to generate a rewritten version of q for each combination of applicable rewrite strategies, use the optimizer to generate a query plan for each version, and execute the query plan with the lowest cost estimate. Though the second approach seems to be more feasible, we nonetheless use the first approach for the following reasons. First, in the second approach the number of queries for which we have to produce a query plan can grow exponentially in the number of sublinks in q. Second, it is evident that some rewrite strategies are superior to others. E.g., queries produced *Gen* strategies have a higher degree of complexity than the queries produced by the other rewrite strategies. However, we also implemented the second strategy to be able to experimentally verify this claim (see section 6.4). We use a simple preference order O between strategies as the heuristic for the first approach. I.e., if both strategies X

and *Y* are applicable, we use the strategy that is preferred according to *O*. The heuristic is to always prefer strategies that use un-nesting and de-correlation techniques to other types of rewrites.

The rationale behind this decision is that the rewritten sublink queries have to be joined with the rewritten remainder of the original query to compute the provenance. If sublinks are transformed into joins by un-nesting, these joins can be used to propagate the provenance of the sublink. Therefore, un-nesting strategies often avoid additional join operations if applied to provenance queries.

```
rewriteSublink (Querynode q) {
sublinkList ← findSublinks(q)
infoList ← ∅
foreach s in sublinkList
  infoList ← infoList ▶ createInfo(s)
foreach i in sublinkList
  appStrategies ← getApplicableStrategies(i)
  bestStrategy ← applyHeuristic(appStrategies)
  applyRewriteStrategy(q,i,bestStrategy)
return q
}
```

Figure 5.15: SPJ-sub Block Rewrite Procedure

Figure 5.15 presents an abstract overview of the algorithm used to rewrite sublinks. Note that we have left out some implementation details in the discussion of the rewrite strategies. Several problems had to be overcome in the implementation of the sublink rewrites that do not show in the algebraic version of the rewrite strategy. For example, in SQL there are severe restrictions on which outer attributes can be used in correlation expressions inside a sublink. This restrictions prohibit the application of the *Unn* strategy to sublinks that contain nested sublinks that have been rewritten with the *Gen* strategy. While it is beyond the scope of this thesis to discuss every problem faced in implementing *Perm*, we nonetheless included this example to demonstrate that it is necessary to fully implement a concept to be able to understand all its implications.

5.5.1.2 NONDET Rewrite Algorithm

We have discussed in section 5.2.8 how to rewrite queries that use the *LIMIT*, *ORDER BY*, or *DISTINCT ON* clause. For query nodes with a *LIMIT* clause the original query node *q* has to be joined with the rewritten query node (with striped of *LIMIT* clause) on the original result attributes. This join can be omitted if either (1) we can determine that the rewritten query will not duplicate result tuples (e.g., *q* is an *SPJ* query block without *DISTINCT*) or (2) the original query is preserved unmodified in the rewrite (e.g., if *q* is a *ASPJ* query block). A *DISTINCT ON* clause is handled in the same way. An *ORDER BY* clause is removed from *q* prior to the application of the rewrites and then added to the top query node of the rewritten query. We do not present the procedure *rewriteBlockNONDET* as pseudo-code, because its representation is obvious.

5.5.2 Example

We now present an example application of the *PI-CS* rewrite algorithm implemented by *rewritePIQueryNode*. Reconsider the example database from Figure 5.1 (section 5.1, a database with shops, items, and sales). The *SQL-PLE* statement presented below computes the provenance of a query *q* that computes the total revenue for each shop in the database.

```
SELECT PROVENANCE name, sum(price)
   FROM shops, sales, items
   WHERE name=sName AND itemId=id
   GROUP BY name;
```

In *PQTM* this query is represented as a single query node *q* that contains the aggregation functions in its *target list*, the group by on *name*, the selection predicate, and the base relation accesses in its *range table*. A simplified representation of *q* is depicted in Figure 5.16.(1). The *traverseQueryTree* procedure (Figure 5.7) recognizes that its input, the top query node *q*, is marked for provenance rewrite and calls *rewritePIQueryNode* to rewrite *q*. *rewritePIQueryNode* (Figure 5.8) uses procedure *identifyQueryBlock* to determine the type of query block it is processing. *q* is an ASPJ query node, so *rewritePIQueryNode* calls *rewriteBlockASPJ* (Figure 5.10) to rewrite *q*. The *rewriteBlockASPJ* procedure creates a new query node *top* (line 3), creates a copy *sub* of *q* and removes the aggregation function calls, *GROUP BY*, and *HAVING* clauses from *sub* (lines 4 and 5). The result of this operations in presented in Figure 5.16.(2). Afterwards procedure *rewritePIQueryNode* is called to rewrite the *sub* query node. *sub* is an *SPJ* query block. Therefore, procedure *rewriteBlockSPJ* (Figure 5.9) is used to rewrite it. This procedure rewrites each range table entry of *sub* by another call to *rewritePIQueryNode*. Since the range table entries of *sub* are the base relation accesses to *shops*, *items*, and *sales*, *rewritePIQueryNode* rewrites these query nodes by pushing their provenance attribute lists on *pStack*. The state after each of these rewrites is presented in Figure 5.16.(3) to 5.16.(5). For instance, after rewrite of *shops* (Figure 5.16.(3)) the provenance attribute list for *shops* is on the top of *pStack*. After the rewrite of all base relations accesses there are three attribute lists on *pStack* (see Figure 5.16.(5)). In the last step of *rewriteBlockSPJ* the provenance attribute lists of the range table entries of *sub* are popped from *pStack* and are combined to form the provenance attribute list for *sub*, which is pushed on *pStack* before *rewriteBlockSPJ* returns (5.16.(6)). Afterwards, procedure *rewriteBlockASPJ* adds the original query node *q* and the rewritten query node *sub* to the range table of the new query node *top* and adds the join between *q* and *sub* on the group-by expressions (Figure 5.16.(7)). Finally, *rewriteBlockASPJ* pops the provenance attribute list of *sub* from *pStack*, adds it to the target list of *top*, and pushes it on *pStack* again. The query node *top* returned by *rewriteBlockASPJ* is then return to *traverseQueryTree* via *rewritePIQueryNode*.

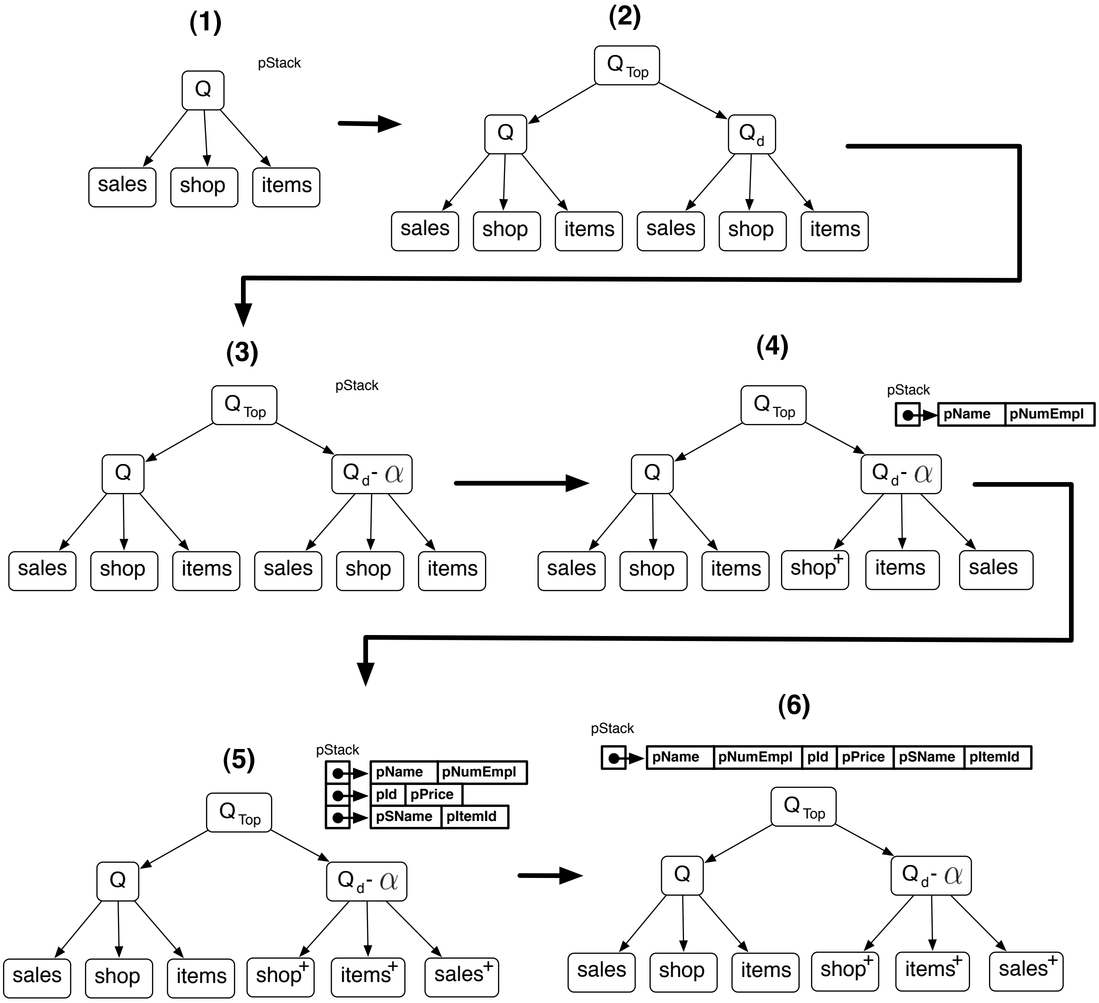


Figure 5.16: Example Application of the *PI-CS* Rewrite Algorithm

```
generateCMExpr (Querynode $q$) {
  subCM ← ∅
  foreach $q'$ in $q$.rangetable
    subCM ← subCM ▶ generateCMExpr($q'$)
  opTree = getOperators($q$)
  $CM$ ←generateCMForOps($q$,subCM, opTree)
  $CM$ ←simplifyCMExpr ($CM$)
  return $CM$
}

simplifyCMExpr (CopyInfo ci) {
  foreach $if\ (C)\ then\ (e_1)\ else\ (e_2)$ in ci
    if (isConstantTrue($C$))
      substitute(ci, $if\ (C)\ then\ (e_1)\ else\ (e_2)$, $e_1$)
    if (isConstantFalse($C$))
      substitute(ci, $if\ (C)\ then\ (e_1)\ else\ (e_2)$, $e_2$)
  foreach $e_1 \cup e_2$ in ci
    if (isConstant($e_1$) ∧ isConstant($e_2$))
      substitute(ci, $e_1 \cup e_2$, $[[e_1 \cup e_2]]$)
  foreach $e_1 \cap e_2$ in ci
    if (isConstant($e_1$) ∧ isConstant($e_2$))
      subsitute(ci, $e_1 \cap e_2$, $[[e_1 \cap e_2]]$)
  return ci
}
```

Figure 5.17: CM Expression Generation Procedure

5.5.3 Extension of the Rewrite Algorithm for C-CS

The algebraic and SQL rewrites that implement provenance computation for the *C-CS* types are derived from the ones for *PI-CS*, because these *CS* types are extensions of *PI-CS*. In *Perm* we currently implement only *CDC-CS* as a proof of concept on how to integrate these *CS* types in a *provenance management system*. In this section we describe how this has been achieved by extending the *PI-CS* rewrite algorithm and point out how the approach could be modified to implement the other *C-CS* types. The only difference between *C-CS* rewrites and the *PI-CS* rewrite are the $\mathscr{C}$ copy attributes that are propagated in addition to the provenance attributes of a query and the inclusion expressions that filter out parts of the provenance based on the content of these attributes. A straightforward approach to implement the *C-CS* rewrites is to add the generation of the *CM* expressions to the rewrite rules. However, we have demonstrated in section 4.4.3 that several simplifications are applicable for *C-CS* rewrites that do not apply to *PI-CS* rewrites. For instance, the rewrite of a part of a query can be completely omitted, if the provenance of this part is guaranteed to be filtered out by the inclusion conditions applied in the outermost projection of a *C-CS* rewrite. To be able to apply these simplifications, we use the following approach. In a first step the *CM* expressions of the query tree to be rewritten are generated bottom-up and analyzed for simplification potential. The following simplification opportunities are investigated starting at the *CM* expressions for base relations (which are constant):

- Derive constant parts of *CM* expressions.
  - A conditional $if\ (C)\ then\ (e_1)\ else\ (e_2)$ is replace with e_1 if C evaluates to $true$ for all inputs and replaced with e_2 if C evaluates to $false$ on all inputs.
  - A union or intersection is precomputed if both of its inputs are constant.
- If all *CM* expressions for a sub-query q are constant, we do not need to rewrite the sub-query at all.

Figure 5.17 presents the algorithm for the *CM* generation. The procedure *generateCMExpr* produces the *CM* expressions (stored in a so-called *CopyInfo* data structure) for a query node q. First, the *CM* expressions of all the range table entries of q are generated by recursive calls to *generateCMExpr*. Afterwards, the current query node is analyzed regarding the algebra tree it represents. This step is needed to determine how the *CM* expressions for q should be build. Afterwards, the *CM* for q are generated by recursively generating the *CM* expression for each operator in q using the precomputed *CM* expressions for the range table entries of q (procedure *generateCMForOps*). Finally, the *CM* expressions are simplified using procedure *simplifyCMExpr*. *simplifyCMExpr* searches for expressions in the input *CopyInfo* that are constant and, hence, can be replaced with their evaluation result (Recall that $[[e]]$ denotes the result of evaluating e). *substitute* is used to replace all occurrences of the second parameter in *ci* with the third parameter of this procedure.

The result of *CM* expression generation and simplification is used in the adaptation of the *PI-CS* rewrite algorithm to *C-CS*. Figure 5.18 presents *rewriteCSQuery*, the entry point for *C-CS* rewrites. This procedure calls *generateCMExpr* to generate the *CopyInfo* for its input query. Afterwards, *rewriteCSQueryNode* is called to recursively rewrite each query node in q. In a last step the inclusion conditions $\mathscr{P}^*$ are added to the *SELECT* clause of the outermost query node in q^+.

For an input query node q, procedure *rewriteCSQueryNode* determines if it is necessary to rewrite this query node by inspecting the *CopyInfo* data structure. For query nodes that do not need to be rewritten, constant *CM* expressions are added to the *SELECT* clause of q before returning. Otherwise, a modified version of *rewritePIQueryNode* is called to determine the query node type of q and rewrite it accordingly. *rewriteModPIQueryNode* basically works like *rewritePIQueryNode* with the sole exception that it also adds *CM* expressions to rewritten query nodes.

```
Input: QueryNode $q$
Output: QueryNode $q^+$

rewriteCSQuery (Querynode $q$) {
  ci ← generateCMExpr (Querynode $q$)
  $q^C$ ←rewriteCSQueryNode ($q$, ci)
  generateInclusionExpr ($q^C$, ci)
  return $q^C$
}

rewriteCSQueryNode (Querynode $q$, CopyInfo ci) {
  if (IsFixed($q$,ci)
    $q$ ←addConstantCMExpr ($q$, ci)
    return $q$
  $q^C$ ←rewriteModPIQueryNode ($q$, ci)
  return $q^C$
}
```

Figure 5.18: CDC-CS Rewrite Algorithm

5.5.4 Transformation Provenance Rewrite Algorithm

SQL-PLE extends SQL with three keywords for computing *transformation* provenance: *TRANSPROV*, *TRANSSQL*, and *TRANSXML*. All these keywords trigger the same provenance computation, but apply a different result representation.

5.5.4.1 Result Representation

Recall that in section 4.5, we modeled the *transformation* provenance of a result tuple t as a set of node identifiers Θ_w for each annotated algebra tree in the *transformation* provenance of t (one tree per witness list w in the *PI-CS* provenance of t). This representation is valid, because all annotated algebra trees represent

R

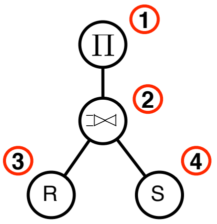
a	b
1	2
1	3
2	3
2	5

S

c
2
3

Q$_a$

a
1
2



q_a = SELECT a FROM R LEFT JOIN S ON (b = c); $q_a = \Pi^S{}_a(R ⟕_{b=c} S)$

TRANSPROV

a	transprov
1	1111
1	1111
2	1111
2	1110

TRANSSQL

a	transprov
1	SELECT a FROM R LEFT JOIN S ON (b = c);
1	SELECT a FROM R LEFT JOIN S ON (b = c);
2	SELECT a FROM R LEFT JOIN S ON (b = c);
2	SELECT a FROM R LEFT JOIN <NOT>S</NOT> ON (b = c);

TRANSXML

a	transprov
1	<Query><Select><Attr><Var>R.a</Var></Attr></Select><From><LeftJoin><Relation>R ...
1	<Query><Select><Attr><Var>R.a</Var></Attr></Select><From><LeftJoin><Relation>R ...
2	<Query><Select><Attr><Var>R.a</Var></Attr></Select><From><LeftJoin><Relation>R ...
2	<Query><Select>... <NOT><Relation>S</Relation></NOT><On><Equal><Var>... </Query>

Figure 5.19: Example Transformation Provenance Representations

the same algebra tree with different annotation functions θ_w. Therefore, we factored out the static part (that is the tree) and used the provenance attribute $\mathscr{T}$ to store only the annotation functions. Each value of $\mathscr{T}$ stores Θ_w for one witness-list w of t (represented as the set of operators that have a 1-annotation).

A representation of the annotation sets that enables an efficient computation of the rewrite rules is a *bit-vector* (compare section 5.2.4.2), because its space requirements are low, and the union operation used frequently in the rewrite rules is efficient (bit-wise disjunction). Therefore, we use this representation in the computation of *transformation* provenance.

The *TRANSPROV* keyword represents *transformation* provenance as the raw bit-vectors generated by the rewrite. To provide a useful *transformation* provenance representation to the user and enable queries over the structure of annotated query trees, the bit-vector representation is transformed into either SQL text with markup or XML (which can be queried using the build-in *X-Path* support of *PostgreSQL*) before it is returned to the user. Which representation is chosen is specified by the user by issuing the keyword *TRANSSQL* or *TRANSXML* to trigger provenance computation. The translation from the bit-vector to the external representation is implemented as UDFs f_{SQL} and f_{XML} that are applied in the outermost *SELECT* clause of the rewritten query. The SQL representation is the original query except for parts that do not belong to the *transformation* provenance, which are enclosed by `<NOT>` and `</NOT>`.

The XML representation is a hierarchical representation of an SQL statement where each clause is modeled as an XML element. The XML representation generated by *TRANSXML* is closely related to the SQL syntax of a query, but explicitly highlights its underlying structure. For example, the <Query> element contains an element for each clause of the query and the element for the *SELECT* clause contains an <Attr> element for each attribute this clause projects on. The main use case of the XML representation is to enable a user to ask meaningful queries over the *transformation* provenance. For instance, the build-in XSLT functionality of *PostgreSQL* can be used to answer queries like: To which result tuples did the join in the query contribute too. In the SQL text representation such structural queries are cumbersome or not possible at all, because the query support of *PostgreSQL* for text is much less powerful than its query support for XML data.

Example 5.9. *Figure 5.19 shows an example for the three types of transformation provenance representation. The query q_a depicted in this Figure is a left join between two base relations R and S projected on attribute a from relation R. On the top right of the Figure the algebra tree with the generated node identifiers is shown (Recall that the node identifiers are generated by a pre-order traversal of the algebra tree. See section 4.5). On the bottom of Figure 5.19 the result representations generated by the three transformation provenance keywords is presented. Keyword TRANSPROV represents the provenance as the bit-vectors used in the provenance computation. The transformation provenance of q_a contains annotated algebra trees with 1-annotations for all witness lists except $w =< (2,5), \perp) >$ for which S carries a 0-annotations. The result tuple for w is the last tuple in the presented relation with the bit-vector* 1110. *The corresponding representation for keyword TRANSSQL represents the annotations as mark-up on the SQL text of q_a. In the last tuple of this relation (the one corresponding to w) the relation S is enclosed in <Not> and </Not> to indicate the 0-annotation of this part of the query.*

```
Input: QueryNode $q$
Output: QueryNode $q^T$

rewriteTransQuery (Querynode $q$) {
  transInfo ← generateTransInfo ($q$)
  transInfo ← deduceStaticTransParts ($q$,transInfo)
  $q^T$ ←rewriteTransQueryNode ($q$,transInfo)
  addReprFuncInvocation ($q^T$ ,transInfo)
  return $q^T$
}

rewriteTransQueryNode (Querynode $q$, TransInfo tInfo) {
  if (IsFixed($q$,tInfo)
    $q$ ←addConstantIdentSet ($q$,tInfo)
    return $q$
  $q^T$ ←rewriteModTransQueryNode ($q$,tInfo)
  return $q^T$
}
```

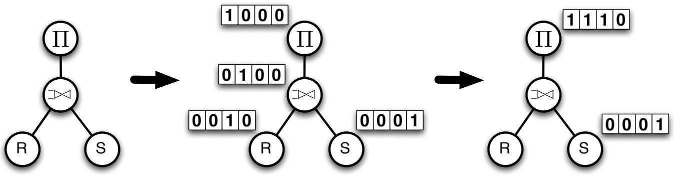
Figure 5.20: Transformation Rewrite Algorithm

5.5.4.2 Rewrite Algorithm

Like for *C-CS* the computation for *transformation* provenance is build upon the *PI-CS* rewrite algorithm. We perform the computation in three steps. In the first step the query tree is analyzed to identify the algebra tree it represents and the node identifiers for each operator in the algebra tree are generated. During this traversal of the query tree auxiliary data structures are build that store the association between the algebra tree nodes and parts of the query tree and information about which parts of the query tree are guaranteed to have a constant *transformation* provenance. Recall that we discussed in section 4.6.1 how to simplify the computation of *transformation* provenance based on the structure of an algebra expression. In the second step a modified version of the *PI-CS* rewrite algorithm is applied to rewrite the query and propagate the bit-vectors that store the *transformation* provenance. If the *TRANSSQL* or *TRANSXML* keyword is used then the bit-vectors that represent the provenance of the query are passed to the user defined functions f_{SQL} or f_{XML} to transform them into the SQL respective XML representations.

Figure 5.20 presents procedure *rewritTransQuery* that implements this algorithm. In this procedure *generateTransInfo* is used to generate the auxiliary data structures (called *TransInfo*). The *TransInfo* is analyzed with *deducedStaticTransParts* to identify parts of the query that have constant *transformation* provenance. Afterwards, q is rewritten using procedure *rewriteTransQueryNode*. This procedure applies a modified version of the *rewritePIQueryNode* procedure to rewrite a query node according to its type or

q_a = SELECT TRANSSQL R.a
FROM R LEFT JOIN S ON (R.b = S.c);



${q_a}^T$ = SELECT
R.a,
f_{SQL}(1110 $\vee^B$ $\varepsilon(c, 0001)$) AS tprov
FROM R LEFT JOIN S ON (R.b = S.c);

a	transprov
1	SELECT a FROM R LEFT JOIN S ON (b = c);
1	SELECT a FROM R LEFT JOIN S ON (b = c);
2	SELECT a FROM R LEFT JOIN S ON (b = c);
2	SELECT a FROM R LEFT JOIN <NOT>S</NOT> ON (b = c);

Figure 5.21: Transformation Provenance Computation Example

adds a constant node identifier set (*addConstantIdentSet*) to the *SELECT* clause of the query node if the *transformation* provenance of the currently processed sub-tree is constant. Finally, *addReprFuncInvocation* adds an invocation of the f_{SQL} or f_{XML} functions to the transformation provenance attribute of the outermost query block if necessary.

Example 5.10. *Figure 5.21 demonstrates the use of the TRANSSQL keyword to trigger transformation provenance computation for example query q_a from Figure 5.19. Procedure generateTransInfo of the rewrite algorithm generates the algebra tree for q_a (the left tree depicted in Figure 5.21), traverses this tree to generate the node identifiers, and for each operator in the query a bit-vector representing the singleton set containing the node identifier of this operator is generated (Fig. 5.21 the middle algebra tree). deduceStaticTransParts analyzes the resulting TransInfo to identify static parts. In this case the right-hand side of the left-join is static, therefore, a fixed bit-vector for this sub-tree is pre-computed. The same applies for the combination of projection, left join and the left input of the join. Either all or none of these operators are in the transformation provenance. Therefore, they can be represented as a single bit-vector (shown as the right tree in Figure 5.21).*

Figure 5.21 also shows the query q^T after application of rewriteTransQueryNode. In this example, the basic structure of the query is preserved and the provenance computation is limited to a projection expression. Function ε in the example is used to check if attribute b is null (it returns an empty bit-vector if its first argument is null and its second argument otherwise). The final representation is generated by function f_{SQL} (addReprFuncInvocation). The result of the rewritten query is presented at the bottom of Figure 5.21. For the fourth tuple, the right input of the left outer join did not contribute anything to the result and is therefore not in the transformation provenance. In contrast, all operators contributed to the first, second, and third result tuple.

5.5.5 Extended Selection Push-down

In many cases, a user is interested in the provenance of just part of a query result. In *SQL-PLE* a part of a query's provenance is computed by using the provenance computation as a sub-query. The *WHERE* clause of the outer query can be used to restrict the result to the part the user is interested in.

Example 5.11. *For instance, assume the user is interested in the provenance of result tuples of example query q_a from Figure 5.1 with a sName attribute value of 'Merdies':*

```
SELECT *
FROM
    (SELECT PROVENANCE s.sName, i.price
    FROM sales s JOIN items i ON s.itemId = i.id) AS p
WHERE sName = 'Merdies';
```

This kind of computation can be expensive if the selection conditions are not pushed down to lower levels of the query. *PostgreSQL* does apply selection push-down during the optimization of a query, but several important push-down possibilities are not considered. To reduce the cost of partial provenance computation, we added a specialized selection push-down module that applies more aggressive push-down of selection conditions and introduces new additional filter conditions derived from selection predicates. For instance, we derive implied inequalities, which is not done by *PostgreSQL*. The rationale behind this approach is that provenance computation for most types of queries is expensive and can produce large amounts of tuples. Therefore, we should exploit every possibility to reduce the number of intermediate results.

In addition to filtering out irrelevant tuples, the selection conditions sometimes enable us to transform outer joins introduced by the rewrite rules into inner joins. This is due to the fact that most of the outer joins are only necessary to handle special cases like *null*-values. For instance, the aggregation rewrite rule uses a left outer join on the group-by expressions. If an equality selection condition on the grouping attributes is used in the outer query, tuples that fulfill the selection condition cannot have *null* values in these attributes[4] and, thus, the outer join can be transformed into an inner join.

The selection push-down module operates on the rewritten version of a query. The query tree structure is traversed to identify selection push-down opportunities. We build auxiliary data structures that store, e.g., which inequalities hold for which parts of a query. This data structures are then used to push original and deduced selection predicates down the query tree as far as possible.

[4] In SQL the result of comparing a value with the *null*-value on equality is defined to return *null*. If a *WHERE* clause condition evaluates to *null* on a tuple t, then t filtered out.

5.5.6 Complete Query Processing Example

Having presented the inner workings of the *Perm* module we now present a complete example query evaluation with *Perm*. The example illustrates the provenance computation of the following query q_{ex}:

```
SELECT PROVENANCE *
FROM v1;
```

The view v_1 used in this query is defined over the example database from Figure 5.1 and created by the following SQL statement:

```
CREATE VIEW v1 AS
   SELECT sName, sum(price) AS revenue
   FROM sales JOIN items ON (itemId = id)
   GROUP BY sName;
```

Figure 5.22 illustrates how *Perm* processes this query. The user issues query q_{ex} to compute the provenance of view v_1. The incoming SQL query is parsed by the *Parser* and transformed into a *PQTM* query tree with a single query block that is marked for provenance rewrite (represented by the red dot in the figure). At this point the view is handled like a base relation. The *Analyzer* performs the semantic analysis of the query during which the star expression is replaced with the attributes from view v_1 and the provenance attributes are added to the target list (here we use the prefix p to identify a provenance attribute to not clutter the presentation with long attribute names). Afterwards, view v_1 is expanded by the *Postgres Rewriter*. The result is a query tree with two blocks. One for the outer query and one for the view definition. The *Perm* module traverses the query tree, recognizes the provenance marker in the top query block and rewrites the marked sub-tree. The top-level query block is an *SPJ* query block. Thus, this block is rewritten by rewriting all its range table entries and adding the provenance attributes generated by this rewrites to the target list of the query block. The only range table entry of this query block is the query block that represents the view. This query block is an *ASPJ* query block. It is rewritten by creating a new top level query block that joins the original *ASPJ* query block with the rewritten input of the aggregation. In this case the input to the aggregation is a simple join between two base relations. It is rewritten by adding the provenance attributes for the base relations to the target list of this query block. The resulting query tree contains four query blocks. This query tree is then fed into the *Optimizer* which generates an execution plan and calls the *Executor* to execute the plan. Finally, the results produced the executor are send back to the user.

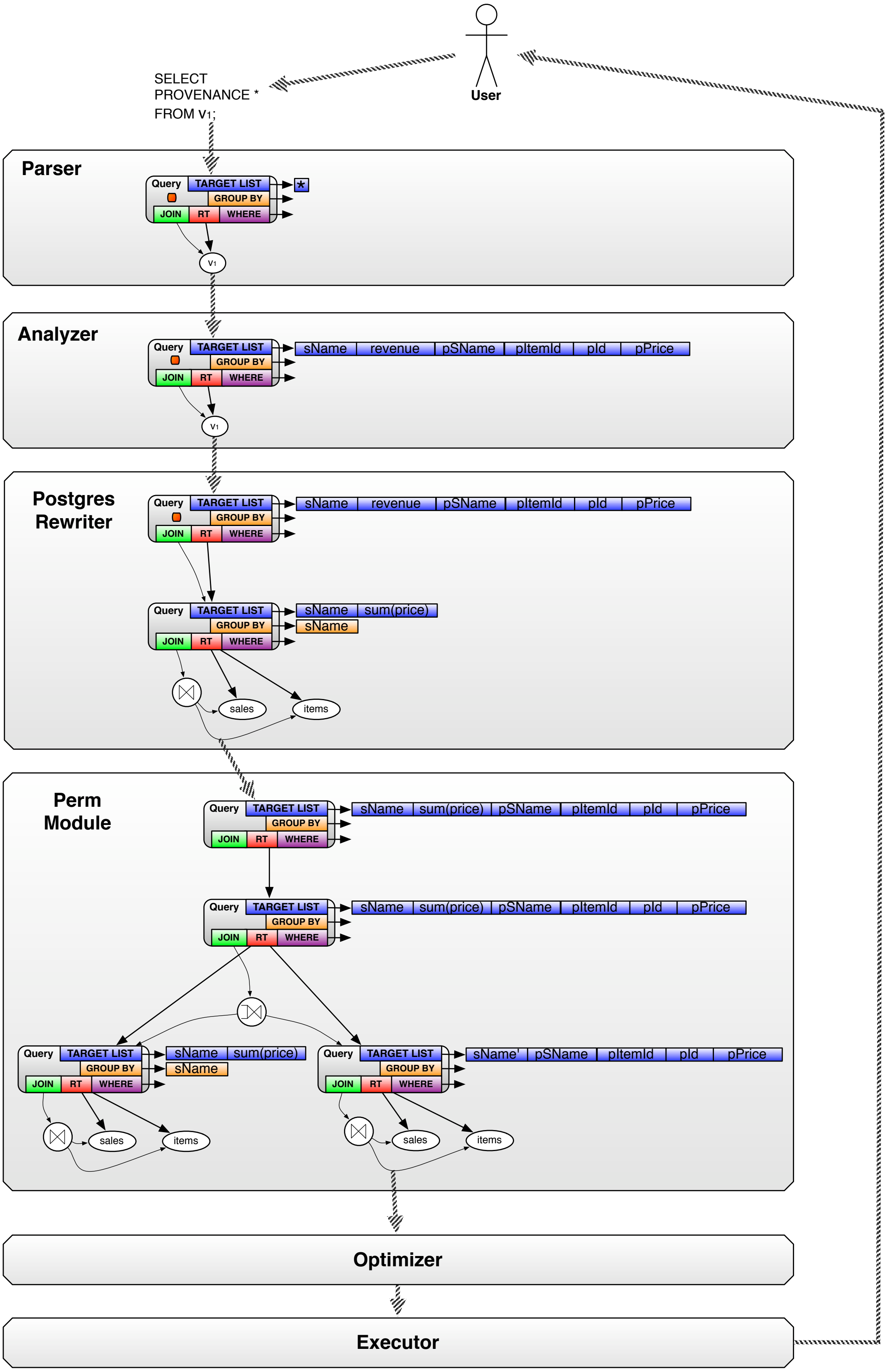


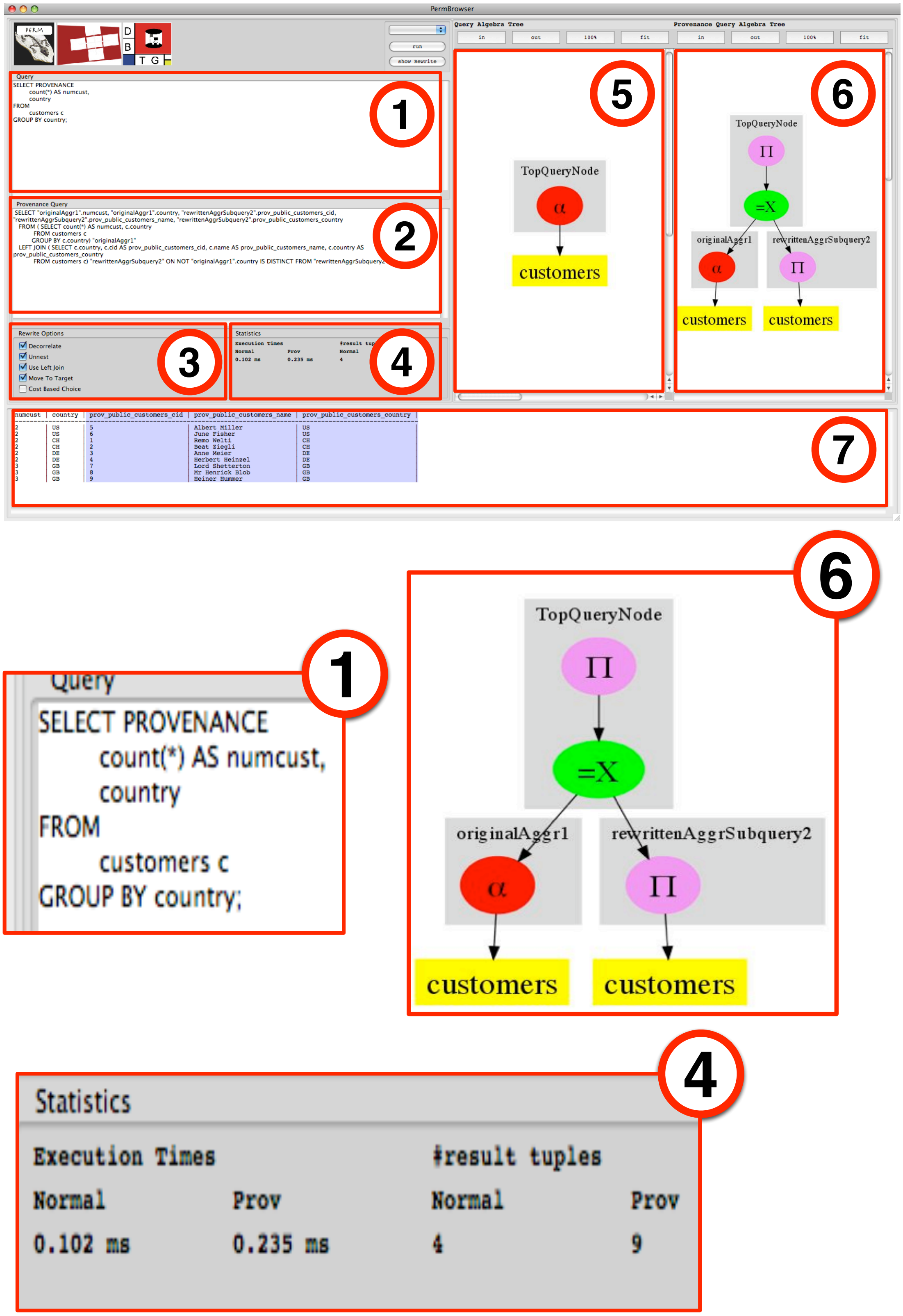
Figure 5.22: Example Processing of a *SQL-PLE* Query by *Perm*

5.6 The Perm Browser

The *Perm Browser* is a client application for *Perm* that visulizes the rewrites applied by the system. The browser enables a user to send *SQL-PLE* queries to the system, view query results, activate or deactivate rewrite strategies, and choose between different contribution semantics. In addition to the query results, the browser presents the rewritten query as an SQL statement, algebra trees for the original and rewritten query, and statistics about the run-time and number of result tuples for both the original and rewritten query.

Figure 5.6 shows the graphical user interface of the *Perm* browser (A screenshot on the top and below some enlarged parts of the screenshort). On the top left of the screen the user can enter queries into a text field (marker 1 in the Figure). If the user hits the run button, the browser application executes the query and presents the results in the text box at the bottom of the user interface (Figure 5.6 marker 7). In the example use case presented in Figure 5.6 the user has entered a *SQL-PLE* statement that uses the *PROVENANCE* keyword to compute the provenance of an aggregation over relation *customer*. In addition to the query results the browser shows the rewritten query produced by *Perm* as SQL text (see Figure 5.6 marker 2), the original query as an algebra tree (see Figure 5.6 marker 5) and an algebra tree for the rewritten query (see Figure 5.6 marker 6). In the example use case the original query is an aggregation over a base relation. Hence, the algebra tree for the original query contains two nodes; the top level aggregation (the red α) and the base relation access (the yellow box). Recall that an aggregation query block is rewritten into three query blocks (The browser places all operators of a query block into a grey box). As shown by the browser a new top level query block is introduced that joins (the green node) the original aggregation (the left child) with the rewritten input (the right child). In the example the rewritten input is a simple projection. To enable a user to understand the cost of a provenance computation the browser presents the run-time and number of result tuples for both the original and rewritten query (see Figure 5.6 marker 4). The user can activate and deactivate the sublink rewrite strategies to understand their impact on the run-time and structure of the rewritten query (see Figure 5.6 marker 3).

The *Perm Browser* is implemented as a Java application that connects to a *Perm* server via JDBC (see [Ree00]). The algebra trees are constructed by executing *EXPLAIN GRAPH* commands for a query (see section 5.1) and passing the dot script produced by this command to the dot tool [GN00] that generates a layout for the tree. The rewritten SQL text is fetched from the server by sending an *EXPLAIN SQLTEXT* command. Statistics are collected with the standard *EXPLAIN ANALYZE* command of *PostgreSQL*.



Query

```
SELECT PROVENANCE
        count(*) AS numcust,
        country
FROM
        customers c
GROUP BY country;
```

Provenance Query

```
 SELECT "originalAggr1".numcust, "originalAggr1".country, "rewrittenAggrSubquery2".prov_public_customers_cid,
"rewrittenAggrSubquery2".prov_public_customers_name, "rewrittenAggrSubquery2".prov_public_customers_country
   FROM ( SELECT count(*) AS numcust, c.country
           FROM customers c
          GROUP BY c.country) "originalAggr1"
   LEFT JOIN ( SELECT c.country, c.cid AS prov_public_customers_cid, c.name AS prov_public_customers_name, c.country AS
prov_public_customers_country
           FROM customers c) "rewrittenAggrSubquery2" ON NOT "originalAggr1".country IS DISTINCT FROM "rewrittenAggrSubquery2
```

Rewrite Options

☑ Decorrelate
☑ Unnest
☑ Use Left Join
☑ Move To Target
☐ Cost Based Choice

Statistics

```
Execution Times                 #result tuples
Normal          Prov            Normal          Prov
0.102 ms        0.235 ms        4               9
```

```
numcust  | country | prov_public_customers_cid | prov_public_customers_name | prov_public_customers_country |
---------------------------------------------------------------------------------------------------------------
2        | US      | 5                         | Albert Miller              | US                            |
2        | US      | 6                         | June Fisher                | US                            |
2        | CH      | 1                         | Remo Welti                 | CH                            |
2        | CH      | 2                         | Beat Ziegli                | CH                            |
2        | DE      | 3                         | Anne Meier                 | DE                            |
2        | DE      | 4                         | Herbert Heinzel            | DE                            |
3        | GB      | 7                         | Lord Shetterton            | GB                            |
3        | GB      | 8                         | Mr Henrick Blob            | GB                            |
3        | GB      | 9                         | Heiner Hummer              | GB                            |
```

Figure 5.23: Perm Browser User Interface

5.7 Summary

In this chapter we presented the implementation of the *Perm* system as an extension of *PostgreSQL*. In detail, the *SQL-PLE* language has been discussed that enriches SQL with constructs for provenance computation and management, we presented the translation of the algebraic rewrite rules to SQL that are used in the *SQL-PLE* implementation, and have discussed the rewrite algorithms applied by the *Perm* module.

Adding provenance support as an orthogonal language extension enables the use of SQL to query provenance information. Because of the implementation of provenance computation as the execution of rewritten queries, *Perm* benefits from the advanced query optimization techniques supplied by *PostgreSQL*. In contrast to alternative approaches like *Trio* or *DBNotes*, our system is capable of computing the provenance for more complex queries (e.g., *Perm* is the only relational provenance management system that supports sublinks), supports external provenance, and in addition provides full SQL query functionality for provenance. The implementation of provenance computation for sublinks required the development and implementation of sophisticated rewrite strategies to deal with these constructs. In addition to the powerful *PI-CS* contribution semantics, *Perm* also supports *transformation* provenance and *CDC-CS*.

Chapter 6

Experimental Evaluation

In this chapter we present an extensive performance evaluation of the *Perm* system using the TPC-H benchmark and synthetic data-sets. With this experiments we want to answer the following questions:

- What is the overhead *Perm* introduces for execution of standard SQL queries?
- What is the cost of provenance computation as implemented in *Perm*?
- Is the heuristic for choosing the rewrite strategies for sublinks feasible?
- How does *Perm* compare to the competitive *Trio* system?

In the following we first discuss the hardware configurations on which the experiments have been performed. Afterwards, the data-sets and queries used in the experiments are presented. Finally, we discuss the experiments that were run to answer the questions outlined above. The experimental results we are going to present demonstrate the feasability of our approach.

Hardware Configuration 1

Characteristic	Value
Main Memory	1GB (2 x 512MB 667MHz DDR2)
Processor	1 x Intel Core Duo Processor T2300 1.66 Ghz
Operating System	Mac OS X 10.5.5
Hard Disk	Seagate Momentus 5400.2 ST96812AS SATA (5400 rpm) - 60GB

Hardware Configuration 2

Characteristic	Value
Main Memory	8GB (4 x 2GB 667 MHz DDR2 ECC)
Processor	1 x Intel Quad-Core Xeon 5150 2.66 GHz
Operating System	Mac OS X 10.5.5
Hard Disk	5.5TB RAID-5-array: 14 x Hitachi Deskstar 7K500 (7200 rpm) - 500GB

Figure 6.1: Hardware Configurations

6.1 Experimental Setup

6.1.1 Hardware Configuration

We used two types of machines for the experiments. The first is a small desktop machine with limited resources. The other one is a server machine with an extensive amount of main memory and a RAID storage. The two systems were chosen to investigate the performance of the system on different hardware. Figure 6.1 shows the characteristics of these machines. The operating system installed on both types of machines is Mac OS X 10.5.5.

6.1.2 Test Database and Query Generation

Before presenting the experiments we discuss the generation of test data and queries. Many of the experiments were performed with TPC-H decision support benchmark. The remaining experiments use synthetic data. For each data-set and set of test queries we give a short overview of their characteristics and how they were created.

6.1.2.1 TPC-H Benchmark

The *TPC-H* [Tra09] benchmark provided by the *Transaction Processing Council* is a standard benchmark for decision support systems. The benchmark uses a fixed database schema and is supplied with a data generator that can be used to create randomized instance databases of various sizes. In *TPC-H* terminology the database size is called *scale factor*. A *scale factor* of 1.0 corresponds to roughly 1GB of data. The benchmark consists of 22 complex query templates using aggregation and sublinks. Each of these templates contains one or more substitution parameters. The TPC-H benchmark is provided with a query generator that randomly sets the parameters of a template and adapts the queries for a certain database size. We used this generator to generate a set of 1000 queries for each benchmark query template and database size. The same sets of generated were used in all experiments. Database instances were created in sizes 1MB, 10MB, 100MB, 1GB, 10GB and 100GB. We choose to conduct a huge fraction of the experiments using this benchmark for the following reasons. Firstly, TPC-H is a widely used and fairly developed benchmark for ad hoc OLAP queries, a prominent use case for provenance. Secondly, no standard benchmark for relational provenance management systems exists. Furthermore, TPC-H is a stress test of the system, because is uses operations like nested sublinks for which the provenance is hard to compute.

6.1.2.2 Additional Queries for the TPC-H Schema

We generated additional queries over the TPC-H schema tailored for investigating the performance of the provenance computation for specific types of query blocks. We generated queries with solely *SPJ*, *ASPJ*, and *SET* query blocks. These queries are described in detail in section 6.3.2 where the experiments based on these queries are presented.

6.1.2.3 Synthetic Data-sets

In addition to the TPC-H data-sets and queries we generated synthetic data-sets for the evaluation of the rewrite strategy selection heuristic for sublinks. For these experiments data-sets with a controllable distribution of values were needed to be able to directly influence the number of result and intermediate result tuples of the queries used in these experiments.

Query	Postgres Run-Time (s)	Perm Run-Time (s)	Relative Overhead (%)	Absolute Overhead (s)
1	0.090849	0.091332	0.53	0.00048299
2	0.003766	0.003816	1.32	0.00005000
3	0.003602	0.003610	0.22	0.00000799
4	0.001707	0.001755	2.81	0.00004800
5	0.008751	0.008809	0.66	0.00005800
6	0.004306	0.004409	2.39	0.00010300
7	0.007090	0.007133	0.60	0.00004299
8	0.010141	0.010166	0.24	0.00002499
9	0.022854	0.022991	0.59	0.00013700
10	0.007417	0.007523	1.42	0.00010600
11	0.002384	0.002399	0.62	0.00001500
12	0.008677	0.008690	0.14	0.00001299
13	0.004920	0.004932	0.24	0.00001199
14	0.004941	0.005078	2.77	0.00013700
15	0.009948	0.010234	2.87	0.00028600
16	0.006741	0.006768	0.40	0.00002700
17	0.001001	0.001031	2.99	0.00003000
18	0.012350	0.012550	1.61	0.00020000
19	0.007639	0.007763	1.62	0.00012400
20	0.001863	0.001914	2.73	0.00005099
21	0.002070	0.002137	3.23	0.00006700
22	0.005554	0.005795	4.33	0.00024100

Figure 6.2: Overhead of Perm for Normal Queries of the TPC-H Benchmark

6.2 Overhead for Normal Operations

To demonstrate that the performance overhead of *Perm* for queries without provenance computation in comparison with *PostgreSQL* is negligible, we compared the run-times of the TPC-H queries on *Perm* with their run-time on a normal *PostgreSQL* installation. Note that the overhead is not dependent on the database size, since a standard SQL query is executed by *Perm* using the standard *PostgreSQL Optimizer* and *Executer* modules. Therefore, we focus on the 1MB TPC-H database instance where the relative overhead is more apparent. We also ran these experiments on larger database sizes with the expected result of the same absolute overhead. One query instance of all 22 TPC-H query templates was executed 10.000 times on *Postgres* and *Perm*. The absolute and relative overhead introduced by *Perm* for normal query execution is depicted in Figure 6.2 (The median of the run-times is presented). This experiment clearly shows that the overhead of *Perm* for normal operations is negligible. The maximal absolute overhead is about 0.5 milliseconds; the maximal relative overhead is 4.33 percent. As mentioned before we omit the results for larger database sizes, because the overhead is insignificant small in comparison with the overall execution time.

6.3 Cost of Provenance Computation

In this section we investigate the cost of computing the provenance of SQL queries. For most experiments we could only compare the performance of provenance computation in *Perm* with the performance of the same queries without provenance computation, because most of the queries used in the experiments are not supported by other provenance management systems. For a comparison of *Perm* to the *Trio* we had to use very primitive test queries. Another approach could have been to only run experiments with queries that are supported by both systems. We refrained from this approach, because it would give an incomplete view on the cost of provenance computation. For instance, the provenance for queries with sublinks may lead to an enormous growth in the number of result tuples. We deliberately choose to not omit the complicated cases and also investigate the performance of language constructs with expensive provenance computation.

6.3.1 TPC-H Queries

In the first set of experiments we ran 100 instances of each TPC-H query template for database sizes ranging from 1MB to 100GB. For each database size, the whole set of queries was run with and without provenance computation. Provenance was computed using *PI-CS* because it is more complex than *CDC-CS* and is better suited for queries that use aggregation and sublinks operations. Due to the heavy use of aggregation in TPC-H, *CDC-CS* gives non-empty results only for three queries in the benchmark. Each combination of database size and query template was given a 2 day time-slot. If the execution of the 100 instance queries for this combination did not finish within this time limit we did not execute more query instances for this configuration. This experiments were run on hardware configuration 2.

The results of the experiments for database sizes 100MB and 1GB are shown in Figure 6.3. The results for plain queries are shown in Figure 6.3 on the left and for rewritten queries in Figure 6.3 on the right. The missing bar for query 18 on a 100MB database is due to the fact that the result set of this query is empty. As far as we are aware, *Perm* is the first system able to compute the provenance of all 22 queries.

The results show that *Perm* is able to deal with complex queries. However, the provenance of some queries is intrinsically very large. This can be seen in the significant differences between the time to compute queries with (up to a few hours) and without provenance (up to a few minutes). The reason for this overhead can be seen in the two lower graph of Figure 6.3. The result size for queries without provenance vary between one and up to 10.000 tuples. Queries with provenance lead to result sizes between hundred and hundreds of millions of tuples according to the semantics we have used. For instance, the average number of tuples of the provenance for query type 22 on a 1GB database instance consists of approximately 230 million tuples. Now that with *Perm* the provenance of these queries can be computed, it should be possible to study ways to reduce this complexity. These result sizes also confirm the need for different contribution semantics within *Perm*, particularly those that would result in smaller result sizes than *PI-CS*. An open problem is to find meaningful contribution semantics that reduce the result sizes but do not end up with empty results in most cases (like *CDC-CS*). The same applies to different provenance representations, which could also contribute to reduce the result size.

In terms of scalability, the problem of large result sizes comes up again. Most queries that do not contain sublinks (1,3,5,6,7,8,9,12,13,14 and 19) exhibit a run time growth that is linear in the size of the database. We can confirm this because we have additional experiments for database sizes of 1MB, 10MB, 10GB, and 100GB. For larger databases, only a few queries complete in reasonable time. Queries that contain sublinks are hard to optimize and not well-supported by *PostgreSQL*. The rewrite strategies that use un-nesting and de-correlation are what has enabled us to compute the provenance for these queries in reasonable time, but nevertheless do result in a super-linear growth in run-time. This opens up several lines for future work to make provenance computation also feasible for large databases in combination with complex queries.

To be able to better understand the overhead of provenance computation for different query types, Figure 6.4 shows the total run-times and run-time per result tuple for queries 1, 2, 21 , and 22 and for database sizes ranging from 1MB to 1GB. We selected these query types because they represent different types of query characteristics. Query 1 is an aggregation without sublinks. Query 2 is an *SPJ* query with a single correlated sublink. Query 21 and 22 both contain multiple sublinks. As apparent from the graphs, the total execution time increases rapidly with database size, but the growth is based on the increase in the

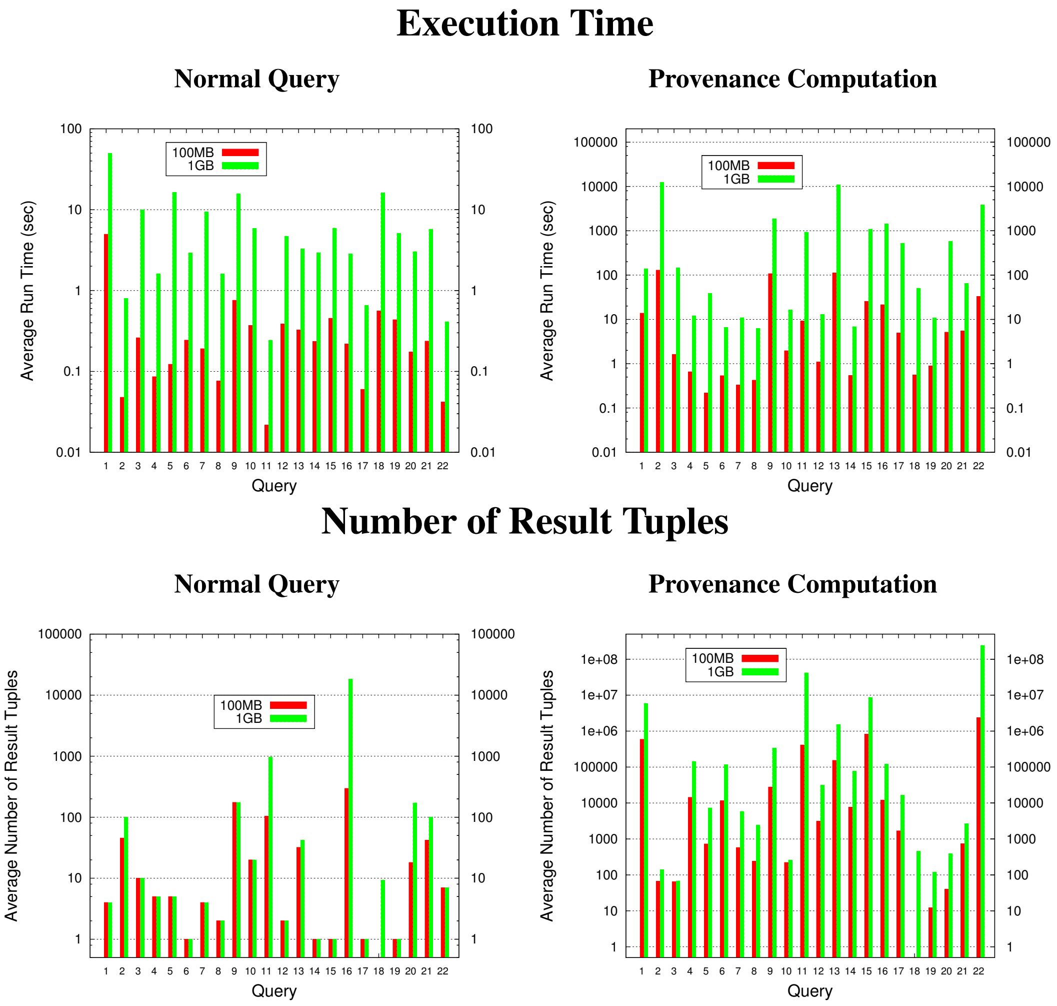


Figure 6.3: TPC-H Comparison of Run-times

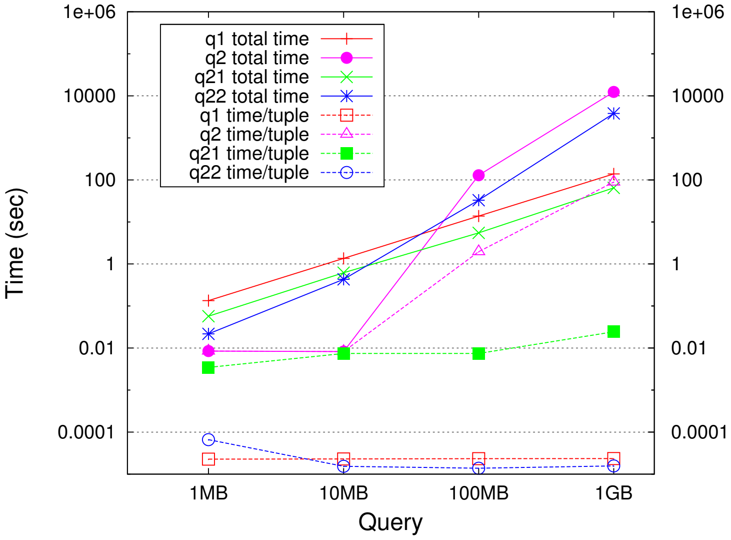


Figure 6.4: TPC-H Average Total and Per-Tuple Execution Times for queries 1, 2, 21, and 22

number of tuples that belong to the provenance of a query as demonstrated by the minimal growth of time spend to produce a single tuple of the result. For smaller database sizes, the query results contain only a few tuples. Thus, the total execution time is dominated by the time spent to analyze and optimize the query. This explains the increase in execution time per tuple of query 22 for small database sizes.

6.3.1.1 Analysis of the Correlation between Normal and Provenance Run-times

To gain a deeper understanding of the correlation of provenance query run-times and normal query run-times, we present a subset of the previous experiments on TPC-H as scatter-plots. These plots compare the provenance and normal run-times for a specific query template and database size. Each point p in these graphs represents the execution of one query with and without provenance computation. The x-axis value of p is the normal run-time of a query and the y-axis value of p is the run-time of the query with provenance computation. These plots enable us to understand how these two run-times are correlated for different types of queries. We only present some typical scatter-plots to highlight the types of correlations that occur. Figure 6.5 shows scatter-plots for query templates 6 and 14 and database size 1MB, 10MB, 100MB, and 1GB. While for small database sizes the correlation between normal and provenance query run-time is mostly covered by noise, for large database sizes the two measures are highly correlated. As expected, queries instances with a higher normal run-time also have a higher run-time for provenance computation. Query 6 is a good example for how different values of the parameters of a query template generate clusters of queries with similar run-time behaviour.

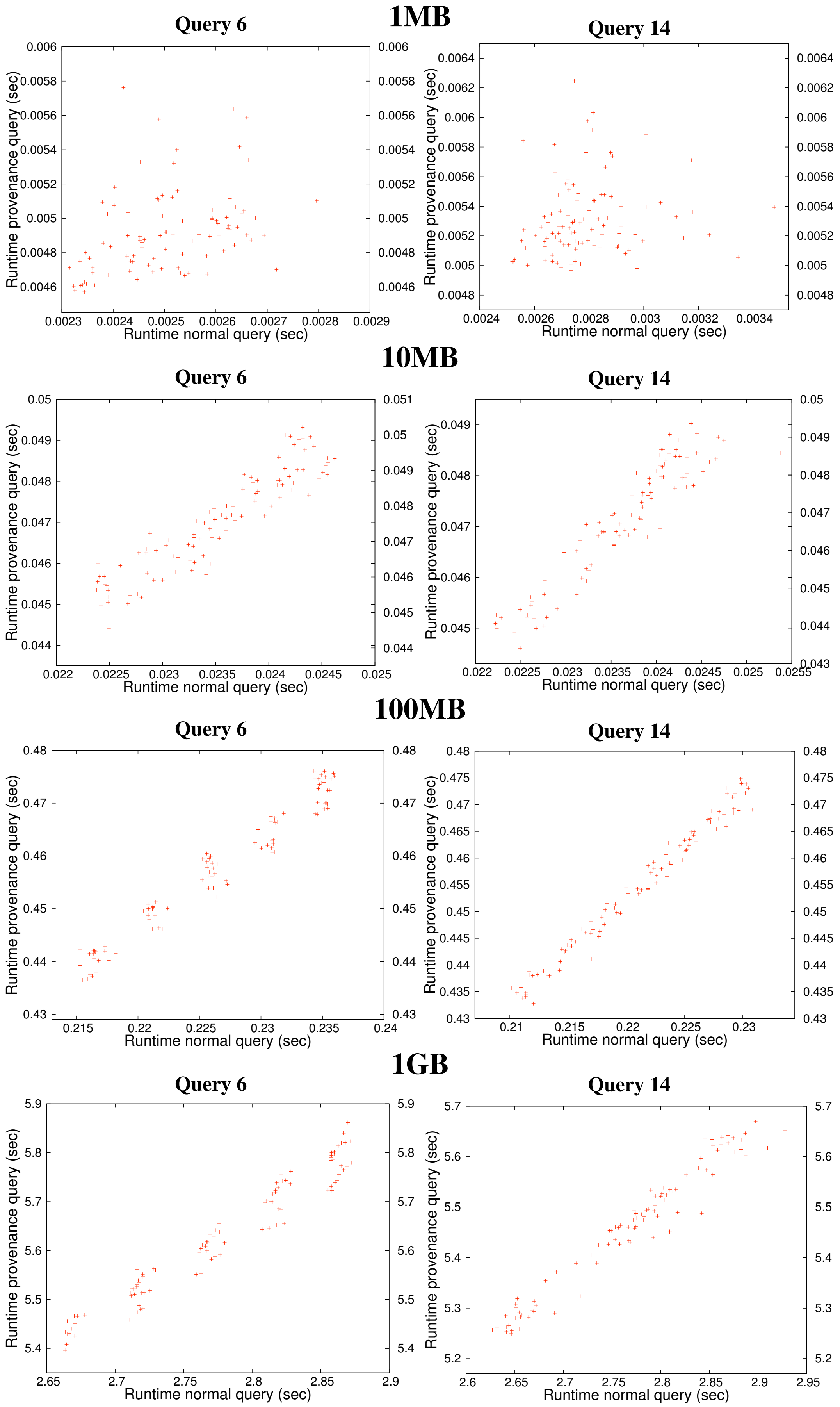


Figure 6.5: Scatter Plots for Queries 6 and 14

6.3.2 Analysis of Query Block Types

The *TPC-H* queries present a well-balanced workload, but to better understand the cost of provenance computation for specific operators we generated simple queries to investigate the influence of the type of query block on the run-time of provenance queries. These queries were executed over the TPC-H data-sets using hardware configuration 1.

6.3.2.1 SPJ Query Blocks

The first set of artificial queries tests the performance of *SPJ* query blocks. Each query is constructed using *numSub* leaf sub-queries. For each query a random query tree is created. A run of this experiment consists of 100 queries. The results indicate that provenance computation of SPJ queries is a very efficient operation. As shown in Figure 6.6, the provenance computation results in a maximal average overhead of a factor of 10. This is expected behavior, because the rewrite algorithm only adds new attributes to the target list of an SPJ query and does not change its structure.

6.3.2.2 ASPJ Query Blocks

In the next set of experiments, the performance of nested aggregation operations is tested. An aggregation test query consists of *agg ASPJ* query blocks. Each query block is an aggregation over the results of its child query block (No joins are used). The leaf operation accesses the TPC-H table *part*. Every aggregation groups the input on a range of primary key attribute values. The ranges are chosen so that each operation performs approximately the same number of aggregation function computations. This is achieved by grouping on the primary key attribute divided by $numGrp = \sqrt[agg]{\mid part \mid}$. Figure 6.6 demonstrates that the execution time grows linear in database size and number of aggregation operations, because each aggregation operation introduces a new join between the original aggregation and the rewritten input to the provenance query.[1]

6.3.2.3 SET Query Blocks

We generated queries consisting only of set operations over simple selections on the TPC-H base table *part* restricted to a random range of primary key attribute values. Each set operation query is a random set operation tree structure with *numSetOp* leaf nodes (Selections on *part*). For one value of *numSetOp*, we measured the average execution time of 100 of these set operation queries. For this set of experiments we applied the *Witness-List* provenance semantics for union (**R6.a**) and used only union and intersection, because the rewrite for set difference prunes the provenance computation for the right input of the set difference (**R8.a**) which dramatically reduces the complexity of the query if the set difference is used in one of the top nodes of the set operation tree. If rewrite rule (**R8.b**) is used for set difference the complexity of the rewritten query is increased, because in the worst case, the computation of a query rewritten with this rewrite rule can degrade to a cross product between the left and right input of the set difference. In this case the number of result tuples grows exponentially in the number of set difference operations. Therefore, we omitted set difference in the experiment to evaluate the effect of the computational complexity of a provenance query instead of the effect of potential exponential result growth. The results presented in Figure 6.6 show that union and intersection operations are handled well by the *Perm* system.

6.3.3 Comparison with Trio

To investigate how *Perm* performs in comparison with other provenance management systems we compared the execution of queries between *Perm* and the *Trio* system. *Trio* was chosen because its source code is freely available and the system is based also on *PostgreSQL*. For these experiments we had to use simple SPJ queries and one level set operations, because other query types are not supported by *Trio*. We generated 1000 simple selections on a range of primary key attribute values of relation *supplier* for the TPC-H

[1] For some values of *agg* the expression *numGrp* yields an integer result. In these cases the group-by expression is of less computational complexity, leading to faster query execution.

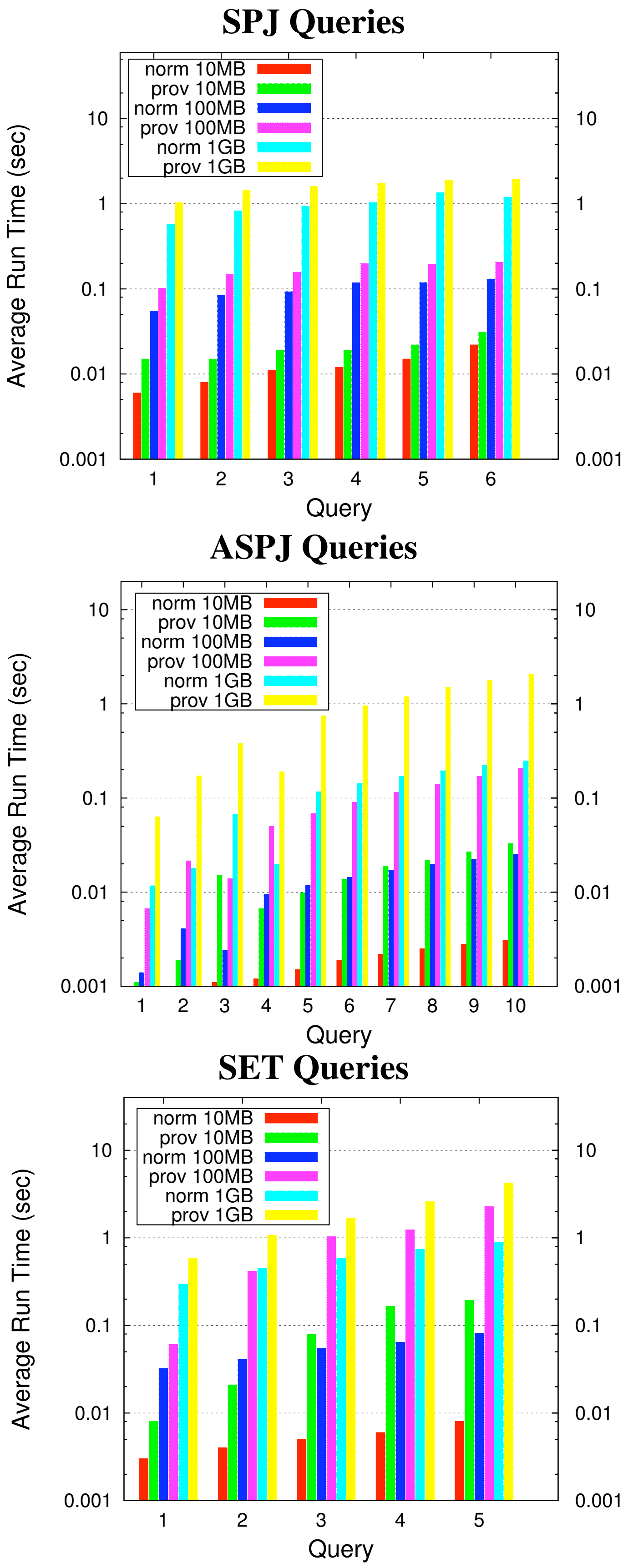


Figure 6.6: SET Query Blocks: Execution Time Comparison

System	10MB	100MB	1GB
Trio	113s	922s	9309s
Perm	3s	25s	249s

Figure 6.7: Execution Time Comparison with Trio

database schema. Figure 6.7 presents the overall execution times in seconds for the complete set of queries. We only measured the run-time of the complete set of queries, because for *Trio* the overhead to start a client application is much larger than for *Perm*. Hence, measuring the execution time of single queries would result in an unfair comparison.

Perm outperforms *Trio* by a factor of at least 30. Note that *Trio* does not support lazy provenance computation, so the provenance was computed beforehand. The measured execution time includes only the time to retrieve the stored provenance. For *Perm* the provenance was computed lazily. In spite of the fact that we did not use the uncertainty management features provided by *Trio*, this feature may account to a certain amount of the observed overhead.

6.4 Evaluation of the Heuristic Rewrite Strategy Selection

The *Perm* system support several rewrite strategies for sublink queries that implement different ways of computing the provenance for such queries. In section 5.5.1.1 we presented the heuristic that is used to decide which strategy will be applied to a query. In this section we investigate the quality of this heuristic. I.e., does the strategy chosen by the heuristic produce the query with the most efficient query plan if compared to the queries generated by the other applicable strategies. Recall that in addition to the heuristic choice of rewrite strategies we implemented a selection of rewrite strategies based on the cost estimation computed by the optimizer. Rewritten queries are produced for all combinations of applicable rewrite strategies, the optimizer is used to generate a query plan for each combination, and the combination with the lowest estimated cost is executed. This approach should not be used in production, because the number of queries that have to be optimized can be exponential in the number of sublinks in a query. However, it is very useful to investigate if the heuristic selects a 'good' rewrite strategy.

We generated execution plans for 1.000 instances of all TPC-H query templates that use sublinks for database sizes between 1MB and 1GB; once using the heuristic and once using the optimizer to find the rewrite strategy that generates the query with the lowest expected run-time. The *EXPLAIN* command of *PostgreSQL* was modified to return the rewrite strategies that were used in addition to the query plan. The result of this experiments confirmed that the heuristic selects the best rewrite strategy. In all cases the two methods used the same rewrite strategies.

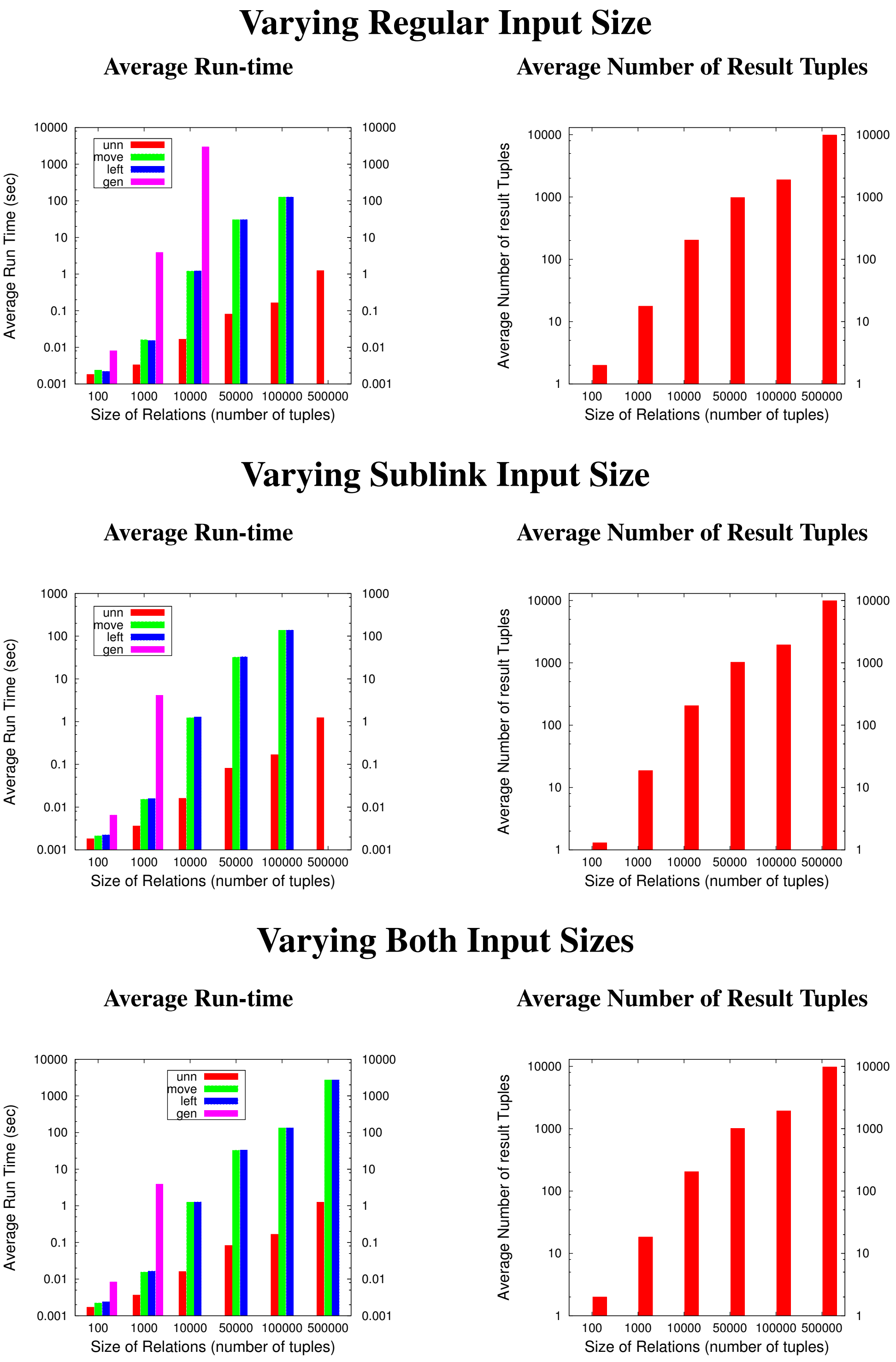
The cost estimation of the optimizer relies on heuristics and statistics about the data stored in the database. Therefore, the actual run-time of a query can differ from the cost estimated by the optimizer. We performed a second set of experiments to verify that the query plans which are identified by the optimizer to have the lowest cost are the ones with the lowest run-times. In these experiments the run-times of the rewritten versions of a query generated by different rewrite strategies are compared. The experiments were run on hardware configuration 2. We produced tables with two integer attributes (a and b) in sizes from 100 to 500.000 tuples. The attribute values were drawn from a gaussian distribution with a fixed mean and a standard derivation of 100 times the table size. For the experiment we used one parametrized query q containing a sublink that uses an *ANY*-sublink in the *WHERE*-clause. The SQL and algebra representations for this query are presented below.

$$q \;=\; \texttt{SELECT} \;*\; \texttt{FROM}\; R_1 \;\texttt{WHERE}\; range \;\texttt{AND}\; \texttt{a} \;=\; \texttt{ANY}\; (\texttt{SELECT} \;*\; \texttt{FROM}\; R_2 \;\texttt{WHERE}\; range2);$$

$$q = \sigma_{range \wedge a\, =\, ANY\ (\sigma_{range2}(R_2))}(R_1)$$

The *range* and *range*2 conditions restrict the input tables on a random range (with a fixed size) of values from attribute *b*. We used the synthetic data-sets (see section 6.1.2.3) to run three sets of experiments. For the first set of experiments we employed a fixed size of 1000 tuples for the relation R_2 used in the sublink and varied the size of the regular input relation of the selection (R_1). In the second set of experiments, we fixed the size of the input relation R_1 and varied the size of the sublink relation R_2. For the last set of experiments the sizes of both relations were varied. All experiments were run for 100 queries of type q. The 100 queries for each configuration were given a time slot of 12 hours. The queries were executed using all rewrite strategies that are applicable for q (*Unn*, *Left*, *Move*, and *Gen*).

The average run times and number of result tuples are given in Figures 6.8. The results demonstrate that the specialized *Unn* strategy outperforms the other strategies by several orders of magnitude. The *Left* and *Move* strategies showed a significant improvement over the *Gen* strategy. The *Move* strategy did not result in the expected improvement over the *Left* strategy - The run-times of both strategies are almost identical. The heuristic for rewrite strategy selection applied the *Unn* strategy in all cases. The results verify the quality of this heuristic. Furthermore, the huge differences in run-times justify the development of specialized rewrite strategies in addition to the *Gen* strategy.



Varying Regular Input Size

Average Run-time

Average Number of Result Tuples

Varying Sublink Input Size

Average Run-time

Average Number of Result Tuples

Varying Both Input Sizes

Average Run-time

Average Number of Result Tuples

Figure 6.8: Comparison of the Run-times for Different Rewrite Strategies

6.5 Discussion

In this chapter we have demonstrated the feasibility of the *Perm* approach with extensive experiments. The questions we presented at the beginning of this chapter could all be answered in favor of *Perm*. The overhead the system introduces for the execution of standard SQL queries is negligible. We could confirm the quality of the heuristic used to select rewrite strategies for sublinks. Provenance computation can be expensive, but the experiments indicate that the cost is intrinsic to this operation and not due to a disadvantageous approach for provenance computation. To the best of our knowledge *Perm* is the first provenance management system that is capable to compute the provenance of all *TPC-H* queries. Our system outperforms *Trio* significantly on the queries supported by this system.

Chapter 7

Using Perm to Debug and Understand Schema Mappings

The *Perm* system integrates powerful provenance functionality into a standard DBMS. In the last chapters we have developed the formal background, discussed the implementation, and evaluated the performance of *Perm*. In this chapter we study the applicability of *Perm* to a common provenance application domain - schema mapping debugging for data integration and data exchange. We choose this application domain, because the usefulness of provenance for it is widely established. Several data exchange systems support provenance computation (e.g., see [GKT$^+$07b, CT06, VMM05]). Therefore, it will be interesting to see how *Perm* compares to these systems on this application domain. *Perm* was originally designed as a generic relational provenance management system not tied to a specific application domain. Hence, the system had to be extended to deal with non-SQL transformations applied in data integration and data exchange. We would like to clarify that only small extensions were necessary to adapt the system to the new use case. With these extensions *Perm* provides novel schema debugging functionality not supported by alternative approaches. We refer to the extended version of *Perm* as *TRAMP* (TRAnsformation Mapping Provenance). In the following we motivate the use case, introduce necessary background information about schema mappings, present a running example, and discuss how provenance information can be used to debug schema mappings (section 7.1). In section 7.2 we discuss the implementation of the schema mapping debugging extensions. Afterwards, we evaluate the practicability of the approach by means of an example debugging process (section 7.3).

7.1 Motivation

Schema mappings, declarative constraints that model relationships between schemas, are the main enabler of data integration and data exchange. Schema mappings are used to translate queries over a *target schema* into queries over a *source schema* (data integration) or to generate *transformations* that produce a target instance from a source instance (data exchange). The executable *transformations* are generated from the declarative, logical specification of the *mappings*. A single transformation may implement a single mapping, several mappings, and a mapping may be implemented by a set of transformations.

Nowadays, mappings are often generated semi-automatically using tools like *Clio* [MHH00, MHH$^+$09], BEA AquaLogic [BBC$^+$09], and many others [ATV08]. The generation of executable transformations from mappings is also largely automated. The complexity of large schemas, lack of schema documentation, and the iterative, semi-automatic process of mapping and transformation generation are common sources of errors. These issues are compounded by limitations and idiosyncrasies of mapping tools (which can produce wildly different transformations for the same input schemas [ATV08]). Understanding and debugging an integration, its mappings and transformations is far from trivial and is often a time consuming, expensive procedure. In addition, schema mapping is often done over data sources that are themselves dirty or inconsistent. Errors caused by the data cannot be neatly and cleanly separated from errors caused by an incorrect mapping or transformation.

As a result, a number of research efforts have emerged to support users in debugging and understanding schema mappings and mapping alternatives (these include SPIDER [CT06] and MXL [VMM05], along with mapping understanding-by-example systems such as the Clio data-viewer [YMHF01] and MUSE [ACMT08]). Such systems focus on fixing or refining the mapping specification.

In contrast, by extending *Perm* for this application domain we developed a more holistic approach that aims at providing a robust data integration debugging tool for tracing errors, no matter what their source (the data, inconsistencies between data sources, the schemas, schema constraints, the mappings, or the transformations). We argue that a robust tool for understanding the behavior of complex schema mappings can be provided only by making all elements of a mapping scenario (schemas, schema constraints, mappings, transformations, and data) and their inter-relationships query-able. We show that query support can be provided by using provenance information on all elements of a mapping scenario.

Perm is an ideal candidate to achieve this goal, because it provides full SQL query support for an extended version of the traditional concept of *data* provenance and the novel concept of *transformation* provenance. As we will demonstrate in the remainder of this chapter, both types of provenance are needed in schema mapping debugging. We show that many queries important to debugging data integrations can only be answered using such a comprehensive approach where all inputs to the integration and all forms of provenance can be used collectively in query answering. *TRAMP* extends *Perm* with the following new functionality for schema mapping debugging:

- Support for *mapping*-provenance, a novel type of provenance information suited for schema mappings.
- A *meta-query* facility that can be used to query mapping and transformation provenance. This facility is fully integrated in SQL, thus, access to all types of data, provenance, and the transformations themselves can be combined in a single query.

7.1.1 Background and Notations

In this section, we introduce schema mappings and the types of transformation queries used to implement data integration or data exchange. Commonly, a schema mapping is modeled as a tuple $\mathscr{M} = (\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_s, \Sigma_t)$ where **S** is a *source* schema, **T** is a *target* schema, and Σ_s (and Σ_t) are sets of *constraints* over **S** (resp. **T**) [Len02, FKMP05]. The constraints Σ_{st} are the mapping(s) representing the relationship between **S** and **T**. In data exchange applications [FKMP05], Σ_s may be omitted since the source instance is given but we include it for generality. In our implementation, the source and target constraints (Σ_s and Σ_t) may be any SQL constraint, including primary keys, unique constraints and foreign keys. Our framework and definitions are general enough to include any constraint, including the commonly studied *tuple-* and *equality-generating dependencies* [AHV95].

The source and target constraints (Σ_s and Σ_t) may be sets of *tuple-generating-dependencies* (*tgd*'s) and *equality-generating dependencies* (*egd*'s). A *tgd* is of the form $\forall \mathbf{x} \phi(\mathbf{x}) \Rightarrow \exists \mathbf{y} \psi(\mathbf{x},\mathbf{y})$ where $\phi(\mathbf{x})$ and $\psi(\mathbf{x},\mathbf{y})$ are conjunctions of atomic formulas over **S** for source constraints and over **T** for target constraints. For example, relational foreign key constraints are modeled as *tgd*'s. An *egd* is of the form $\forall x \phi(x) \Rightarrow x_1 = x_2$ where $\phi(x)$ is a conjunction of atomic formulas and x_1 and x_2 are variables occurring in x. For example, relational primary key constraints are modeled as *egd*'s. Σ_{st} describes the actual mapping. Our implementation assumes Σ_{st} is a set of source-to-target *tuple-generating dependencies* (*s-t tgds*): $\forall \mathbf{x} \phi(\mathbf{x}) \Rightarrow \exists \mathbf{y} \psi(\mathbf{x},\mathbf{y})$ where $\phi(\mathbf{x})$ is an SPJ query over **S** and $\psi(\mathbf{x},\mathbf{y})$ is an SPJ query over **T**.

Schema mappings of this form are quite general and are the basis for most work in data exchange, data integration, peer data sharing, pay-as-you go integration, and other integration tasks. For schema mapping debugging, however, we need to consider a wider range of aspects. In addition to the many possible sources of errors, the tools used in data exchange or data integration that are based on schema mappings often operate in a way that leads to unexpected results. Such unexpected results arise from *ad hoc* choices made by the tools when several solutions are possible (in data exchange systems), from different strategies in query rewriting (in data integration systems), and from optimizations as well as limitations of the languages used to implement the operations.

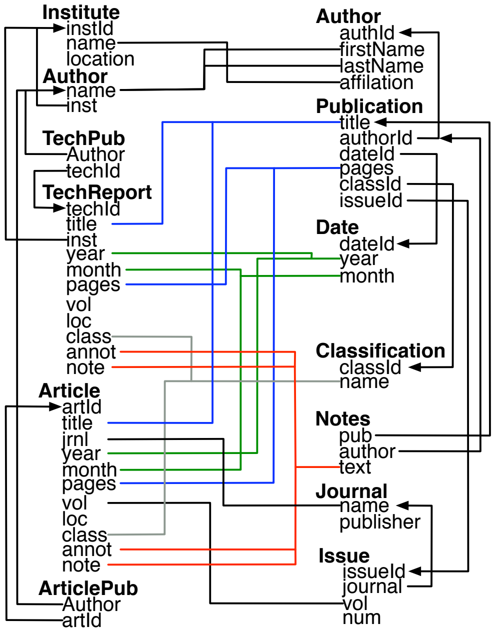
Schema mappings can be generated (semi)-automatically from schema constraints and *correspondences* (or matches) between the source and target schema. A correspondence is of the form $\forall s_1,\ldots,s_n : R(s_1,\ldots,s_n) \Rightarrow \exists t_1,\ldots,t_n : S(t_1,\ldots,t_n) \wedge s_i = t_j$ where R is a source relation, S is a target relation. Informally, a correspondence means that attributes s_i and t_j are related in a sense that data from s_i should appear in t_j. Errors can arise if the schema is missing constraints, if a matching is incorrect, or, in general, if the semantics of a concept that appears in the target has not been modeled explicitly in the source. Nevertheless, understanding the results of schema mappings is difficult even when no errors have been made. We analyze this aspect here in more detail.

More formally, for a given instance $\mathscr{I}$ of the source schema, an instance $\mathscr{J}$ of the target schema is called a *solution* of a schema mapping $\mathscr{M}$ if $\mathscr{J}$ satisfies all the constraints in Σ_t and $\mathscr{I}$, $\mathscr{J}$ satisfy all the constraints in Σ_{st}. In general, there can be many such solutions for a given schema mapping. One of these solutions will be produced through *transformations* that implement the schema mapping.

While data exchange systems such as Clio support a *universal solution* [MHH$^+$09, FKMP05], mapping tools often select a solution from all possible solutions in an *ad hoc* manner, leading to unexpected results. This leads to the definition of "certain" answers. A "certain" answer of a query q is an answer that is included in the result of evaluating q over each possible solution. To compute certain answers for common queries, it is know that some solutions, called *universal solutions* (which can be thought of as a "most general" solution) , can be used directly (see [FKMP05] for a more detailed discussion).

Similarly, in data integration systems (such as a federated DBMS), transformations are actually query rewrites: a transformation of a query over the target into a query (or a set of queries) over the source. Such query rewriting plays the same role as a transformation, but instead of producing a full target instance it produces a view over the target (a view whose definition is the target query). Ad hoc decisions when rewriting the queries lead to several possible outcomes and, hence, to queries that do not produce the expected data. Research data integration systems produce rewritings that compute certain answers, but industrial systems may (as with data exchange) make *ad hoc* decisions in the rewriting. To support the debugging of schema mappings in both scenarios, we will use the term *transformation* to refer to either the queries used to implement data exchange or to the rewritten queries used in data integration. We focus on transformations implemented in SQL, the language for which *Perm* provides provenance support, but the ideas should be generalizable to other implementation languages such as XSLT or Java.

In the general case, there is no one-to-one relation between transformations and mappings. A single transformation may implement several mappings, or a single mapping may be implemented by several transformations. The actual nature of the transformation(s) depends on the application (e.g., the queries used to implement data exchange or query rewrites in data integration) and the actual language used (e.g., SQL, XSLT, or Java). The many-to-many relationship between transformation and mappings is due to restrictions in the implementation language (e.g., an SQL query can produce only one target relation while a mapping may define several target relations) and/or because the splitting or merging of transformations might improve their run-time performance. Such restrictions and optimizations add to the complexity of debugging a schema mapping even when no errors have been made. This is especially true in cases



$\mathbf{M_1}$:$Author(a,b) \Rightarrow \exists c,d : Author(c, first(a), last(a), d)$

$\mathbf{M_2}$:$Author(a,b) \wedge Institute(b,c,d) \Rightarrow$
$\exists e : Author(e, first(a), last(a), c)$

$\mathbf{M_3}$:$TechReport(a,b,c,d,e,f,g,h,i,j,k) \wedge TechPub(l,a)$
$\wedge Author(l,m) \wedge Institute(c,n,o) \wedge Institute(m,p,q) \Rightarrow$
$\exists r,s,t,u : Publication(b,r,s,f,t,u) \wedge Date(s,d,e)$
$\wedge Classification(t,i) \wedge Notes(b,r,j)$
$\wedge Notes(b,r,k) \wedge Author(r, first(l), last(l), p)$

$\mathbf{M_4}$:$Article(a,b,c,d,e,f,g,h,i,j,k) \wedge ArtPub(l,a)$
$\wedge Author(l,m) \wedge Institute(m,n,o) \Rightarrow$
$\exists p,q,r,s,t,u : Publication(b,p,q,f,r,s) \wedge Date(q,d,e)$
$\wedge Classification(r,i) \wedge Notes(b,p,j) \wedge Notes(b,p,k)$
$\wedge Journal(c,t) \wedge Issue(s,c,g,u)$
$\wedge Author(p, first(l), last(l), n)$

Figure 7.1: Example Schemas and Mappings

where a transformation implements multiple mappings or a mapping is implemented by more than one transformation.

To capture all the aspects that influence the result of a schema mapping, we model a *mapping scenario* by explicitly including the transformations $\mathscr{T}$ that implement the schema mappings $\mathscr{M}$ along with the relationship between them, as well as their relationship to the data and its provenance.

Definition 7.1 (Mapping Scenario)**.** *A mapping scenario MS is a tuple* $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_s, \Sigma_t, \mathscr{I}, \mathscr{J}, \mathscr{T}, \mathscr{A}, \mathscr{C})$ *where* $\mathscr{I}$ *is a source instance,* $\mathscr{J}$ *is a target instance.* $\mathscr{C}$ *is the set of correspondences from which* Σ_{st} *is derived from.* $\mathscr{T}$ *is the set of transformations that implement the mappings from* Σ_{st} *(Formally represented as algebra expressions). The binary relation* $\mathscr{A}$ *models the relationship between mappings and transformations. Each element of* $\mathscr{A}$ *associates a mapping M from* Σ_{st} *with a sub-expression* q_{sub} *of an element from* $\mathscr{T}$*.*

This additional information is needed to be able to properly debug schema mapping scenarios regardless of whether the source of the problem are actual errors or limitations and idiosyncrasies of mapping tools.

7.1.2 Running Example

We illustrate the complexity associated with the debugging of schema mappings and associated transformations using an example based on the *Amalgam* [MFH$^+$01] Integration benchmark. Amalgam uses real-world data (containing errors) and several schemas which are not *perfect* in the sense that they may contain modeling errors or may have missing constraints or semantics. Such imperfections may lead a mapping tool to produce imperfect mappings or to miss potential mappings. The *Amalgam* schemas represent bibliographic data from various sources. As a running example we use a modified version of schema S_1 from Amalgam as the source schema and our own new schema as the target schema. Figure 7.1 shows parts of the source (left) and target schema (right) as well as some of the *correspondences* between the schemas. The source schema models authors, technical reports, articles, and the relationships between authors and their publications. The target schema represents the same information organized in a different way. Several properties of a publication are outsourced into separate relations and there is only a single relation that represents publications (regardless of their type). In addition, the affiliation of an author is recorded in the

$$\begin{aligned}
\mathbf{M_{2a}} &: Author(a,b) \wedge Institute(b,c,d) \Rightarrow \exists e : Author(e,a,a,c) \\
\mathbf{M_{3a}} &: TechReport(a,b,c,d,e,f,g,h,i,j,k) \wedge TechPub(a,l) \wedge Author(l,m) \\
&\quad \wedge Institute(c,n,o) \wedge Institute(m,p,q) \Rightarrow \\
&\quad \exists r,s,t,u : Publication(b,r,s,f,t,u) \wedge Date(s,d,e) \wedge Classification(t,i) \\
&\quad \wedge Notes(b,r,j) \wedge Notes(b,r,k) \wedge Author(r,l,l,n)
\end{aligned}$$

Figure 7.2: Incorrect Mappings For the Example

Author relation directly, rather than in a separate *Institute* relation. The example includes the following basic scenarios defined in the *STbenchmark* [ATV08]: *copying*, *vertical partitioning*, *manipulating atomic values*, and *Object Fusion*.

A possible mapping between the two schemas is presented on the right of Figure 7.1. Mappings $\mathbf{M_1}$ and $\mathbf{M_2}$ map authors from the source to the target schema, splitting the *name* attribute into first and last name.[1] $\mathbf{M_1}$ handles authors independent of their affiliations and $\mathbf{M_2}$ maps authors with an affiliation (stored in the relation *Institute*). The *Techreport* and *Article* relations are vertically partitioned into relations *Publications*, *Journal*, *Issue*, *Notes*, *Classification*, and *Date* (using $\mathbf{M_3}$ and $\mathbf{M_4}$). In addition, the authors for each publication are also mapped by these two mappings. In mapping $\mathbf{M_3}$ the *Institute* relation is referenced twice. The first reference represents the institute of an author and the second one the institute field of the *TechReport* relation. Note that these mappings might lead to missing data if, for example, a technical report is not associated to an institute (i.e., its *inst* attribute is null). For simplicity, however, we assume mappings $\mathbf{M_1}$-$\mathbf{M_4}$ are the correct mappings.

7.1.3 Types of Errors

To illustrate common types of errors, we consider two incorrect mappings ($\mathbf{M_{2a}}$, $\mathbf{M_{3a}}$), shown in Figure 7.2 in addition to the correct mappings ($\mathbf{M_1}$, $\mathbf{M_2}$, $\mathbf{M_3}$, $\mathbf{M_4}$). Errors in an mapping scenario can be categorized according to whether they are caused by incorrect mappings, erroneous transformations, or other causes.

7.1.3.1 Mapping Errors

Missing Mappings. A mapping missing from a mapping scenario may lead to empty target relations or incomplete target relations. In our example from Figure 7.2 we are missing mappings $\mathbf{M_1}$ and $\mathbf{M_4}$. Missing mappings may be caused by missing correspondences, missing schema constraints, misinterpreting the semantics of a relation, or problems understanding mappings and schemas. Correspondences can be missed through errors by a matching tool [RB01]. A missing schema constraint, such as the foreign key constraint from *Author* to *Institution* leads to missing $\mathbf{M_2}$ or $\mathbf{M_{2a}}$ as it is not obvious how to associate an author with an affiliation. Misinterpreting the meaning of the relation *Publication* (not realizing that an article is also a publication) leads to missing mapping $\mathbf{M_4}$. Finally, not realizing that some authors might not be associated to an *Institute* leads to missing mapping $\mathbf{M_1}$.

Incomplete Mappings: Incomplete mappings (i.e., mappings that are missing relations or missing conditions) may also arise due to missing correspondences or missing schema constraints. For instance, if the source includes publications from authors around the world, and in the target we desire publications for authors from a specific region, then mapping $\mathbf{M_2}$ would be missing a condition on location (or missing a join with a relation *Region* on which we could specify this condition).

Oversized mappings: Oversized mappings (that is, mappings with too many relations) may be caused by using an association in the source with different semantics than the corresponding association in the target. In our example, if the *inst* of an *Author* in the source represents a *works-in relationship*, but the *affiliation* of an *Author* in the target represents the PhD granting institution, then mapping $\mathbf{M_2}$ would be oversized. Thus, an oversized mapping associates relations that should not be associated according to the desired mapping semantics.

[1] We assume the existence of functions *first* and *last* that return the first (respectively, last) name from a name string.

Incorrect Association Paths: Schema constraints, query logs, or even connections in the data may be used by mapping systems to figure out how to join source or target relations in a mapping. In a schema there might be several ways to reach one relation from another via connecting constraints, but not all of them represent a semantically correct way of associating source relations. In the example scenario, there are two ways of associating the relations *Author* and *Institute*. Mapping $\mathbf{M_{3a}}$ incorrectly maps the institute of a technical report to the *affiliation* attribute of an author in the target schema.

7.1.3.2 Transformation Errors

The most commonly used mapping languages today (including the s-t tgds we consider) are constraints that specify properties that must be true of the transformation. However, they do not determine a unique target instance, and hence do not determine a unique transformation. This is necessary in practice for many reasons including incompleteness (the source may not populate all target data). As a result, new types of errors arise due to the fact that a mapping can be handled in different ways by the transformations that implement the mapping.

Incorrect Handling of Atomic Values: This type of error arises if either a non-atomic attribute is handled as an atomic one or if incorrect functions are applied to split a non-atomic attribute value. In the example scenario, mappings $\mathbf{M_{2a}}$ and $\mathbf{M_{3a}}$ generate the target *Author* relation by copying the *name* attribute unmodified to the *firstName* and *lastName* attributes instead of applying functions *first* and *last* to extract the relevant part of the attribute value.

Redundant Data: Consider mappings $\mathbf{M_1}$ and $\mathbf{M_2}$ in the example. If the transformation generating target relation *Author* is implemented as a set-union between the transformations implementing $\mathbf{M_1}$ and $\mathbf{M_2}$, then authors with an affiliation will be recorded twice in the *Author* relation: once with the affiliation and once without an affiliation. A user may not want the redundant, incomplete tuples.

7.1.3.3 Other Errors

Our motivation is to provide a tool to help find and repair errors in mappings and transformations. To do this well, we need to consider errors in elements that may be used as input for mapping creation tools - specifically, the data, schemas, schema constraints, and correspondences. In talking about mapping and transformation errors, we have already motivated how errors in these inputs can lead to errors in the mapping. Another common source of error is dirty source data:

Instance Data Errors: Incorrect source instance data can cause errors in the target instance. Instance data errors may confuse a user and lead her to conclude a correct mapping is incorrect. Consider a source instance with dirty data. If the mapped target data lists Alonso's affiliation as UCSB, a user who knows this is incorrect may mistakenly think the mapping is incorrect because it is using a source association that represents authors' PhD institutions rather than a works-in relationship. However, if the data is dirty, it may just be that this one value is out-of-date. Hence, to really help a user understand when a mapping/transformation is correct, we need to recognize that errors can arise from many different sources.

7.1.4 Tracing Mapping Errors

We now present which types of provenance (and other information) support a user in debugging which types of errors in a mapping scenario.

7.1.4.1 Data Provenance

Data provenance helps in tracing erroneous target data back to erroneous source data and in understanding mapping errors that are caused by mapping data from the wrong sources. It can also be used to trace *incorrect handling of atomic values*-type errors. For instance, the *data* provenance of the transformation implementing mapping $\mathbf{M_{2a}}$ will reveal that the *firstName* and *lastName* attributes are copied from the *name* attribute of the source *Author* relation. The user can then introduce the *first* and *last* functions into the mappings to extract the relevant part of the *name* attribute. As another example, *data* provenance can also be used to trace *Incomplete mappings* by examining to which source relations the provenance of a result belongs. For *Oversized Mappings* and *Incorrect Association Paths data* provenance can be used to understand where the incorrect data is coming from. For instance, in the example scenario the *data* provenance of the transformation implementing mapping $\mathbf{M_{3a}}$ reveals that the *affiliation* data in the *author* relation is copied from the institute associated with a technical report instead of the institute associated with an author.

7.1.4.2 Transformation Provenance

Data provenance relates output and input data, but does not provide any information about how data was processed by a transformation. More specifically, it does not contain information about which parts of a transformation were used to derive an output tuple. As an example, consider a transformation that uses the SQL *union all* operator. Each output tuple of the union is produced by exactly one of the queries that are input to the union. Recall that we refer to this type of provenance as *transformation* provenance (see sections 3.3.2 and 4.6).

Transformation provenance is extremely useful in understanding how data is integrated through a mapping, because it allows us to understand which parts of a transformation (that is, which operators in the transformation) produced a data item. Naturally, *transformation* provenance is very useful to debug *transformation* errors (*Incorrect Handling of Atomic Values* and *Redundant Data*). Also *Oversized mappings* and *Incorrect Association Paths* errors can be seen in the transformation that implements these mappings and can therefore be debugged using *transformation* provenance. E.g., if all erroneous data is produced by a certain part of a transformation this part of the transformation should be investigated more closely.

7.1.4.3 Mapping Provenance

In a mapping scenario each transformation implements one or more mappings and each mapping is implemented by one or more transformations. To be able to understand such a mapping scenario it is crucial to know which parts of a transformation correspond to which mapping. *Mapping provenance* enriches *transformation* provenance with a representation of this relationship. *Mapping* provenance can be used to understand which of the mappings implemented by a transformation produced erroneous data, thus, limiting the scope of the error tracing process and can be used to trace the same types of errors as *transformation* provenance.

7.1.4.4 Meta-Querying

Meta-querying [VdBVV05] is the ability to ask queries over the structure and/or run-time properties of queries. This type of functionality is useful to investigate static properties of a transformation in a mapping scenario, for instance, to investigate which relations are accessed by a transformation. In the example scenario we can use *meta-querying* to investigate which relations are accessed by any transformation (of which there may be many) generating *Publication* tuples. In our example of Figure 7.1(b), such a query reveals that the *article* relation is not accessed by any transformation, and therefore a new mapping should be added to map the data from this relation. In general *meta-querying* can be used to trace all types of *mapping errors* if they are understandable without data and run-time information. Therefore, *transformation* errors are normally not trace-able by using solely *meta-querying*.

Powerful new query functionality is gained by combining *meta-querying* with provenance and mapping meta-data. This combination is needed, e.g., to answer queries like “Is a certain tuple in the result created by a part of a transformation that accesses the relation Author?” and to isolate the parts of a transformation or mapping that caused a given error. Recall that in *SQL-PLE* the keyword *TRANSXML* is used to return *transformation* provenance as XML data. This keyword was added to enable *meta-querying* of *transformation* provenance information.

7.1.4.5 Discussion

In summary, to understand the reasons for errors in a mapping scenario it is crucial to know where generated data is coming from (*data* provenance) and how it was processed (*transformation* and *mapping* provenance). In addition to exposing this information, a system that aides a user in understanding a mapping scenario should provide appropriate query facilities like *meta-querying* to process this information in useful ways.

- **Data provenance**: Aides in tracing erroneous target data back to erroneous source data and in understanding mapping errors that are caused by mapping data from wrong sources.
- **Transformation and mapping provenance**: Exhibits errors in transformations and underlying mapping by explaining which parts of a transformation and which mappings are responsible for a certain transformation result.
- **Meta-querying**: Can be used to investigate transformation provenance, to understand instance independent problems in a transformation, and to relate mappings and transformations.

(a)

$$q_a = \Pi^{2}{}_{SK_1(name),first(name),last(name),SK_2(name)}(A^{3})$$
$$\cup^{1}\ \Pi^{4}{}_{SK_3(A.name,I.name),first(A.name),last(A.name),I.name}(A^{6} \bowtie^{5}{}_{inst=instId} I^{7})$$
$$q_b = \Pi^{1}{}_{SK_1(A.name,I.name),first(A.name),last(A.name),I.name}(A^{3}]\!\bowtie^{2}{}_{inst=instId} I^{4})$$

(b)

$$\mathscr{M}(q_b,t_1) = \{\{M_2\}\}$$
$$\mathscr{M}(q_b,t_2) = \{\{M_1\}\}$$
$$\mathscr{M}(q_b,t_3) = \{\{M_2\}\}$$

(c)

$\mathbf{q_a}$

$$\mu_{M_1}(op) = \begin{cases} 1 & \forall op \in \{1,2,3\} \\ 0 & \forall op \in \{4,5,6,7\} \end{cases}$$

$$\mu_{M_2}(op) = \begin{cases} 1 & \forall op \in \{1,4,5,6,7\} \\ 0 & \forall op \in \{2,3\} \end{cases}$$

$\mathbf{q_b}$

$$\mu_{M_1}(op) = \begin{cases} 1 & \forall op \in \{1,2,3\} \\ 0 & \forall op \in \{4\} \end{cases}$$

$$\mu_{M_2}(op) = 1$$

(d)

Author

name	inst
Peter Smith	1
Helga Hauser	ε
Hilde Schneider	2

Institute

instId	name	location
1	A	Denver
2	B	Dallas

$\mathbf{Q_b}$

authId	firstName	lastName	affiliation
$SK_1(PeterSchmith,A)$	Peter	Smith	A
$SK_1(HelgaHauser,\varepsilon)$	Helga	Hauser	ε
$SK_1(HildeSchneider,B)$	Hilde	Schneider	B

Figure 7.3: Mapping Provenance Example

7.2 TRAMP: Schema Debugging Extensions For Perm

We now define mapping provenance and discuss the implementation of the *TRAMP* extensions to *Perm*. In addition to *mapping* provenance, *TRAMP* also adds several new language constructs for *meta-querying* to *SQL-PLE*.

7.2.1 Mapping Provenance Definition

In a mapping scenario, transformations may be derived from a set of declarative schema mappings. For example, consider the mappings $\mathbf{M_1}$ and $\mathbf{M_2}$ from the motivation. Two possible implementations of these mappings are presented in Figure 7.3(a) (we use A and I as a shortcut for the *Author* and *Institute* relations). In our example, SK_1, SK_2, and SK_3 are skolem functions which may be generated by a mapping system or user-defined functions that are used to fill in values for un-mapped attributes. Notice that both q_a and q_b implement M_1 and M_2, but they are not equivalent.

They may produce different target data for the same source instance. Both actually produce *universal solutions* [FKMP05], and so may be generated by mapping tools that are based on data exchange theory [MHH$^+$09, RBC$^+$08, and others]. M_2 may produce a less redundant target, but both these and other transformations may be generated by a mapping tool. Hence, in debugging mappings and transformations, we would like to know not only what parts of a transformation produced a target tuple t (the *transformation* provenance of t), but also from what mappings these transformations (or operators withing a transformation) were derived. Hence, we define mapping provenance based on transformation provenance and the correspondences between transformations and mappings (the $\mathscr{A}$ relation of a mapping scenario MS).

Example 7.1. *For the first transformation q_a, the correspondence between mappings and transformations is quite clear. The union and the left input of the union corresponds to mapping* $\mathbf{M_1}$ *in the sense that they process tuples to create target data as specified by* $\mathbf{M_1}$*. The union and its right input corresponds to mapping* $\mathbf{M_2}$*. For the second transformation q_b, the entire transformation implements mapping* $\mathbf{M_2}$*, while* $\mathbf{M_1}$ *is implemented by the join with only its left input (A) and the final projection which provides values for unmapped target elements and implements functions specified in both mappings.*

The *mapping* provenance of an tuple t that was generated by some transformation q should contain all the mappings that contributed to t indirectly by corresponding to a part of the transformation that is in the transformation provenance of t. Therefore, mapping provenance can be defined on-top of transformation provenance.

A correspondence between a mapping and a part of a transformation is modeled by adding additional annotations (specifically, mapping identifiers) to the algebra tree for a transformation. For an algebra tree, $Tree_q = (V,E)$, we introduce one new annotation function, μ_M, per mapping $M \in \Sigma_{st}$. The function μ_M is 1 for each operator that implements this mapping, 0 otherwise.

Example 7.2. *For example, consider transformation q_a in Figure 7.3(a) and the annotation functions for this transformation (Figure 7.3(c)). We use the superscript (red) preordering to refer to the individual operators of a query (these are actually the node identifiers for algebra tree nodes used for transformation provenance). The operator 3 (representing the relation A), implements M_1 and tuples from A may flow through every operator above 3 in the tree so μ_{M_1} annotates every operator on the path from 3 (A) to the root with a 1, and all other operators with a 0. Similarly, the operators 6 (A) and 7 (I) implement M_2, as does every operator above these two in the tree. So μ_{M_2} annotates each of the nodes 1,4,5,6,7 with a 1 and all other nodes with a 0.*

Figure 7.3(c) also presents the two mapping annotation functions for transformation q_b. Here, there is a single node for A (Node 3) which implements both M_1 and M_2. The node for I (Node 4) implements only M_2. Hence, μ_{M_1} assigns a 1 to operator 3 and all nodes above 3 in the tree, and μ_{M_2} assigns a 1 to operator 4 and all nodes above 4 in the tree.

Notice that the mapping annotation function will depend on the language used for mappings. We have implemented mapping annotation functions for source-to-target tgds, but of course this could be extended to other languages, including the visual mapping languages of some commercial tools. Below we formalize the notation of *mapping* provenance using the annotation functions μ_M (Recall that θ_w is the *transformation* provenance annotation function for witness list w).

Definition 7.2 (Mapping Provenance)**.** *The mapping provenance $\mathcal{M}(q,t)$ for a tuple t from the result of query q is defined using the mapping annotation functions μ_M over the transformation provenance $\mathcal{T}(q,t)$ as follows:*

$$\mathcal{M}(q,t) = \{\mathcal{M}_w \mid w \in \mathcal{DD}(q,t)\}$$
$$\mathcal{M}_w = \{M \mid \forall op \in V : \mu_M(op) = \theta_w(op)\}$$

Example 7.3. *As an example of mapping provenance consider Figure 7.3(d), which presents an instance of the Author and Institute relation, the result of applying q_b on this instance, and the mapping provenance (b) for each result tuple of q_b.*

7.2.2 Implementation of TRAMP

TRAMP extends *Perm* with the following functionality: Computation of *mapping* provenance, relational representation of mapping scenario information, and *meta-querying*. In the following we first present the relational representation of mapping scenario information. This is the information that is not already stored as relations; e.g., the source and target instances. The relational representation of correspondences between

```
SELECT
    SK1(A.name, I.name), first(A.name), last(A.name), I.name
FROM
        Author A ANNOT('M1','M2')
    LEFT JOIN
        Institute I ANNOT('M2')
    ON (A.inst = I.instId);
```

Figure 7.4: Example Use of ANNOT

mappings and the transformations that implement them is used in the computation of *mapping* provenance presented afterwards. Finally, we discuss the implementation of *meta-querying* constructs and present an overview of the new *SQL-PLE* language features added by *TRAMP*.

7.2.2.1 Relational Representation of Mapping Scenario Data

A mapping scenario *MS* after definition 7.1 is a tuple $(\mathbf{S},\mathbf{T},\Sigma_{st},\Sigma_s,\Sigma_t,\mathscr{I},\mathscr{J},\mathscr{T},\mathscr{A},\mathscr{C})$. The source and target instances are already represented in the relational model. The source and target schemas are stored in the database catalog of the source and target databases. Also the constraints defined on both schemas (Σ_s and Σ_t) can be accessed over the database catalog. The transformations $\mathscr{T}$ that implements the mappings from the scenario can be stored as views. *PostgreSQL* provides functionality to access the definition of views. Using $f_{SQL \rightarrow XML}$, a function we will introduce in the discussion of *meta-querying*, the textual representation of a transformation's view definition can be transformed into an XML representation.

Example 7.4.
The XML representation of the SQL query SELECT a.name FROM Author a WHERE a.inst = 'MIT'; is presented at the top of Figure 7.5. The mappings Σ_{st} and correspondences $\mathscr{C}$ are stored as XML data. Figure 7.5 shows an part of the XML representation for the example mapping scenario.

To represent $\mathscr{A}$, the correspondences between mappings and transformations, SQL is extended with the ability to annotate parts of a query using the new keyword *ANNOT*. These annotations are used to represent the correspondences between mappings and transformations as defined by the annotation functions μ_M presented in section 7.2.1. If the transformations are generated from a set of *s-t tgds* mappings, we can automatically annotate the base relations of a transformation according to the mappings.[2] For other mapping languages, a user can specify the appropriate mapping annotations of a transformation or the mapping system can be changed to create these annotations. These annotations are only used for *mapping* provenance computation and *meta-querying*. During normal query processing the annotations are ignored.

Example 7.5. *Figure 7.4 demonstrates how the new keyword ANNOT annotates parts of query q_b from Figure 7.3.*

7.2.2.2 Mapping Provenance

To implement *mapping* provenance, a new keyword *MAPPROV* is added to *SQL-PLE*. If a query uses this keyword, the provenance of the result is represented as the list of contributing mappings (textual representation). This is similar to the relational representation of *transformation* provenance where provenance information is stored in a single attribute. If the query uses the keyword *TRANSSQL* or *TRANSXML* then the mapping provenance is computed as part of the transformation provenance as follows.

To support *mapping* provenance the transformation provenance computation is modified to use the annotations added with the *ANNOT* keyword in the final representation construction. We represent the

[2] As described in section 7.2.1 the annotations for the other operators in the query can be derived by propagating an annotation on a child to its parent.

(a)

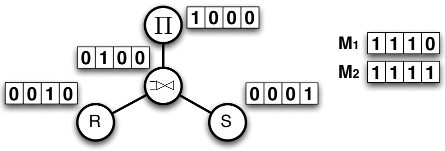
```
<query>
   <select>
      <attr>a.name</attr>
   </select>
   <from>
      <table alias=”a”>Author</table>
   </from>
   <where>
      <op name=”=”>
         <attr>a.inst</attr>
         <const>MIT</const>
      </op>
   </where>
</query>
```

(b)

```
<MappingScenario>
   <Correspondence>
      <from>
         <Table name=”Author”>
            <Column>name</Column>
         </Table>
      </from>
      <to>
         <Table name=”Author”>
            <Column>firstName</Column>
         </Table>
      </to>
   </Correspondence>
...
   <Mapping id=”M1”>
      <foreach>
         <Table name=”Author”>
            <Var>a</Var>
            <Var>b</Var>
         </Table>
      </foreach>
      <exists>
         <Table name=”Author”>
            <Var>c</Var>
            <Var>first(a)</Var>
            <Var>last(a)</Var>
            <Var>d</Var>
         </Table>
      </exists>
   </Mapping>
...
</MappingScenario>
```

Figure 7.5: XML Representation Examples

(a)



(b)

authId	firstName	lastName	affiliation	tprov
$SK_1(PeterSchmith, A)$	Peter	Smith	A	<M2>SELECT SK_1(A.name, I.name), ... FROM ... </M2>
$SK_1(HelgaHauser, \varepsilon)$	Helga	Hauser	ε	<M1>SELECT SK_1(A.name, I.name), ... LEFT JOIN <Not>Institute I</Not> ON ... </M1>
$SK_1(HildeSchneider, B)$	Hilde	Schneider	B	<M2>SELECT SK_1(A.name, I.name), ... FROM Author A LEFT JOIN Institute I ON ... </M2>

Figure 7.6: Example of the Computation of Mapping Provenance

mapping annotation function μ_M for each mapping **M** as a bit-vector (compare sections 5.2.3.3, 5.2.3.2, and 5.5.4) that contains a zero at a position, if $\mu_M(op) = 0$ for the operator op with this identifier/position, and a one otherwise. Recall that a mapping belongs to the transformation provenance iff its mapping function μ_M agrees with θ_w on every input ($\mu_M(op) = \theta_w(op)$). Translated to the bit-vector representations of μ_M and θ_w this is an equality-check on these bit-vectors. In the final result representation, mappings are represented as additional annotations in the representation. E.g., SELECT ... FROM <M1>R</M1> ...

Example 7.6. *Figure 7.6(a) presents the bit-vectors for the support functions μ_{M_1} and μ_{M_2} for query q_b from Figure 7.3. Recall that* $\mathbf{M_2}$ *corresponds to the complete query and* $\mathbf{M_1}$ *does not correspond to the right input of the outer join. Figure 7.6(b) presents the result of computing the transformation and mapping provenance for q_b using the TRANSSQL keyword.*

7.2.2.3 Meta-Querying

In principle we follow the approach for *meta-querying* presented by Van den Bussche et al. [VdBVV05]. In this work queries are represented as XML data. The built-in XPath and XSLT support of *PostgreSQL* is the basis for our implementation of *meta-querying*. This approach has the advantage that the hierarchical structure of a query is reflected well in the XML representation. In addition, with XPath and XSLT powerful meta-querying functionality is provided. The XML query representation used in *TRAMP* is similar to the one from [VdBVV05].

XML Query Representation Generation We provide a function $f_{SQL \rightarrow XML}$ that transforms an SQL query into its XML representation as presented in Figure 7.5. $f_{SQL \rightarrow XML}$ is implemented as a UDF that uses the PostgreSQL parser and analyzer to parse its input and generates the XML output by traversing the internal query representation produced by the parser.

THIS Construct With the new construct *THIS* a query can inspect its own XML representation. This is implemented as a query rewrite that replaces the *THIS* construct with an XML constant that is generated using the same XML generation as $f_{SQL \rightarrow XML}$ for the query the construct is used in (or of a part of this query). *THIS* simplifies *meta-queries* that need introspection. The *THIS* construct contains an optional path expression of the form *child* $<$ *sequence* $>$ that is used to reference a part of the query. For example, THIS.child1 references the first sub-query in the *FROM*-clause of the current query.

(a)

```
SELECT getAnnots(f_SQL→XML(get_view(v1)));
```

(b)

```
SELECT *
FROM (SELECT TRANSXML * FROM v1) AS sub
WHERE hasAnnot(tprov, 'M1');
```

(c)

```
SELECT *
FROM (SELECT * FROM v1) AS sub
WHERE hasAnnot(THIS.child1, 'M1');
```

Figure 7.7: Meta-Querying Example

Example 7.7. *In the query presented below THIS is expanded to the XML representation of the subquery sub:*

```
SELECT THIS.child1, *
FROM (SELECT * FROM R) AS sub;
```

XML Aggregation Function *PostgreSQL* already provides an aggregation function for XML data that combines XML documents into a single document. For example, the query ”Which relations are created by mapping $\mathbf{M_1}$?” can be expressed as an XPath expression over an XML document that contains all transformation definitions, which in turn can be produced by aggregating the XML documents for each transformation.

Pre-defined XSLT- and XPath-Functions XSLT functions combined with the support functions presented above provide all the needed *meta-querying* functionality, but for convenience we provide several additional *meta-querying* UDF functions build upon this functionality. Here we present just two examples: *hasAnnot* and *getAnnots*. hasAnnot(XML,annot) checks if the query represented as parameter *XML* has the annotation *annot*. This function can be realized as a XPath expression over the XML query representation. getAnnots(XML) returns all annotations used in a query tree given as parameter XML.

Example 7.8. *Figure 7.7 shows some example queries that use the meta-querying functionality developed in this section. The query in 7.7(a) demonstrates a query that extracts the annotations from a view definition.*[a]*. Figure 7.7(b) demonstrates how to use meta-querying on the result of a transformation provenance computation. This query returns tuples with their transformation provenance, if the provenance contains annotation M1. Finally, 7.7(c) shows an application of the THIS construct.*

[a] *get_view* returns the SQL text of a view

7.2.2.4 Overview of the SQL-PLE Extensions

Figure 7.8 gives an overview of the new *SQL-PLE* language constructs added by *TRAMP*. The *MAPPROV*, *THIS*, and *ANNOT* constructs have been discussed above. *query_sql_to_xml* is the implementation of the $f_{SQL \rightarrow XML}$ function. The *XSLT.f* construct is added for convenience. In *PostgreSQL* XSLT functions are applied by passing the function definition and its argument as XML to a generic *XSLT* processing function. The *XSLT.f* construct simplifies this process by allowing to directly call XSLT functions (the function definitions and names have to be stored in relation *xslt_funcs*). *cxpath* allows XPath expressions to be used

Construct	Description
MAPPROV	Compute the *mapping* provenance and represent it as sets of mappings.
THIS.child*path*	A shortcut for generating the XML representation of the sub-query accessed by *path*.
XSLT.*f* (xml_param)	Apply XSLT function *f* to XML document `xml_param`.
cxpath (path_expr , input)	Evaluate XPath expression `path_expr` over `input`. Returns true if the evaluation of the XPath expression returns at least one result.
ANNOT(annotation)	Annotates the *FROM* clause item it is appended to with `annotation`.

Figure 7.8: New SQL-PLE Language Constructs

as conditions. It returns true if the XPath expressions given as parameter *path_expr* returns more than zero elements if evaluated over parameter *input*.

T_1

```
SELECT
    SK1(a.name, i.name) AS authId,
    a.name AS firstName,
    a.name AS lastName,
    i.name AS affiliation
FROM
    source.author ANNOT('M2a') a,
    source.institute ANNOT('M2a') i
WHERE a.inst = i.inst_id
```

T_2

```
SELECT
    SK1(a.name, i.name) AS authId,
    first(a.name) AS firstName,
    last(a.name) AS lastName,
    i.name AS affiliation
FROM
    source.author ANNOT('M2') a,
    source.institute ANNOT('M2') i
WHERE a.inst = i.inst_id
UNION
SELECT
    SK2(a.name) AS authId,
    first(a.name) AS firstName,
    last(a.name) AS lastName,
    SK3(a.name) AS affiliation
FROM
    source.author a ANNOT('M1')
```

Figure 7.9: Implementing Transformations

7.3 Evaluation

We demonstrate the usefulness of *TRAMP* by means of an example debugging process based on the erroneous mappings presented in the motivation ($\mathbf{M_{2a}}$, $\mathbf{M_{3a}}$ shown in Figure 7.2). Recall that instead of the correct mappings $\mathbf{M_1}$ and $\mathbf{M_2}$ the user has chosen mapping $\mathbf{M_{2a}}$, which incorrectly copies the name attribute from the source to the first and last name attributes in the target. We assume that the mapping system generated the implementing transformation T_1 (Figure 7.9) for this mapping.

The user might recognize that the first and last name attribute both contain the same complete name and that several authors he expected to be present in the target are missing. (S)he then decides to investigate the source of this error. First, it would make sense to check why the name attribute value is duplicated. So the user issues the following query to understand which source data was used to produce the *Author* relation tuples.

```
SELECT PROVENANCE * FROM target.author;
```

If the target instance is large the result size of the query can be reduced by using a *WHERE* clause condition to search, e.g., for an author known to the user like:

```
... WHERE firstName = 'Jim␣Gray'
```

This query reveals that a target author is derived from a source author and the name attributes seem to be copied. Therefore, the error in the target instance is not caused by erroneous source data. This means that the mapping (or its implementation) have to be changed to extract the first and last name parts. The query presented below identifies the mappings that have to be changed.

SELECT getAnnot($f_{SQL\rightarrow XML}$ ('SELECT␣*␣FROM␣target.author'));

This query retrieves the mapping annotations from the transformation that generates the target *Author* relation by first generating the XML representation of this query using $f_{SQL\rightarrow XML}$ and second extracting the mapping annotations using the XSLT function *getAnnot*. This query reveals that mapping $\mathbf{M_{2a}}$ generates the *Author* relation. Correcting mapping $\mathbf{M_{2a}}$ means replacing it with $\mathbf{M_2}$ which changes the implementing transformation by adding the *first* and *last* functions to extract the name parts.

Afterwards the user can focus on the missing authors. A query over the provenance of the *Author* relation can be asked that returns all source authors that are not in the *data* provenance of any target author tuple:

```
SELECT *
FROM source.author
WHERE name NOT IN
    (SELECT prov_source_author_name
    FROM
        (SELECT PROVENANCE * FROM target.author) p);
```

Note that this kind of query over provenance is only possible because *Perm* represents provenance as complete tuples in a relational format. The authors returned by this query all have no affiliation.[3] Therefore, it is obvious that a new mapping is needed to map authors without an affiliation. This leads to the generation of mapping $\mathbf{M_1}$.

For sake of the example assume that the mapping system uses transformation $\mathbf{T_2}$ to implement mappings $\mathbf{M_1}$ and $\mathbf{M_2}$. This transformation generates two versions of an author who has an affiliation, one with the affiliation and one without an affiliation. The user might recognize this problem and want to resolve it. To understand why author duplicates are produced (s)he can query the *transformation* and *mapping* provenance of one author and its duplicate. For instance:

```
SELECT *
FROM
    (SELECT TRANSSQL * FROM target.author) AS prov
WHERE
    firstName = 'Jim' and lastName = 'Grey'
```

This query reveals that the duplicate with affiliation was created by the left input of the union in $\mathbf{T_1}$ and the one without an affiliation was produced by the right input of the union. The computation of *transformation* provenance in *TRAMP* also propagates mapping annotations and, therefore, the user is aware that the left input corresponds to mapping $\mathbf{M_1}$ and the right one to $\mathbf{M_2}$. Checking the definitions of these mappings the user will realize that $\mathbf{M_1}$ and $\mathbf{M_2}$ are correct, and, thus, the error must have been caused by the transformation. If the mapping system supports it the user can request a different implementing transformation or he may have to correct the error by manually changing the implementing transformation. At this point the target *Author* relation contains the desired data. We do not follow the debugging process from here on.

[3] If the target instance is large, the user can check that this assumption is true by adding a WHERE clause condition to the query.

7.4 Discussion

We now compare the functionality *TRAMP* provides for schema mapping debugging with the functionality of approaches proposed in the literature and afterwards summarize the findings of this chapter.

7.4.1 Comparison with Schema Mapping Debugging Systems

The need to support users in understanding the results of schema mapping has been addressed before, although not in as comprehensive and inclusive manner as in TRAMP. The *Clio data-viewer* [YMHF01] system helps a user in understanding a schema mapping by presenting the result of applying a transformation for a schema mapping on small example source instances (examples chosen by the tool). This approach was later extended for XML data and mappings in *Muse* [ACMT08].

The rationale is that by examining the result data produced by a mapping, a user can gain an understanding of the inner workings of a mapping and alternative mappings that may be considered by a design tool. This rationale is true to some degree, but for complex mappings having just the source and target instance is not enough to understand the mapping. More precise information about the relationships between input and output data is needed (*data* provenance). Also, if the mapping is correct, but the transformation used to generate examples is incorrect (or the data is dirty), these approaches do not help users in understanding the source of these errors.

Recent approaches to schema debugging facilitate provenance information in the debugging process. *SPIDER* [CT06] uses provenance in defining *routes* computed for a subset of a target instance. Each route is a possible way of producing the tuples of interest by sequentially applying mappings to tuples (*route-steps*) in the source instance (and the tuples generated by previous mapping applications in the route). A route combines *data* with *mapping* provenance. *SPIDER* does not provide any querying facilities over the routes and lacks support for debugging incorrect transformations. Routes consider only the logical specification of a mapping. Thus, they have the advantage of being independent of the concrete implementation of a mapping in the form of a transformation query or program. On the other hand this independence can also be problematic if an error is caused by an incorrect transformation. Furthermore, no query facilities for routes are provided, they can only be explored using the visual interface of *SPIDER*. A sequential representation of routes is used that is not fully determined by the mappings. This representation may mislead a user into expecting a meaning from the chosen ordering.

MXQL [VMM05] generates provenance information eagerly during the execution of a transformation. The generated target instance is enriched with transformations that store *mapping* provenance information and provenance that relates source to target schema elements (*schema-schema* provenance). *MXQL* is, together with *TRAMP*, the only approach that provides full query language support for provenance and mapping information (SQL is used by both systems). In contrast to *TRAMP*, *MXQL* supports neither *meta-querying* nor *transformation* provenance.

Transformation provenance has some similarities with *How* provenance [GKT07a] and *Why-not* provenance [CJ09]. *How* provenance has the disadvantage that, unlike *transformation* provenance, it does not record any information about which operators of a query contributed to a result. The major differences of Why-not provenance to our approach is that we compute and present *transformation* provenance for each witness list, whereas *Why-not* provenance is computed for an input pattern and there is only one output for a certain input pattern. Furthermore, no method to query provenance is provided. Finally, to compute *Why-not* provenance, the *data* provenance of several parts of the query have to be computed. Our implementation has the advantage that, though we also define *transformation* provenance based on *data* provenance, we never have to pay the price of instantiating this information in the computation of *transformation* provenance.

Different forms of *Mapping* provenance have been implemented in *Orchestra* [GKT07a] and *MXQL* [VMM05]. In *Orchestra*, a so-called *collaborative data sharing system*, schema mappings are used to propagate updates between peers with different schemas. *How* provenance is used to store the origin of data in a peer instance and to determine if data should be rejected because it came from an untrusted peer. *Mapping* provenance is modeled by adding functions to the *how*-provenance semi-ring model. Each function represents a mapping and is applied to a sub-expression in the provenance. For instance, $m_1(t_1t_2) + m_2(t_3)$ means that the result tuple t was produced by mappings m_1 and m_2 where m_1 used input tuples

t_1 and t_2 in conjunction and m_2 used only tuple t_3 to produce t. The *mapping* provenance provided by *MXQL* represents static (meaning instance independent) information [VMM05]. MXQL uses annotations to associate information about both mappings and source schemas with target data. However, using static information has the disadvantage that it is not possible to determine exactly which mappings produce a tuple. In contrast to *MXQL*, *TRAMP* uses run-time information and, thus, can identify the mapping that generated an output tuple.

7.4.2 Summary

In this chapter we demonstrated that the *Perm* system (with the *TRAMP* extensions to cope with schema mappings) can be used to realize a holistic approach for schema mapping debugging. *TRAMP* extends *Perm* with *mapping* provenance and provides efficient and powerful query and *meta-query* support for this information. It is known that provenance can be important in understanding and debugging integrated information, but we show how only the full combination of different kinds of provenance and query functionality can answer many questions that are important in understanding and disambiguating the sources of the myriad of errors than can occur when integrating data. Furthermore, the new functionality can be implemented efficiently as extensions of the existing rewrite algorithms in *Perm*. For instance, the implementation of *mapping* provenance required only small extensions to the *transformation* provenance computation of *Perm*. We evaluated the viability of our holistic approach by means of an example debugging process.

In summary, we demonstrated that only minor extensions to the generic *Perm* system are necessary to adapt the system to a specific application domain with its own, unique requirements. Furthermore, the novel features of *Perm*, especially the relational representation and computation of different types of provenance, enabled a more holistic approach towards schema mapping debugging than provided by existing approaches like *SPIDER* [CT06], *MXQL* [VMM05], *Muse* [ACMT08], and *Clio data-viewer* [YMHF01].

Chapter 8

Conclusions and Outlook

8.1 Thesis Summary

In this thesis, we have presented the *Perm* approach for integrating provenance support into relational database systems. In contrast to other *PMS* our system is ”purely relational” in the sense that provenance information is represented alongside with normal data as standard relations and is computed by executing a rewritten version of the SQL query for which provenance should be computed.

With *PI-CS* we defined a new contribution semantics type that adapts *Lineage-CS* to a purely relational representation and solves the problems of this *CS* type regarding queries that use negation or sublinks. We presented algebraic rewrite rules that transform a query q into a query q^+ computing the original result and the provenance of q and have proven their correctness. Furthermore, based on *PI-CS*, we developed several *C-CS* types that can be computed by applying filtering steps to the *PI-CS* provenance of a query and presented a contribution semantics for *transformation* provenance that is also based on *PI-CS*.

To integrate the presented provenance functionality into SQL, we developed the *SQL-PLE* language extension that enriches SQL with constructs for triggering and controlling provenance computation. With the implementation of *SQL-PLE* as an extension to *PostgreSQL* we have presented a full-fledged relational *PMS* that supports provenance computation and querying for almost all SQL query constructs supported by *PostgreSQL* (except for non-deterministic functions and recursion). Optimizations such as de-correlation and un-nesting for sublinks and avoidance of rewrites for sub-queries with constant provenance are integrated in the system to improve the performance of provenance computations. *Perm* generates provenance *lazily* on demand, supports external provenance and partial provenance computation.

The performance measurements conducted with *Perm* indicate that our approach is feasible. The inherent cost of computing the provenance for sublink queries on large databases is evident in our results. Large run-time differences between the rewrite strategies for sublinks demonstrate the benefits of integrating un-nesting and de-correlation techniques into the rewrites.

The application of *Perm* to a specific use case - schema mapping debugging - illustrated the general applicability of the system. By implementing marginal extensions to *Perm* a novel holistic approach to mapping debugging was realized which supports debugging functionality not found in other provenance enabled mapping systems.

In summary, to the best of our knowledge we are the first to provide a fully implemented relational *PMS* build upon a sound theoretical foundation that is capable of computing the provenance of complex SQL queries.

8.2 Future Work

Based on the results presented in this thesis, we propose several avenues of future work. Being fully implemented, *Perm* provides a platform for exploring topics such as provenance-aware physical operators, compressed provenance storage, provenance of data manipulations, and support for specialized provenance needs like in uncertain databases.

8.2.1 Provenance Aware Physical Operators

While implementing and developing the query rewrites for *Perm*, patterns of typical computations performed in rewritten queries were identified. The efficiency of provenance computation could be increased by developing new physical operators that directly propagate provenance information. This could lead to significant improvements, because the algebraic rewrites often require the recreation of intermediate results, leading to expensive join or sublink operations. For example, a modified aggregation operator, that in addition to computing aggregation functions, also passes on all tuples from a group could be used in provenance computation to propagate provenance information directly through an aggregation without the need for joins between provenance information and original aggregation.

In a typical query execution engine, attribute values are copied a lot during the processing of a query. This is a feasible approach for normal operations. For provenance computation this approach is suboptimal, because during execution provenance information is either passed on unmodified by an operator or dropped; only the association between provenance and original intermediate results changes. This fact could be utilized by modifying the physical operators of a DBMS to allow the reuse of provenance information by different operators in a query plan.

8.2.2 Data Models and Transformation Types

It would be interesting to generalize the results of this thesis to other data models and transformation languages. For example, is it possible to represent the provenance of XML data in XML and compute such representation using XQuery or XSLT? If applied to transformations that are executable programs written in a procedural language like Java, rewriting a transformation is program instrumentation. This would be along the lines of [ZZZP07b] where *Valgrind* [htt09a] is used to instrument a binary to propagate provenance information.

8.2.3 Index Structures and Compressed Representations

Index structures for provenance have been proposed (e.g., [KW09, HA08]), but their performance has not been evaluated inside a *PMS*. *Perm* could serve as a platform for such evaluations. The same applies for compression of provenance information. Compression schemes for provenance data have been studied without integrating them into a *PMS*, which would enable the estimation of their impact on query performance in such a system. Furthermore, compression could be integrated with provenance aware physical operators with similar benefits as, e.g., the integration of string compression in a query engine. Meaningful contribution semantics like *PI-CS*, *Lineage-CS* and *Why-CS* may produce huge amounts of provenance information. One approach to reduce the size of the generated provenance could be to replace parts of the provenance with a symbolic representation. For instance, instead of representing parts of the provenance as a collection of tuples it could be represented as a query that returns the tuples that belong to the provenance (this is similar to how [SV07a] represents annotations). In addition to making provenance more intelligible for the user, such representations open up new opportunities for optimizing provenance computation.

Appendix A

SQL-PLE Grammar

This appendix presents a grammar for *SQL-PLE*. We use the grammatical representation that is used in the *PostgreSQL* manual. *SQL-PLE* extensions are highlighted in red. The new language constructs added by *SQL-PLE* are presented below.

Construct	Description
PROVENANCE	Marks a query for provenance computation
ON CONTRIBUTION (cs_type)	Instructs *Perm* to use a certain *CS* type
BASERELATION	Handle a from-clause item as if it were a base relation
PROVENANCE (attr_list)	Handle attributes from attr_list as provenance attributes
TRANSPROV/TRANSSQL/TRANSXML	Mark a query for transformation provenance computation
EXPLAIN SQLTEXT	Return the (rewritten) SQL text of an query
EXPLAIN GRAPH	Return an algebra tree for an query (as a dot-language script)
MAPPROV	Compute the *mapping* provenance and represent it as sets of mappings.
THIS.child*path*	A shortcut for generating the XML representation of the subquery accessed by *path*.
XSLT.*f* (xml_param)	Apply XSLT function *f* to XML document `xml_param`.
cxpath (path_expr , input)	Evaluate X-Path expression `path_expr` over `input`. Returns true if the evaluation of the XPath expression returns at least one result.
ANNOT(annotation)	Annotates the *FROM* clause item it is appended to with `annotation`.

A.1 SELECT

Synopsis

```
select_statement :=
SELECT [ provenance_clause ] distinct_clause select_item [, ...]
[ FROM from_item [ from_provenance ] [, ...] ]
[ WHERE condition ]
[ GROUP BY expression [, ...] ]
[ HAVING condition [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [, ...]
[ LIMIT { count | ALL } ]
[ OFFSET start ]
[ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

where from_item can be one of:

```
from_item :=
[ ONLY ] table_name [ * ] opt_ple [ alias ] [ col_alias ]
| ( select ) opt_ple [ alias ] [ col_def ]
| function_name ( [ argument [, ...] ] ) [ alias ] [ col_alias ]
| function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
| from_item [ NATURAL ] join_type from_item [ ON join_condition | using_expr ]

alias :=
[ AS ] aliasname

col_def :=
[ ( column_alias [, ...] ) ] ]

opt_ple :=
[ annot_expr ] [ from_provenance ]
```

Description

SQL-PLE extends the *PostgreSQL* select statement with triggers for provenance computation, clauses for controlling the rewrite process for *FROM* clause items, and annotation functionality.

Provenance Clause

Triggers different types of provenance computation.

```
provenance_clause :=
PROVENANCE [ ON CONTRIBUTION ( cs_type ) ]
| MAPPROV
| TRANSPROV
| TRANSXML
| TRANSSQL
```

ANNOT Expression

Used to annotate a *FROM* clause item.

```
annot_expr :=
ANNOT (annotation [, ...])
```

From Clause Provenance Expression

Controls the rewrite process for *FROM* clause items.

```
from_provenance :=
BASERELATION
| PROVENANCE ( column [, ...] )
```

A.2 EXPLAIN

Synopsis

```
EXPLAIN [SQLTEXT | GRAPH] [ ANALYZE ] [ VERBOSE ] statement
```

Description

Perm adds two extensions to the EXPLAIN command of *PostgreSQL*: SQLTEXT and GRAPH.

Parameters

SQLTEXT
Return the SQL text of the input query, but apply *SQL-PLE* rewrites beforehand.
GRAPH
Return an algebra tree for the input query.

Bibliography

[AB03] Michael O. Akinde and Michael H. Böhlen. Efficient Computation of Subqueries in Complex OLAP. *ICDE '03: Proceedings of the 19th International Conference on Data Engineering*, pages 163–174, 2003.

[ABJF06] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance Collection Support in the Kepler Scientific Workflow System. In *IPAW '06: International Provenance and Annotation Workshop*, pages 118–132, 2006.

[ABML09a] Manish Kumar Anand, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Efficient Provenance Storage over Nested Data Collections. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 958–969, 2009.

[ABML09b] Manish Kumar Anand, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Exploring Scientific Workflow Provenance Using Hybrid Queries over Nested Data and Lineage Graphs. In *SSDBM '09: Proceedings of the 21th International Conference on Scientific and Statistical Database Management*, pages 237–254, 2009.

[ABS$^+$06a] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. An Introduction to ULDBs and the Trio System. *IEEE Data Engineering Bulletin*, 29(1):5–16, 2006.

[ABS$^+$06b] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1151–1154, 2006.

[ACMT08] Bogdan Alexe, Laura Chiticariu, Renée J. Miller, and Wang-Chiew Tan. Muse: Mapping Understanding and Design by Example. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering*, pages 10–19, 2008.

[AF00] Paul Avery and Ian T. Foster. The GriPhyN Project: Towards Petascale Virtual Data Grids. *The 2000 NSF Information and Technology Research Program*, 2000.

[AHV95] Serge Abiteboul, Rick Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[Ale05] Bogdan Alexe. An Alternative Storage Scheme for the DBNotes Annotation Management System for Relational Databases. Technical report, University of California, Santa Cruz, 2005.

[ATV08] Bogdan Alexe, Wang-Chiew Tan, and Yannis Velegrakis. STBenchmark: Towards a Benchmark for Mapping Systems. *PVLDB: Proceedings of the VLDB Endowment archive*, 1(1):230–244, 2008.

[AW07] Parag Agrawal and Jennifer Widom. Confidence-Aware Joins in Large Uncertain Databases. Technical report, Stanford University, 2007.

[AW09] Parag Agrawal and Jennifer Widom. Confidence-Aware Joins in Large Uncertain Databases. In *ICDE '09: Proceedings of the 25th International Conference on Data Engineering*, 2009.

[AZV$^+$02] James Annis, Yong Zhao, Jens Voeckler, Michael Wilde, Steve Kent, and Ian T. Foster. Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey. In *Supercomputing '02: Proceedings of the Conference on Supercomputing*, pages 1–14, 2002.

[BA97] Amos Bairoch and Rolf Apweiler. The SWISS-PROT Protein Sequence Data Bank and its Supplement TrEMBL. *Nucleic Acids Research*, 25(1):31, 1997.

[BB88] Howard S. Bilofsky and Christian Burks. The GenBank Genetic Sequence Data Bank. *Nucleic Acids Research*, 16(5):1861, 1988.

[BBC$^+$09] Michael Blow, Vinayak Borkar, Michael Carey, Christopher Hillery, Alexander Kotopoulis, Dimitry Lychagin, Radu Preotiuc-Pietro, Panagiotis Reveliotis, Joshua Spiegel, and Till Westmann. Updates in the AquaLogic Data Services Platform. In *ICDE '09: Proceedings of the 25th International Conference on Data Engineering*, pages 1431–1442, 2009.

[BCC06] Peter Buneman, Adriane Chapman, and James Cheney. Provenance Management in Curated Databases. Technical report, The University of Edinburgh, 2006.

[BCCV06] Peter Buneman, Adriane Chapman, James Cheney, and Stijn Vansummeren. A Provenance Model for Manually Curated Data. In *IPAW '06: International Provenance and Annotation Workshop*, pages 162–170, 2006.

[BCTV04] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 900–911, 2004.

[BCTV05] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An Annotation Management System for Relational Databases. *The VLDB Journal*, 14(4):373–396, 2005.

[BCV08] Peter Buneman, James Cheney, and Stijn Vansummeren. On the Expressiveness of Implicit Provenance in Query and Update Languages. *ACM Transactions on Database Systems (TODS)*, 33(4):1–47, 2008.

[BGH$^+$06] Uri Braun, S. Garfinkel, David A. Holland, Kiran-Kumar Muniswamy-Reddy, and Margo Seltzer. Issues in Automatic Provenance Collection. *Lecture Notes in Computer Science*, 4145:171, 2006.

[Bin08] Carsten Binnig. *Generating Meaningful Test Databases*. PhD thesis, University of Heidelberg, 2008.

[BKL06] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse Query Processing. Technical report, ETH Zürich, 2006.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and Where: A Characterization of Data Provenance. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 316–330, 2001.

[BKT02] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On Propagation of Deletions and Annotations through Views. In *PODS '02: Proceedings of the 21th Symposium on Principles of Database Systems*, pages 150–158, 2002.

[BKTT04] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. Archiving Scientific Data. *ACM Transactions on Database Systems (TODS)*, 29(1):2–42, 2004.

[BM95] Lars Baekgaard and Leo Mark. Incremental Computation of Nested Relational Query Expressions. *ACM Transactions on Database Systems (TODS)*, 20(2):111–148, 1995.

[BML$^+$06] Shawn Bowers, Timothy McPhillips, Bertram Ludäscher, Sarah Cohen, and Susan Davidson. A Model for User-Oriented Data Provenance in Pipelined Scientific Workflows. In *IPAW '06: International Provenance and Annotation Workshop*, pages 133–147, 2006.

[BML08] Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. Provenance in Collection-oriented Scientific Workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008.

[Bos02] Rajendra Kumar Bose. A Conceptual Framework for Composing and Managing Scientific Data Lineage. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 15–19, 2002.

[Bos04] Rajendra Kumar Bose. *Composing and Conveying Lineage Metadata for Environmental Science Research Computing*. PhD thesis, University of California, Santa Barbara, 2004.

[BSHW06] Omar Benjelloun, Anish Das Sarma, Alon Y. Halevy, and Jennifer Widom. ULDBs: Databases with Uncertainty and Lineage. In *VLDB '06: Proceedings of the 32th International Conference on Very Large Data Bases*, 2006.

[CAA07] James Cheney, Amal Ahmed, and Umut Acar. Provenance as Dependency Analysis. In *DBPL '07: Proceedings of the 11th International Symposium on Database Programming Languages*, pages 138–152, 2007.

[CAA08] James Cheney, Umut Acar, and Amal Ahmed. Provenance Traces. Technical report, University of Edinburgh, 2008.

[CB07] Bin Cao and Antonio Badia. SQL Query Optimization through Nested Relational Algebra. *ACM Transactions on Database Systems (TODS)*, 32(3):18, 2007.

[CCT09] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[CFS$^+$06] Steven Callahan, Juliana Freire, Emanuele Santos, Carlos Eduardo Scheidegger, Claudio T. Silva, and Huy Vo. VisTrails: Visualization meets Data Management. In *SIGMOD '06: Proceedings of the 32th SIGMOD International Conference on Management of Data (demonstration)*, pages 745–747, 2006.

[CFV$^+$08] Ben Clifford, Ian T. Foster, Jens S. Voeckler, Michael Wilde, and Yong Zhao. Tracking Provenance in a Virtual Data Grid. *Concurrency and Computation: Practice and Experience*, 20(5):565–575, 2008.

[Cha98] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS '98: Proceedings of the 17th Symposium on Principles of Database Systems*, pages 34–43, 1998.

[Che00] James Cheney. A Metaprogramming Approach to Data Provenance. Technical report, University of Edinburgh, 2000.

[Che07] James Cheney. Program Slicing and Data Provenance. *IEEE Data Engineering Bulletin*, 30(4):22–28, 2007.

[CJ08] Adriane Chapman and H. V. Jagadish. Provenance and the Price of Identity. In *IPAW '08: International Provenance and Annotation Workshop*, pages 106–119, 2008.

[CJ09] Adriane Chapman and H. V. Jagadish. Why Not? In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 523–534, 2009.

[CJR08] Adriane Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient Provenance Storage. In *SIGMOD '08: Proceedings of the 35th SIGMOD International Conference on Management of Data*, pages 993–1006, 2008.

[CPS+09] Bin Cao, Beth Plale, Girish H. Subramanian, Ed Robertson, and Yogesh L. Simmhan. Provenance Information Model of Karma Version 3. In *SERVICES I '09: Proceedings of the Congress on Services*, pages 348–351, 2009.

[CT06] Laura Chiticariu and Wang-Chiew Tan. Debugging Schema Mappings with Routes. In *VLDB '06: Proceedings of the 32th International Conference on Very Large Data Bases*, pages 79–90, 2006.

[CTV05] Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. DBNotes: a Post-it System for Relational Databases based on Provenance. In *SIGMOD '05: Proceedings of the 31th SIGMOD International Conference on Management of Data*, pages 942–944, 2005.

[CTX+05] Liming Chen, Victor Tan, Fenglian Xu, Alexis Biller, Paul Groth, Simon Miles, John Ibbotson, Michael Luck, and Luc Moreau. A Proof of Concept: Provenance in a Service Oriented Architecture. In *AHM '05: Proceedings of the UK OST e-Science All Hands Meeting*, pages 274–281, 2005.

[Cui02] Yingwei Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2002.

[CW00a] Yingwei Cui and Jennifer Widom. Lineage Tracing in a Data Warehousing System. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering (demonstration)*, pages 683–684, 2000.

[CW00b] Yingwei Cui and Jennifer Widom. Practical Lineage Tracing in Data Warehouses. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, pages 367–378, 2000.

[CW00c] Yingwei Cui and Jennifer Widom. Storing Auxiliary Data for Efficient Maintenance and Lineage Tracing of Complex Views. In *DMDW '00: Proceedings of the 2th International Workshop on Design and Management of Data Warehouses*, 2000.

[CW01a] Yingwei Cui and Jennifer Widom. Lineage Tracing for General Data Warehouse Transformations. In *VLDB '07: Proceedings of the 33th International Conference on Very Large Data Bases*, pages 471–480, 2001.

[CW01b] Yingwei Cui and Jennifer Widom. Run-time Translation of View Tuple Deletions using Data Lineage. Technical report, Stanford University, 2001.

[CWW00] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[Day83] Umeshwar Dayal. Processing Queries with Quantifiers a Horticultural Approach. In *PODS '83: Proceedings of the 2th Symposium on Principles of Database Systems*, pages 125–136, 1983.

[DCBE+07] Susan B. Davidson, Sarah Cohen-Boulakia, Anat Eyal, Bertram Ludäscher, Timothy McPhillips, Shawn Bowers, and Juliana Freire. Provenance in Scientific Workflow Systems. *IEEE Data Engineering Bulletin*, 32(4):44–50, 2007.

[DSANW08] Anish Das Sarma, Parag Agrawal, Shubha Nabar, and Jennifer Widom. Towards Special-Purpose Indexes and Statistics for Uncertain Data. Technical report, Stanford University, 2008.

[DSTW08] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Data Modifications and Versioning in Trio. Technical report, Stanford University, 2008.

[EAE+09] Mohamed Y. Eltabakh, Walid G. Aref, Ahmed K. Elmagarmid, Mourad Ouzzani, and Yasin N. Silva. Supporting Annotations on Relations. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 379–390, 2009.

[EGLGJ07] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind Joshi. Execution Strategies for SQL Subqueries. In *SIGMOD '07: Proceedings of the 33th SIGMOD International Conference on Management of Data*, pages 993–1004, 2007.

[EKA$^+$08] Tommy Ellkvist, David Koop, Erik W. Anderson, Juliana Freire, and Claudio T. Silva. Using Provenance to Support Real-Time Collaborative Design of Workflows. In *IPAW '08: International Provenance and Annotation Workshop*, pages 266–279, 2008.

[EOA07] Mohamed Y. Eltabakh, Mourad Ouzzani, and Walid G. Aref. BDBMS - A Database Management System for Biological Data. In *CIDR '07: Proceedings of the 3th Conference on Innovative Data Systems Research*, 2007.

[EOA$^+$08] Mohamed Y. Eltabakh, Mourad Ouzzani, Walid G. Aref, Ahmed K. Elmagarmid, Yasin Laura-Silva, Muhammad U. Arshad, David Salt, and Ivan Baxter. Managing Biological Data using BDBMS. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering (demonstration)*, pages 1600–1603, 2008.

[FB01] James Frew and Rajendra Kumar Bose. Earth System Science Workbench: A Data Management Infrastructure for Earth Science Products. In *SSDBM '01: Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, pages 180–189, 2001.

[FGT08] J. Nathan Foster, Todd J. Green, and Val Tannen. Annotated XML: Queries and Provenance. In *PODS '08: Proceedings of the 27th Symposium on Principles of Database Systems*, 2008.

[FKMP05] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[FKSS08] Juliana Freire, David Koop, Emanuele Santos, and Claudio T. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science and Engineering*, 10(3):11–21, 2008.

[FMS08] James Frew, Dominic Metzger, and Peter Slaughter. Automatic Capture and Reconstruction of Computational Provenance. *Concurrency and Computation: Practice and Experience*, 20(5):485, 2008.

[Fos03] Ian T. Foster. The Virtual Data Grid: A new Model and Architecture for Data-Intensive Collaboration. In *SSDBM '03: Proceedings of the 15th International Conference on Scientific and Statistical Database Management*, pages 11–11, 2003.

[FS08a] James Frew and Peter Slaughter. Automatic Run-Time Provenance Capture for Scientific Dataset Generation. In *AGU Fall Meeting Abstracts*, page 1039, 2008.

[FS08b] James Frew and Peter Slaughter. ES3: A Demonstration of Transparent Provenance for Scientific Computation. In *IPAW '08: International Provenance and Annotation Workshop*, pages 200–207, 2008.

[FSP07] James Frew, Peter Slaughter, and Thomas H. Painter. ES3: Automatic Capture and Reconstruction of Science Product Lineage and Metadata. In *AGU Fall Meeting Abstracts*, page 4, 2007.

[FVWZ02] Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *SSDBM '02: Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, pages 37–46, 2002.

[GA09a] Boris Glavic and Gustavo Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE '09: Proceedings of the 25th International Conference on Data Engineering*, pages 174–185, 2009.

[GA09b] Boris Glavic and Gustavo Alonso. Provenance for Nested Subqueries. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 982–993, 2009.

[GA09c] Boris Glavic and Gustavo Alonso. The Perm Provenance Management System in Action. In *SIGMOD '09: Proceedings of the 35th SIGMOD International Conference on Management of Data (demonstration)*, pages 1055–1058, 2009.

[GD07] Boris Glavic and Klaus R. Dittrich. Data Provenance: A Categorization of Existing Approaches. In *BTW '07: Proceedings of Datenbanksysteme in Buisness, Technologie und Web*, pages 227–241, 2007.

[GGS$^+$03] Mark Greenwood, Carole Goble, Robert D. Stevens, Jun Zhao, Matthew Addis, Darren Marvin, Luc Moreau, and Tom Oinn. Provenance of e-Science Experiments - Experience from Bioinformatics. In *AHM '03: Proceedings of the UK OST e-Science All Hands Meeting*, pages 223–226, 2003.

[GJM$^+$06a] Paul Groth, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau. An Architecture for Provenance Systems. Technical report, University of Southampton, 2006.

[GJM$^+$06b] Paul Groth, Sheng Jiang, Simon Miles, Steve Munroe, Victor Tan, Sofia Tsasakou, and Luc Moreau. An Architecture for Provenance Systems - Executive Summary. Technical report, University of Southampton, 2006.

[GKIT07] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update Exchange with Mappings and Provenance. In *VLDB '07: Proceedings of the 33th International Conference on Very Large Data Bases*, pages 675–686, 2007.

[GKM05] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. MONDRIAN: Annotating and Querying Databases through Colors and Blocks. Technical report, University of Edinburgh, 2005.

[GKM06] Floris Geerts, Anastasios Kementsietsidis, and Diego Milano. iMONDRIAN: A Visual Tool To Annotate and Query Scientific Databases. In *EDBT '06: Proceedings of the 9th International Conference on Extending Database Technology (demonstration)*, pages 1168–1171, 2006.

[GKT07a] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance Semirings. In *PODS '07: Proceedings of the 26th Symposium on Principles of Database Systems*, pages 31–40, 2007.

[GKT$^+$07b] Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Olivier Biton, Zachary G. Ives, and Val Tannen. ORCHESTRA: Facilitating Collaborative Data Sharing. In *SIGMOD '07: Proceedings of the 33th SIGMOD International Conference on Management of Data*, 2007.

[GLM04a] Paul Groth, Michael Luck, and Luc Moreau. A Protocol for Recording Provenance in Service-Oriented Grids. In *OPODIS '04: Proceedings of the 8th International Conference on Principles of Distributed Systems*, pages 124–139, 2004.

[GLM04b] Paul Groth, Michael Luck, and Luc Moreau. Formalising a Protocol for Recording Provenance in Grids. In *AHM '04: Proceedings of the UK OST e-Science All Hands Meeting*, pages 147–154, 2004.

[GMF$^+$05] Paul Groth, Simon Miles, Weijian Fang, Sylvia C. Wong, Klaus-Peter Zauner, and Luc Moreau. Recording and Using Provenance in a Protein Compressibility Experiment. In *HPDC '05: Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, 2005.

[GMM05] Paul Groth, Simon Miles, and Luc Moreau. PReServ: Provenance Recording for Services. In *AHM '05: Proceedings of the UK OST e-Science All Hands Meeting*, pages 282–289, 2005.

[GMTM05] Paul Groth, Simon Miles, Victor Tan, and Luc Moreau. Architecture for Provenance Systems. Technical report, University of Southampton, 2005.

[GN00] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *Software Practice and Experience*, 30(11):1203–1233, 2000.

[Gre09] Todd J. Green. Containment of Conjunctive Queries on Annotated Relations. In *ICDT '09: Proceedings of the 16th International Conference on Database Theory*, 2009.

[Gro04] Dennis P. Groth. Information Provenance and the Knowledge Rediscovery Problem. In *IV '04: Proceedings of the 8th International Conference on Information Visualisation*, pages 345–351, 2004.

[GVdB07] Floris Geerts and Jan Van den Bussche. Relational Completeness of Query Languages for Annotated Databases. *Lecture Notes in Computer Science*, 4797:127, 2007.

[HA08] Thomas Heinis and Gustavo Alonso. Efficient Lineage Tracking for Scientific Workflows. In *SIGMOD '08: Proceedings of the 34th SIGMOD International Conference on Management of Data*, pages 1007–1018, 2008.

[HCDN08] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the Provenance of Non-answers to Queries over Extracted Data. *PVLDB: Proceedings of the VLDB Endowment archive*, 1(1):736–747, 2008.

[HE97] Kathleen Hornsby and Max J. Egenhofer. Qualitative Representation of Change. In *COSIT '97: Proceedings of the International Conference on Spatial Information Theory: A Theoretical Basis for GIS*, pages 15–33, 1997.

[HHT09] Melanie Herschel, Mauricio A. Hernández, and Wang-Chiew Tan. Artemis: A System for Analyzing Missing Answers. In *VLDB '09: Proceedings of the 35th International Conference on Very Large Data Bases (demonstration)*, pages 1550–1553, 2009.

[HLB$^+$08] Bill Howe, Peter Lawson, Renee Bellinger, Erik W. Anderson, Emanuele Santos, Juliana Freire, Carlos Eduardo Scheidegger, Antonio Baptista, and Claudio T. Silva. End-to-End eScience: Integrating Workflow, Query, Visualization, and Provenance at an Ocean Observatory. In *eScience '08: Proceedings of the 4th IEEE International Conference on eScience*, pages 127–134, 2008.

[HQGW93] Nabil I. Hachem, Ke Qiu, Michael A. Gennert, and Matthew O. Ward. Managing Derived Data in the Gaea Scientific DBMS. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 1–12, 1993.

[htt09a] http://valgrind.org. Valgrind, 2009.

[htt09b] http://www.ittvis.com/. IDL, 2009.

[IGK$^+$08] Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira. The ORCHESTRA Collaborative Data Sharing System. *SIGMOD Record*, 37(2), 2008.

[IKKC05] Zachary G. Ives, Nitin Khandelwal, Aneesh Kapur, and Murat Cakir. ORCHESTRA: Rapid, Collaborative Sharing of Dynamic Data. In *CIDR '05: Proceedings of the 2th Conference on Innovative Data Systems Research*, 2005.

[ILJ84] Tomasz Imieliński and Witold Lipski Jr. Incomplete Information in Relational Databases. *Journal of the ACM (JACM)*, 31(4):761–791, 1984.

[IW09] Robert Ikeda and Jennifer Widom. Data Lineage: A Survey. Technical report, Stanford University, 2009.

[KDG$^+$08] Jihie Kim, Ewa Deelman, Yolanda Gil, Gaurang Mehta, and Varun Ratnakar. Provenance Trails in the Wings/Pegasus System. *Concurrency and Computation: Practice and Experience*, 20(5):587–597, 2008.

[Kim82] Won Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.

[KP07] Daisuke Kihara and Sunil Prabhakar. Tracing Lineage in Multi-Version Scientific Databases. Technical report, Purdue University, 2007.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, Inc., 1988.

[KR98] Bogdan Korel and Jurgen Rilling. Dynamic Program Slicing Methods. *Information and Software Technology*, 40(11-12):647–659, 1998.

[KTL$^+$03] Ananth Krishna, Victor Tan, Richard Lawley, Simon Miles, and Luc Moreau. The myGrid Notification Service. In *AHM '03: Proceedings of the UK OST e-Science All Hands Meeting*, pages 475–482, 2003.

[KVVS$^+$06] Tamás Kifor, László Zsolt Varga, Javier Vázquez-Salceda, Sergio Álvarez, Steven Willmott, Simon Miles, and Luc Moreau. Provenance in Agent-Mediated Healthcare Systems. *IEEE Intelligent Systems*, 21(6):38–46, 2006.

[KW09] Anastasios Kementsietsidis and Min Wang. Provenance Query Evaluation: What's so Special about it? In *CIKM '09: Proceeding of the 18th Conference on Information and Knowledge Management*, pages 681–690, 2009.

[KY94] Bogdan Korel and Satish Yalamanchili. Forward Computation of Dynamic Program Slices. In *ISSTA '94: Proceedings of the International Symposium on Software Testing and Analysis*, pages 66–79, 1994.

[LAB$^+$06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew B. Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[Lan89] David P. Lanter. *Techniques and Method of Spatial Database Lineage Tracing*. PhD thesis, University of South Carolina, 1989.

[Lan93] David P. Lanter. Method and Means for Lineage Tracing of a Spatial Information Processing and Database System, March 9 1993. US Patent 5,193,185.

[Len02] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *PODS '02: Proceedings of the 21th Symposium on Principles of Database Systems*, pages 233–246, 2002.

[LNH$^+$05] Jonathan Ledlie, Chaki Ng, David A. Holland, Kiran-Kumar Muniswamy-Reddy, Uri Braun, and Margo Seltzer. Provenance-Aware Sensor Data Storage. In *ICDE '03: Workshop Proceedings of the 21th International Conference on Data Engineering*, page 1189, 2005.

[LPA$^+$08] Bertram Ludäscher, Norbert Podhorszki, Ilkay Altintas, Shawn Bowers, and Timothy McPhillips. From Computation Models to Models of Provenance: the RWS Approach. *Concurrency and Computation: Practice and Experience*, 20(5):507–518, 2008.

[LSV02] Jens Lechtenbörger, Hua Shu, and Gottfried Vossen. Aggregate Queries over Conditional Tables. *Journal of Intelligent Information Systems*, 19(3):343–362, 2002.

[LZW$^+$97] Wilburt J. Labio, Yue Zhuge, Janet L. Wiener, Himanshu Gupta, Hector García-Molina, and Jennifer Widom. The WHIPS Prototype for Data Warehouse Creation and Maintenance. In *SIGMOD '97: Proceedings of the 21th SIGMOD International Conference on Management of Data*, pages 557–559, 1997.

[Mar01] Arunprasad P. Marathe. Tracing Lineage of Array Data. *Journal of Intelligent Information Systems*, 17(2-3):193–214, 2001.

[MBZL08] Timothy McPhillips, Shawn Bowers, Daniel Zinn, and Bertram Ludäscher. Scientific Workflow Design for Mere Mortals. *Future Generation Computer Systems*, 25(5):541–551, 2008.

[MCG$^+$05] Luc Moreau, Liming Chen, Paul Groth, John Ibbotson, Michael Luck, Simon Miles, Omer Rana, Victor Tan, Willmott, and Fenglian Xu. Logical Architecture Strawman for Provenance Systems. Technical report, University of Southampton, 2005.

[MFH$^+$01] Renée J. Miller, Daniel Fisla, Mary Huang, David Kymlicka, Fei Ku, and Vivian Lee. The Amalgam Schema and Data Integration Test Suite, 2001. www.cs.toronto.edu/ miller/amalgam.

[MFM$^+$07] Luc Moreau, Juliana Freire, Jim Myers, Joe Futrelle, and Patrick Paulson. The Open Provenance Model, 2007.

[MGBM07] Simon Miles, Paul T. Groth, Miguel Branco, and Luc Moreau. The Requirements of Using Provenance in e-Science Experiments. *Journal of Grid Computing*, 5(1):1–25, 2007.

[MGM$^+$08] Luc Moreau, Paul Groth, Simon Miles, Javier Vazquez-Salceda, John Ibbotson, Sheng Jiang, Steve Munroe, Omer Rana, Andreas Schreiber, Victor Tan, and Laszlo Varga. The Provenance of Electronic Data. *Communications of the ACM*, 51(4):52–58, 2008.

[MHH00] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema Mapping as Query Discovery. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 77–88, 2000.

[MHH$^+$09] Renée J. Miller, Laura M. Haas, Mauricio A. Hernández, Ronald Fagin, Lucian Popa, and Yannis Velegrakis. Clio: Schema Mapping Creation and Data Exange. *Conceptual Modeling: Foundations and Applications*, page 236, 2009.

[MI06] Luc Moreau and John Ibbotson. Standardisation of Provenance Systems in Service Oriented Architectures. Technical report, University of Southampton, 2006.

[Mom01] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Boston, MA: Addison-Wesley, 2001.

[MPL$^+$06] James D. Myers, Carmen M. Pancerella, Carina S. Lansing, Karen L. Schuchardt, Brett T. Didier, Naveen Ashish, and Carole A. Goble. Multi-scale Science: Supporting Emerging Practice with Semantically Derived Provenance. In *ISWC Workshop '03: Semantic Web Technologies for Searching and Retrieving Scientific Data*, 2006.

[MS97] Arunprasad P. Marathe and Kenneth Salem. A Language for Manipulating Arrays. In *VLDB '97: Proceedings of the 23th International Conference on Very Large Data Bases*, pages 46–55, 1997.

[MTdK$^+$07] Michi Mutsuzaki, Martin Theobald, Ander de Keijzer, J. Widom, Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Raghotham Murthy, and Tomoe Sugihara. Trio-One: Layering Uncertainty and Lineage on a Conventional DBMS. In *CIDR '07: Proceedings of the 3th Conference on Innovative Data Systems Research*, pages 269–274, 2007.

[Mur92] M. Muralikrishna. Improved Unnesting Algorithms for Join Aggregate SQL Queries. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 91–102, 1992.

[Net09] Nicholas Nethercote. Valgrind. http://valgrind.org, 2009.

[RB01] Erhard Rahm and Philip A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.

[RBC$^+$08] Alessandro Raffio, Daniele Braga, Stefano Ceri, Paolo Papotti, and Mauricio A. Hernández. Clip: a Visual Language for Explicit Schema Mappings. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering*, pages 30–39, 2008.

[Ree00] George Reese. *Database Programming with JDBC and Java*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2000.

[SBHW06] Anish Das Sarma, Omar Benjelloun, Alon Y. Halevy, and Jennifer Widom. Working Models for Uncertain Data. In *ICDE '06: Proceedings of the 22th International Conference on Data Engineering*, page 7, 2006.

[SC05] Can Sar and Pei Cao. Lineage file system. Technical report, Stanford University, 2005.

[SCL99] Laurent Spéry, Christophe Claramunt, and Thérèse Libourel. A Lineage MetaData Model for the Temporal Management of a Cadastre Application. In *DEXA '99: Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, page 466, 1999.

[SCL01] Laurent Spéry, Christophe Claramunt, and Thérèse Libourel. A Spatio-Temporal Model for the Manipulation of Lineage Metadata. *Geoinformatica*, 5(1):51–70, 2001.

[SCN$^+$93] Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson, and Jiang Wu. Tioga: Providing Data Management Support for Scientific Visualization Applications. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 25–38, 1993.

[SFC07] Claudio T. Silva, Juliana Freire, and Steven Callahan. Provenance for Visualizations: Reproducibility and Beyond. *Computing in Science and Engineering*, 9(5):82–89, 2007.

[SKS$^+$07] Carlos Eduardo Scheidegger, David Koop, Emanuele Santos, Huy Vo, Steven Callahan, Juliana Freire, and Claudio T. Silva. Tackling the Provenance Challenge one Layer at a Time. *Concurrency and Computation: Practice and Experience*, 2007.

[SM03] Martin Szomszor and Luc Moreau. Recording and Reasoning over Data Provenance in Web and Grid Services. In *ODBASE'03: International Conference on Ontologies, Databases and Applications of SEmantics*, volume 2888 of *Lecture Notes in Computer Science*, pages 603–620, Catania, Sicily, Italy, November 2003.

[SMRH$^+$05] Margo Seltzer, Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Jonathan Ledlie. Provenance-Aware Storage Systems. Technical report, Harvard University, 2005.

[SPG05a] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[SPG05b] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A Survey of Data Provenance Techniques. Technical report, Indiana University, Bloomington IN 47405, 2005.

[SPG08a] YL Simmhan, B. Plale, and D. Gannon. Karma2: Provenance Management for Data-Driven Workflows. *International Journal of Web Services Research*, 5(2):1–22, 2008.

[SPG08b] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. Query Capabilities of the Karma Provenance Framework. *Concurrency and Computation: Practice and Experience*, 20(5):441–451, 2008.

[SRG03] Robert D. Stevens, Alan J. Robinson, and Carole A. Goble. myGrid: Personalised Bioinformatics on the Information Grid. *Bioinformatics*, 19(90001):302–304, 2003.

[SUW07] Anish Das Sarma, Jeffrey Ullman, and Jennifer Widom. Schema Design for Uncertain Databases. Technical report, Stanford University, 2007.

[SV07a] Divesh Srivastava and Yannis Velegrakis. Intensional Associations between Data and Metadata. In *SIGMOD '07: Proceedings of the 33th SIGMOD International Conference on Management of Data*, pages 401–412. ACM Press New York, NY, USA, 2007.

[SV07b] Divesh Srivastava and Yannis Velegrakis. MMS: Using Queries As Data Values for Metadata Management. In *ICDE '07: Proceedings of the 23th International Conference on Data Engineering*, pages 1481–1482, 2007.

[SV07c] Divesh Srivastava and Yannis Velegrakis. Using queries to associate metadata with data. In *ICDE '07: Proceedings of the 23th International Conference on Data Engineering*, pages 1451–1453, 2007.

[SVK$^+$08] Carlos Eduardo Scheidegger, Huy Vo, David Koop, Juliana Freire, and Claudio T. Silva. Querying and Re-using Workflows with VisTrails. In *SIGMOD '08: Proceedings of the 34th SIGMOD International Conference on Management of Data*, pages 1251–1254. ACM, 2008.

[Tan03] Wang-Chiew Tan. Containment of Relational Queries with Annotation Propagation. *DBPL '03: Proceedings of the International Workshop on Database and Programming Languages*, 2003.

[Tan04] Wang-Chiew Tan. Research Problems in Data Provenance. *IEEE Data Engineering Bulletin*, 27(4):42–52, 2004.

[Tan07] Wang-Chiew Tan. Provenance in Databases: Past, Current, and Future. *IEEE Data Engineering Bulletin*, 30(4):3–12, 2007.

[TGM$^+$06] Victor Tan, Paul Groth, Simon Miles, Sheng Jiang, Steve Munroe, Sofia Tsasakou, and Luc Moreau. Security Issues in a SOA-based Provenance System. In *IPAW '06: International Provenance and Annotation Workshop*, pages 203–211, 2006.

[Tra09] Transaction Processing Council. TPC-H Benchmark Specification, 2009.

[Tro05] Vadim Tropashko. Nested Intervals Tree Encoding in SQL. *ACM SIGMOD Record*, 34(2):52, 2005.

[VC07] Stijn Vansummeren and James Cheney. Recording Provenance for SQL Queries and Updates. *IEEE Data Engineering Bulletin*, 30(4):29–37, 2007.

[VdBVV05] Jan Van den Bussche, Stijn Vansummeren, and Gottfried Vossen. Towards Practical Meta-Querying. *Information Systems*, 30(4):317–332, 2005.

[VMM05] Yannis Velegrakis, Renée J. Miller, and John Mylopoulos. Representing and Querying Data Transformations. In *ICDE '05: Proceedings of the 21th International Conference on Data Engineering*, pages 81–92, 2005.

[Wid05] Jennifer Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR '05: Proceedings of the 2th Conference on Innovative Data Systems Research*, pages 262–276, 2005.

[Wid08] Jennifer Widom. Trio: A System for Managing Data, Uncertainty, and Lineage. *Managing and Mining Uncertain Data*, 2008.

[WM07] Jennifer Widom and R. Murthy. Making aggregation work in uncertain and probabilistic databases. In *Proceedings of the Workshop on Management of Uncertain Data*, pages 76–90, September 2007.

[WMF$^+$05a] Sylvia C. Wong, Simon Miles, Weijian Fang, Paul Groth, and Luc Moreau. Provenance-based Validation of E-Science Experiments. In *ISWC '05: Proceedings of 4th Internation Semantic Web Conference*, pages 801–815, 2005.

[WMF$^+$05b] Sylvia C. Wong, Simon Miles, Weijian Fang, Paul Groth, and Luc Moreau. Validation of E-Science Experiments using a Provenance-based Approach. In *AHM '05: Proceedings of the UK OST e-Science All Hands Meeting*, pages 290–296, 2005.

[WS97] Allison Woodruff and Michael Stonebraker. Supporting Fine-grained Data Lineage in a Database Visualization Environment. In *ICDE '97: Proceedings of the 30th International Conference on Data Engineering*, pages 91–102, Washington, DC, USA, 1997. IEEE Computer Society.

[WSU07] Jennifer Widom, Anish Das Sarma, and Jeffrey D. Ullman. Functional Dependencies for Uncertain Relations. Technical report, Stanford University, 2007.

[WTS08] Jennifer Widom, Martin Theobald, and Anish Das Sarma. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *ICDE '08: Proceedings of the 24th International Conference on Data Engineering*, April 2008.

[YMHF01] Ling-Ling Yan, Renée J. Miller, Laura M. Haas, and Ronald Fagin. Data-driven Understanding and Refinement of Schema Mappings. In *SIGMOD '01: Proceedings of the 27th SIGMOD International Conference on Management of Data*, pages 485–496. ACM New York, NY, USA, 2001.

[ZDF$^+$05] Yong Zhao, James E. Dobson, Ian T. Foster, Luc Moreau, and Michael Wilde. A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data. *ACM Sigmod Record*, 34(3):37–43, 2005.

[ZGG$^+$03a] J. Zhao, C. Goble, M. Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-science. *Proceedings of the Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data*, pages 92–106, 2003.

[ZGG$^+$03b] Jun Zhao, Carole Goble, Mark Greenwood, Chris Wroe, and Robert Stevens. Annotating, Linking and Browsing Provenance Logs for e-science, October 2003.

[ZGSB04] Jun Zhao, Carole A. Goble, Robert D. Stevens, and Sean Bechhofer. Semantically Linking and Browsing Provenance Logs for E-science. *First International Conference on Semantics of a Networked World*, pages 157–174, 2004.

[ZHC$^+$07] Yong Zhao, Mihael Hategan, Ben Clifford, Ian T. Foster, Gregor Von Laszewski, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *IEEE Workshop on Scientific Workflows*. Citeseer, 2007.

[ZWF$^+$04] Yong Zhao, Michael Wilde, Ian T. Foster, Jens Vöckler, Thomas Jordan, Elizabeth Quigg, and James Dobson. Grid Middleware Services for Virtual Data Discovery, Composition, and Integration. In *Middleware '04: Proceedings of the 2nd Workshop on Middleware for Grid Computing*, pages 57–62, New York, NY, USA, 2004. ACM Press.

[ZWF06] Yong Zhao, Michael Wilde, and Ian T. Foster. Applying the Virtual Data Provenance Model. In *IPAW '06: International Provenance and Annotation Workshop*, pages 148–161, 2006.

[ZWG$^+$04] Jun Zhao, Chris Wroe, Carole A. Goble, Robert Stevens, Dennis Quan, and R. Mark Greenwood. Using Semantic Web Technologies for Representing e-science Provenance. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *ISWC '04: Proceedings of the 3th International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 92–106, Hiroshima, Japan, 2004. Springer.

[ZZZP07a] Mingwu Zhang, Xiangyu Zhang, Xiang Zhang, and Sunil Prabhakar. Cost Effective Forward Tracing Data Lineage. Technical report, Purdue University, 2007.

[ZZZP07b] Mingwu Zhang, Xiangyu Zhang, Xiang Zhang, and Sunil Prabhakar. Tracing Lineage beyond Relational Operators. In *VLDB '07: Proceedings of the 33th International Conference on Very Large Data Bases*, pages 1116–1127. VLDB Endowment, 2007.

Curriculum Vitae Boris Glavic

2005-2010	Doctoral student at the Database Technology Research Group (DBTG) University of Zurich
2005	Diploma in Computer Science from RWTH Aachen (Germany)
1999-2005	Diploma student in Computer Science at RWTH Aachen (Germany)