Provenance Tools

Rocco Flach, Maximilian Lamster, Chris Röhrs, Nic Scharlau, Tanja Auge

University of Rostock, 18051 Rostock, Germany,
{rocco.flach, maximilian.lamster, chris.roehrs, nic.scharlau, tanja.auge}@uni-rostock.de

ABSTRACT

*The importance of provenance has arose for all kinds of sciences over the recent years. During research on data provenance, several tools have been developed to use provenance in a practical way. We chose seven of those tools and exhaustingly tested five of them: Trio, ORCHESTRA, Perm, GProM, and ProvSQL. In this article, we first introduce the basics of data provenance, especially **where**-, **why**-, and **how**-provenance. After that, we present the results of our tool tests.*

TYPE OF PAPER AND KEYWORDS

Short communication: *Data Provenance, Data Lineage, Databases, Tool Comparison, GProM, ORCHESTRA, Perm, ProvSQL, Trio*

Contents

1 Introduction

Provenance in fact isn't a commonly used term but its popularity is increasing drastically. It's a concept used almost everywhere in scientifical research, even though it is not called provenance but instead "quality", "reconstruction", "trust", or "understandability".

The first appearance of provenance for most of us have been the experiments we made in lessons of physics or chemistry. To clarify, provenance is not the experiment itself but the documentation behind it: what was used, what were the observations, how long

did the process take and what happened at which time? A chemical experiment for example always needs a documentation of tools, chemicals, a drawing of the experimental setup, the exact documentation of the execution, observations, measurements and sources if used. The last section of each documentation is the interpretation and evaluation of the experiment which increases in quality, understandability, reconstruction and trust if detailed documentation or provenance is provided. It is no surprise you wanna keep the provenance of your research as detailed as possible in order to not only justify your results but to ensure the use cases of provenance.

Provenance in data science can be explained similarly with the addition that each provenance item uses additional storage space. If you have a massive calculation going on thats consulting a database in a dimension you are not able to deliver in any attachment and is way to big to store externally for the necessary amount of time you want to make sure that you only store or provide those database entries that really went into the calculation or were crucial for the output.

Data provenance is a term used for provenance that is collected or produced in information systems with a granularity of single data items. This makes it easy not only to save but to track down crucial data items that have been used. Even though provenance may not be a popular term itself one may ask in what extent the concept got adapted into software that is available to the public. In the following we want to present a variety of tools implementing data provenance in different ways including one special tool that uses the concept of provenance to increase the performance of an algorithm rather than increasing quality or else.

1.1 Selection criteria

Our goal is to summarize the development of data provenance tools from the last decades. The selection is based on the survey "A systematic review of provenance systems" by Pérez et al. which already compares a range of tools [PRS18]. The survey is focusing on the literature. In addition we want to test the selected tools ourselves. There are two main criteria. On the one hand the program should work data oriented. On the other hand the granularity has to be at least on tuple level. Thus we can ensure the ability to assign the tools to the ***how***-, ***why***- and ***where***-provenance or ***lineage***, respectively. It is important for us that we can understand how the tools are working. Additionally our readers should be able to test and use the tools themselves. Hence they should be free and open source.

Based on the results of [PRS18] we have chosen seven tools in total. The chosen tools are Trio, ORCHESTRA, Perm, GProM, ProvSQL, Tioga/Tioga-2 and ProvC&B. We picked Trio because it is based on data lineage, and it is the oldest tool we tested. ORCHESTRA is the first tool which introduced the ***how***-provenance. Perm was chosen because it uses annotation propagation and query rewriting. It is also the predecessor of GProM. GProM is interesting because of three reasons. First it realizes an approach to generate provenance game graphs. Second it's only a middleware and not a distinguished tool and third it is the successor of Perm. ProvSQL is the latest tool we decided for testing, and it uses semirings to calculate the provenance. The order in which these tools are discussed is based on their release date shown in Figure 1.

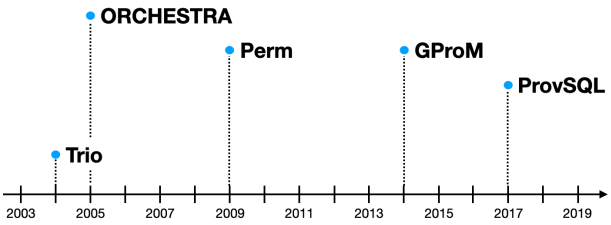

Figure 1: Overview about every tool we decided to test in release order

1.2 Test database and queries

Database We introduce a sample database of a fictional university. It contains five tables: STUDENTS (Table 12) contains eight records of students with first and last name, student id and study course. COURSES (Table 13) attaches a number to each course. LECTURERS (Table 14) assigns each course one or more lecturers. PARTICIPANTS (Table 15) tells us which student takes which course. GRADES (Table 16) shows which grades a student has got. Each row refers to a course and a semester. The full database is located in the appendix on page 23.

Queries We define four queries using the given tables to test the tools. Query 1 returns a list of all study courses which the students are enrolled in. This query is used to test duplicate elimination. Query 2 lists the students that listen to the lecture 005. Joins are used very often in relational database systems, so we want to test them too. Query 3 is a variation of the previous query. Details about this query will be discussed below. Query 4 gives a overview of all grades that the university has given. Aggregation and groupings are used widely in data analyses. These four queries cover most operations of the relational algebra.

```
SELECT s.firstname, l.fullname
```

```
FROM students s, participants p,
     lecturers l
WHERE s.student_id = p.student_id
AND p.course_nr = l.course_nr;
```

Command 1: Query 3

Let us take Query 3 as example. It is written in SQL in Command 1. The corresponding result is listed in Table 1. We can take which lecturers teach which students from the result table. Query 2 and this query have one important difference: A projection is evaluated only for columns from STUDENTS and LECTURERS. We want to know whether the tool does output the tuple from PARTICIPANTS which is used for both joins or not. Then we know whether ***where***-provenance was applied or not. On the right side of the table there are the answers to the ***how***-, ***why***- and ***where***-provenance plus ***data lineage*** displayed. The ***how***-provenance shows the tuple IDs connected through a multiplication with the $\otimes$-symbol. This represents a join operation. The ***why***-provenance shows the same tuple IDs like the previous example but we cannot see how they were processed. All tuples that we need to reproduce are collected in a set called *witness*. There is only one witness consisting of the three tuples. Therefore the outer set called *witness base* has one element. The ***where***-provenance only shows the relation names of the projected attributes. Since participants is not requested in the select clause this relation is omitted. The ***data lineage*** is the predecessor of the data provenance. It lists us all tuples that were involved in generating a specific row of the result. In section 2 we explain in detail how the different provenance levels are working. All queries in SQL and the resulting tables can be reviewed in the appendix, beginning on page 25.

1.3 Structure of this paper

This paper is divided into four further sections. Section 2 deals with the basics of provenance. This includes a general definition of provenance in subsection 2.1, possible types of provenance in 2.2 and their respecting applications in 2.3. Subsection 2.4 then goes further into detail regarding *data provenance*. More detailed descriptions of the questions ***where*** (see 2.4.1), ***why*** (2.4.3), ***how*** (2.4.4) and ***why-not*** (2.4.5) are provided.

Section 3 then presents tools that are able to process provenance information. Each individual tool is first introduced, their ability to handle provenance and associated literature is discussed, observations that have been made for each tool are presented in detail and a summary then concludes each tool test. Tools covered in this paper are **GProM** (see 3.4), **ORCHESTRA** (3.2), **Perm** (3.3), **ProvC&B** (3.7), **ProvSQL** (3.5), **Tioga** (3.6) and **Trio** (3.1).

In section 4 we want to compare the tools with consideration of benchmark queries introduced in appendix A.2, the ability to handle ***how***-, ***why***-, and ***where***-provenance, the current developement status and their license.

Section 4 gives a final summary of the results and findings that have emerged from this paper.

2 Provenance

The whole section about provenance is explained based on [HDB17]. Additional sources used are cited in their corresponding text passages.

2.1 Definition

> "Provenance generally refers to any information that describes the production process of an end product, which can be anything from a piece of data to a physical object." [HDB17].

This definition describes the concept of provenance really well but far too general. In order to be able to process collected provenance information and thus cover specific use cases, some restrictions have to be made. These restrictions result in different types of provenance that are presented in the following subsection 2.2.

2.2 Types of provenance

This subsection covers the fundamentals of the four different types of provenance. These types are called *Provenance meta-data*, *Information system provenance*, *Workflow provenance*, and ***Data provenance***. Figure 2 provides a hierarchy to additionally clarify that *Provenance meta-data* is the most general type and that each overlying type is the result of further restriction of the underlying one. ***Data provenance*** therefore is the most specific type of provenance covered in this article.

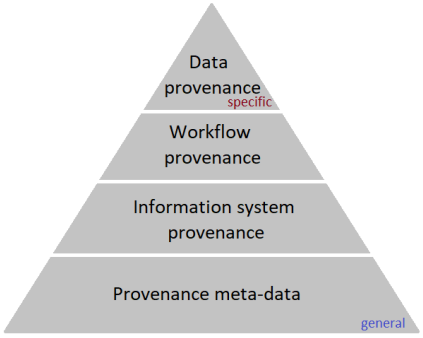

Figure 2: Provenance hierarchy based on [HDB17]

	firstname	fullname	*how*	*why*	*where*	*lineage*
R_1	Donald	Lecturer A	$S_1 \otimes P_1 \otimes L_{1.2}$	$\left\{\{S_1, P_1, L_{1.2}\}\right\}$	STUDENTS, LECTURERS	$S_1, P_1, L_{1.2}$
R_2	Donald	Professor A	$S_1 \otimes P_1 \otimes L_{1.1}$	$\left\{\{S_1, P_1, L_{1.1}\}\right\}$	STUDENTS, LECTURERS	$S_1, P_1, L_{1.1}$
R_3	Sarah	Lecturer A	$S_2 \otimes P_2 \otimes L_{1.2}$	$\left\{\{S_2, P_2, L_{1.2}\}\right\}$	STUDENTS, LECTURERS	$S_2, P_2, L_{1.2}$
	...	...	...	...	...	...

Table 1: Result of Query 3 (first 3 rows) with Provenance

Provenance meta-data Being the most general type, *Provenance meta-data* includes any type of meta-data referring to a production process. It can be applied to any kind of production process and allows various options for modelling and implementation of respecting solutions for provenance management without any restrictions and, more importantly, without disclosing internals. The main difference between other meta-data and *Provenance meta-data* is the intended application.

Information system provenance Restricting *Provenance meta-data* to production processes that produce digital data inside an information system results in *Information system provenance*. An information system in that context is a system that produces, collects, distributes, or processes information in the form of digital data. In order to limit fitting production processes, we further classify *Information system provenance* as *Provenance meta-data* that is being computed inside information systems via input, output, and parameters, while internal processes stay hidden or unknown.

Workflow provenance Workflow provenance emerges out of further restriction of *Information system provenance* to production processes that can be represented as a workflow. A workflow can be understood as a graph that consists of multiple nodes and edges. A node stands for a module that is associated with input, output, and parameters and an edge represents a data or control flow between previous mentioned nodes. This kind of representation allows a simple way to visualize different granularities of provenance information and also simplifies instrumentation.

Data provenance Instead of onward restriction of the production processes, we now decrease the granularity of provenance information to single data items to reach ***Data provenance***. In comparison to other provenance types, ***Data provenance*** allows to track each and every single data item in a whole production process. It is often applied on structured data models – e.g. declarative query languages – to exploit such structure or semantics. This also allows easier instrumentation for all kinds of possible use cases and applications.

Main applications of provenance are covered in section 2.3 and ***Data provenance*** will be further specified in section 2.4, also including important questions like ***where***, ***why***, ***how*** and ***why-not***.

2.3 Applications of provenance

This section shall provide some basic information about the three main use cases of provenance in order to motivate the usefulness of provenance management. Each use case will be further divided into subtypes by classifying provenance authors and provenance consumers as this will help to cover a wider range of applications.

Understandibility The first use case of provenance is *understandability*. Provenance information is used to present results and necessary application steps to a specific audience. Storing information about why a step was performed is optional here.

Collaboration describes understandibility within a group of experts who work together on a project. This group of experts therefore is provenance creator as well as provenance consumer. In this scenario it is necessary to record performed actions and make them available to other users in a comprehensible and understandable way. Collaboration can either be performed synchronous or asynchronous. Synchronous collaboration describes the immediate visibility of changes in a system, including the provision of the resulting provenance information. Asynchronous collaboration describes the collection of such information without immediate change and provision. Asynchronous solutions often update at certain time intervals and include a list of changes that accumulated since the last update was executed.

The choice of how to present provenance information can make a significant contribution to understandibility

if the provenance consumers belong to a wider crowd and not to a group of experts. Presentation therefore describes the visualization of provenance information to support understandibility. The presentation and exploration of such designed provenance information can take place in corresponding applications and also interactively to increase understandibility even more.

Provenance information can often be written or consumed by a large number of different individuals or is even implicitly available in corresponding data sets. Such provenance information is referred to as attribution. A simple example is authorship: provenance information on relevant persons can be obtained just by examining the sender of messages for example. This type of provenance solely focusses on one single aspect and is therefore easier to handle.

Reproducibility The second use case of provenance information is *reproducibility*. The goal is to enable any group of provenance consumers to obtain the same end product using only the same materials and methods. This use case describes the most natural approach on provenance management.

Recall is a subtype of *reproducibility* that represents the recording of every step taken. Recall usually has the same provenance creator and consumer and also stores unnecessary steps in order to be able to remember everything.

Replication on the other hand describes the recording of exclusively necessary steps and therefore answers not only how but also why a step was performed. This type of *reproducibility* is mostly used to enable provenance consumption to a broader audience where unnecessary steps are not of interest.

Quality Provenance information also allows to increase the quality of an end product or to improve the efficiency of a production process itself. Although this type of use case is as important as the others mentioned it is out of scope in this paper and also not part of the implementation in tested tools.

Detour: FAIR principles Another motivation for provenance management are specific guidelines also known as the *FAIR principles* that were introduced in [Wil+16] that aim to improve **F**indability, **A**ccessibility, **I**nteroperability, and the **R**euse of digital assets in order to allow computer systems to better deal with the huge amount of data that is created nowadays. To meet the conditions of the *FAIR principles* and more specifically the **R**euse it is necessary to provide detailed provenance information that is associated with the (meta)data.

Query type	Question answered
where	Where does the data come from?
why	Why did we achieve this result?
how	How was the result calculated?
why-not	Why is a specific element missing?

Table 2: Provenance-Queries

2.4 Data provenance

Data provenance "captures the way in which data is used, combined, and manipulated by [a data-intensive system]" [DFG18]. While this definition is more general, *Buneman et al.* specify data provenance as the process of collecting information about the origins of data, and also mention its movements between databases in that context [BKT00]. As seen by comparing the definitions, data provenance is a wide ranging term – so is provenance itself – and will be covered in this section. In general, the goal of data provenance is to answer four questions that can be seen in Table 2. Hence, there are four main types of data provenance: ***where***-provenance, ***why***-provenance, ***how***-provenance as well as ***why-not***-provenance. We also need to take a look at ***data lineage*** which can be seen as the predecessor of ***why***-provenance.

2.4.1 *Where*-provenance

We start by taking a look at ***where***-provenance. Given a query Q on a database instance I that leads to an output relation $Q(I) = \{t_1, t_2, \ldots, t_n\}$ consisting of tuples t_i ($i \geq 1$). Intuitively, ***where***-provenance has to determine where the information in t_i were copied from, and therefore describes the relation between input and output locations [CCT09]. There are many different ways to describe the origin of such data. *Buneman et al.* make use of location annotations: Given a relation R consisting of tuples t_i which, in itself, consist of attributes A_j. The location of such an attribute can be denoted as location(R, t, A) [BKT01] [CCT09]. When asked where a value v of a tuple t occuring in $Q(I)$ is coming from, we want to know the location(R, t, v) and therefore have to determine R. This can be done by using annotations during the computation of a query result.

For example, take a look at Command 1. FIRSTNAME is an attribute of STUDENTS whereas FULLNAME is an attribute of LECTURERS. If each tuple gets a unique provenance identifier before we perform a query, we can carry these identifiers throughout the whole computation process of that query and eventually derive the locations from the result. If we follow the definition of *Buneman et al.*, we need to annotate the name of the relation, the identifier of the tuple, and the name of the attribute. Therefore, if we keep looking at our

example query, the locations of the first row of the result would be location(STUDENTS, S_1, FIRSTNAME) as well as location(LECTURERS, $L_{1.2}$, FULLNAME).

However, there are alternative ways to denote and calculate ***where***-provenance. One of them is called **tuple-based *where*-provenance**. Using this method, we only need to track the tuple identifiers, assuming that each identifier is unique. In our example, the value "Donald" was copied from tuple S_1 and the value "Lecturer A" was copied from tuple $L_{1.2}$. Hence, the tuple-based ***where***-provenance is $\{S_1, L_{1.2}\}$.

Another method is called the **relation-based *where*-provenance**. Using this method, we only need to store the relation names of according relations holding the information. Since "Donald" was copied from the relation STUDENTS and "Lecturer A" was copied from the relation LECTURERS, the relation-based ***where***-provenance is $\{$STUDENTS, LECTURERS$\}$, as seen in Table 1. Note that this information can also easily be derived from the selection statement of the query itself.

If we take a look at the result of Command 15, seen in Table 17, we can see that the relation-based ***where***-provenance of the first tuple is $\{$LECTURERS$\}$ because every information given in the result has its origin in this relation; other relations are not necessary to produce the output. Again, the provenance information can easily be derived from the `SELECT` statement of the query. The ***where***-provenance of the first tuple in the result of Command 16 (seen in Table 18), however, is $\{$STUDENTS, PARTICIPANTS$\}$ because part of the information in the result were copied from STUDENTS, some were copied from PARTICIPANTS. Again, this can be derived by taking a closer look at the query itself.

2.4.2 *Data lineage*

Before we continue with ***why***-provenance, we shall take a look at ***data lineage***.

The aim of ***data lineage*** is to track all tuples that "contribute to" a query result [BKT00]. However, a definition of that contribution is not as easy as it seems. A common understanding is that a tuple contributes of a result if its removal from the source database would change the result. Another way to describe ***data lineage*** is referred to as "using a simple proof-theoretic definition" by *Buneman et al.* in [BKT00]. According to them, "an input tuple contributes to an output tuple if it is used in some minimal derivation of that tuple".

If we take a look at Command 15 and its result, we can see that the first tuple of the result exists because of the information in both S_3 and S_7 since the value "Computer Science" appears in both tuples, and both are used to produce the output. Hence, the ***data lineage*** of that tuple is S_3, S_7. However, if we remove either S_3 or S_7 from the input database, the output would still be the same, because you only need one of these two tuples to derive the information from. Therefore, S_3 as well as S_7 are two alternative solutions for the ***data lineage*** of the output tuple. Not only that, these two solutions are also minimal but remain undiscovered. Fortunately, ***why***-provenance covers the problem of minimality by introducing (minimal) witnesses.

2.4.3 *Why*-provenance

The goal behind ***why***-provenance is to provide information about the witnesses to a query on an instance. The predecessor of ***why***-provenance is the data lineage which provides a subset of input records that are needed to produce the output records. However, a single output tuple can be witnessed by multiple witnesses, which data lineage does not cover. Hence, ***why***-provenance has been defined to distinguish between multiple witnesses [BKT01] [CCT09]: Let I be a database instance, Q be a query over I, and t be a tuple in $Q(I)$. An instance $I' \subseteq I$ is a **witness** for t in respect to Q if $t \in Q(I')$. It is notated as $\text{Wit}(Q, I, t) = \{J \subseteq I \mid t \in Q(J)\}$. A set that contains all witnesses for a given tuple t is called the **witness basis** of t. This implies that any subinstance of the database that is relevant for t is a witness for t, including the data lineage as well as the whole database. If the instance I is finite, so is $\text{Wit}(Q, I, t)$, however, this set could be exponentially large due to witnesses that contain tuples not needed to produce t. To conquer this problem, *Cheney et al.* define a minimal witness basis consisting of minimal witnesses. A **minimal witness basis** is a set $\text{MWhy}(Q, I, t) = \{J \in \text{Why}(Q, I, t) \mid J \text{ minimal in } \text{Why}(Q, I, t)\}$ [CCT09].

Please note that our notation of ***why***-provenance differs from the one *Cheney et al.* use. Intuitively, if two input tuples t_1 and t_2 lead to the existence of an output tuple t_3, both t_1 and t_2 are witnesses for t_3; $\{t_1, t_2\}$ is the corresponding witness set. Assuming there are no other tuples that could witness t_3, $\{\{t_1, t_2\}\}$ would be the witness basis of t_3. If t_3 could not be witnessed without either t_1 or t_2, the witness basis would also be minimal.

If we revisit the query in Command 1, the first tuple is the result of a join between STUDENTS, PARTICIPANTS, and LECTURERS. As seen previously, "Donald" was derived from tuple S_1, "Lecturer A" from $L_{1.2}$. Additionally, the tuple also relies on P_1 from the PARTICIPANTS relation because it was involved in joining the three relations. Hence, $\{S_1, P_1, L_{1.2}\}$ is a witness set of the tuple, and hence neither any subset of the witness set nor any other combination of tuples could witness the tuple, $\{\{S_1, P_1, L_{1.2}\}\}$ is also its minimal

witness basis.

Let us also take a look at the query in Command 15 again. The ***why***-provenance of the first output tuple is $\{\{S_3\},\{S_7\}\}$. This means that $\{S_3\}$ and $\{S_7\}$ are two (minimal) witness bases for that tuple. The ***data lineage*** on the other hand, is S_3, S_7 and lacks respect to minimality, as been described.

Also, in some cases it is possible to derive the exact ***where***-provenance from the ***why***-provenance by unpacking and uniting the witness bases into a single set of tuple identifiers. The ***why***-provenance, again, is $\{\{S_3\},\{S_7\}\}$; if we put this information into a single set, we recieve $\{S_3, S_7\}$ which satifies both the ***data lineage*** as well as the ***where***-provenance (assuming we know that S stands for STUDENTS). However, this does not work for, e.g., the query in Command 17: its first output tuple's ***why***-provenance is $\{\{S_1, P_1, L_{1.2}\}\}$ which – unpacked – satisfies the ***data lineage*** $S_1, P_1, L_{1.2}$. If we try to derive the ***where***-provenance from that, we would conclude that it is $\{$STUDENTS, PARTICIPANTS, LECTURERS$\}$, yet not a single value of PARTICIPANTS is part of the output. The actual ***where***-provenance (STUDENTS and LECTURERS) is included, though, but our derived result is not minimal.

2.4.4 *How*-provenance

As we see, ***why***-provenance is superior to ***where***-provenance and covers it in respect to a potential loss of minimality. However, ***why***-provenance only gives information about the necessity of a tuple but not about the amount of its occurrences (or importance) in a result computation. Hence, we need a way to measure how often tuples contribute to a certain result. Fortunately, ***how***-provenance solves this problem by introducing **commutative semirings** for provenance computation as well as **provenance polynomials** for describing results, again using annotations. Therefore, while ***why***-provenance can only describe why a certain result exists, ***how***-provenance can also describe how exactly is was calculated.

Green and *Tannen* define some commutative semirings that are useful for calculating ***how***-provenance [GT17]. For example, $\mathbb{B}(\{\text{true}, \text{false}\}, \vee, \wedge, \bot, \top)$ is a commutative semiring for logical expressions and set semantics in databases. $\mathbb{N} = (\mathbb{N}, \oplus, \otimes, 0, 1)$ is a semiring for bag semantics and counting derivations. Another semiring is $\mathbb{N}[X] = (\mathbb{N}[X], \oplus, \otimes, 0, 1)$ where $\mathbb{N}[X]$ is the set of polynomials in indeterminates from X and coefficients from $\mathbb{N}$. It is used for provenance polynomials and therefore most important for us.

As can be seen, there are two main operations needed for ***how***-provenance: $\oplus$, which is idempotent, and $\otimes$, which is commutative, following the definition of commutative semirings. 0 is used as zero element, 1 is used as identity element. The "$\oplus$" element can be used for duplication elemination, as in projections and unions, and the "$\otimes$" element can be used for joining elements, as in joins, selections, intersections, and the cartesian product.

For example, let $a = \{1, 2, 3\}$ and $b = \{3, 4, 5\}$ be two tuples from different relations sharing a common attribute 3. If these tuples get joined over 3 during a join of their relations, the provenance polynomial of the resulting tuple would be $a \otimes b$. If both a and b would be the result of a projection, the provenance polynomial would be $a \oplus b$.

Consider our query in Command 1. The first tuple is the result of a join between STUDENTS, PARTICIPANTS, and LECTURERS. More precisely, this tuple exists because of a join between tuples S_1, P_1 and $L_{1.2}$, hence the provenance polynomial is $S_1 \otimes P_1 \otimes L_{1.2}$.

Technically, following *Green* and *Tannen*, the polynomial for the whole result would be

$$\begin{aligned} & (S_1 \otimes P_1 \otimes L_{1.2}) \otimes 1 \\ \oplus \quad & (S_1 \otimes P_1 \otimes L_{1.1}) \otimes 1 \\ \oplus \quad & (S_2 \otimes P_2 \otimes L_{1.2}) \otimes 1 \\ \oplus \quad & \ldots \\ \oplus \quad & t_N \otimes 0, \end{aligned}$$

where t_N represents all possible combinations of tuples from STUDENTS, PARTICICPANTS and LECTURERS that do not satisfy – or witness – our query result.

Earlier we saw that ***why***-provenance overlaps ***where***-provenance. The same goes with ***how***-provenance and ***why***-provenance, and therefore also with ***where***-provenance. Looking at Command 15, the first tuple of its result (which again can be seen in Table 17) has the ***how***-provenance $S_3 \oplus S_7$ since either S_3 or S_7 are necessary to produce the output tuple (but not both). Hence, $\{S_3\}$ as well as $\{S_7\}$ are witness sets for that tuple; $\{\{S_3\},\{S_7\}\}$ therefore is the corresponding witness basis. As we saw earlier, it is quite easy to derive the ***data lineage*** and ***where***-provenance from that witness basis. Hence, ***how***-provenance all information that are necessary for ***why***- and ***where***-provenance as well as ***data lineage***: ***where*** $\preceq$ ***why*** $\preceq$ ***how***.

2.4.5 *Why-not*-provenance

In contrast to provenance of existing results - presented in ***why***, ***how*** and ***where***- there is also a provenance for missing results which is summarized below under the term ***why-not*** provenance. Since this type of

provenance does not appear in the further course of this work and especially not in the tested tools, it will only be presented in the form of an overview based on [HDB17] for further supplementation. In general the ***why-not*** provenance provides explanations as to why a data record that was expected in the result does not appear in it. Given a 5-tuple $\{T_R, Q, Q(D), D, C\}$ where $T_R = \{t_{R_1}, t_{R_2}, ...\}$ describes the ("conditional") missing tuples, $Q = \{q_1, q_2, ...\}$ describes a set of queries, D describes the source instance, $Q(D) = \{q_1(D), q_2(D), ...\}$ describes the results and C describes constraints over the 4 previous mentioned parts we diversify three main types of ***why-not*** provenance based on their explanations.

Instance based explanations deliver a set of insert, delete or update operations to existing tuples in D such that $T_R \in Q(D)$. Well known algorithms and systems that use this kind of explanations are *Missing Answers*, *Artemis* or *PGames* (Provenance Games).

Query based explanations deliver query conditions (join, select) included in Q that are responsible for pruning expected tuples defined in T_R. Examples of use are *NedExplain* and *TED*.

Refinement based explanations change the input setting and therefore delivers an alternative for Q named Q' and for T_R named T'_R such that $T'_R \in Q'(D)$. Examples are *TALOS*, *ConQuer* and *FlexIQ*.

2.4.6 Types of answers

As diverse as provenance-queries can be so can be the answers. Table 3 is showing an easily understandable overview of the different types of answers.

Extensional answers originate from questions like ***why***, ***how*** and ***where*** and deliver tuples from the original dataset that are responsible for the result.

Whenever original data needs to be kept private *intensional answers* may be a possible solution. Instead of providing tuples from the original dataset *intensional answers* only provide a description of the dataset and therefore keeping exact values anonymous.

Whenever a ***why-not*** question is answered we either have a *query-* or *modification-based answer*. *Query-based answers* deliver a set of selection predicates that are responsible for an expected tuple to be missing in the desired result. *Modification-based answers* on the other hand can either suggest minimal changes to the considered instance in form of delete-, insert- or update-operations or even suggest minimal changes to the whole input setting in order to deliver a desired result.

Answert type	Result
extensional	Tuples from the original data
intensional	Description of the data
query-based	Selection predicates
modification-based	Suggestion for minimal change

Table 3: Provenance-Answers

3 Tool tests

3.1 Trio

What is Trio? Trio was first presented in 2004, is based on PostgreSQL and implemented in Python 2.x. It is Open Source Software and freely available on the Stanford university website. Trio is based on the *uncertainty liniage databases model* (ULDB-Model) which extends the standard SQL relational model by lineage and uncertainty [Ben+06]. The uncertainty values are stored in an additional column. However, the lineage generated by Trio is retained in a new table. The lineage is handled similar to the ***how***-provenance in Section 2.4.4.

Trio uses its own language, named *TrioQL*. It supports the definition of alternative values, called *uncertain* records. Furthermore we can define trust values for each attribute. These describe the probability for the existance of the attribute. Internally Trio calls this *confidence*. If you want to know the most probable result Trio utilizes horizontal subqueries. Commands can be executed through the browser interface *TrioExplorer* or the command line interface *TrioPlus*. You can run SQL scripts from files too. Trio realizes these concepts with three types of tables: *certain* relations already known from SQL, *uncertain* tables and uncertain tables with *confidences*.

Literature & Provenance The literature says that Trio is able to fulfil ***how***-provenance. However, there is a difference between the ***how***-provenance (see Section 2.4.4) and ***lineage*** known in 2004 (see Section 2.4.2). At this time ***lineage*** connects uncertain records with other alternatives from which they were derived. In Section 2.4.2, ***data lineage*** describes all tuples that are considered to produce a given relation. As we can see, Trio uses the ***data lineage*** definition from 2004. Because of this, we call the old lineage ***trio lineage***. It is more informative then ***why***-provenance but less then ***how***-provenance.

provenance type	result
how	$S_1 \otimes P_1 \otimes L_{1.1}$
trio lineage	$\langle S_1, P_1, L_{1.1} \rangle$
why	$\{\{S_1, P_1, L_{1.1}\}\}$

Table 4: Provenance differences between *how*, *trio lineage* and *why*

Let us take a closer look at Query 3that lists which student is taught by which lecturer. In Figure 3 we see the trio lineage of one of the result tuples. Internally, the ***lineage*** is automatically tracked in an additional table. It is not necessary to explicitly write a command down.

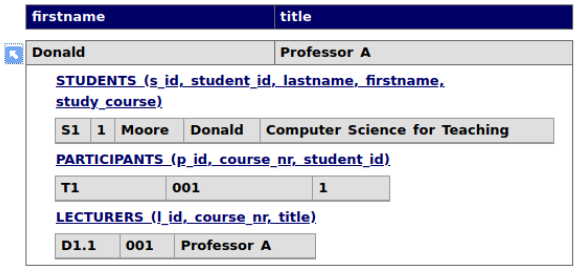

Figure 3: Result of Query 3. Shows the *trio lineage* of the first tuple "Donald" and "Professor A"

This Figure 3 shows the expected result of the Query 3. Opening the lineage tab (blue arrow on the left of the tuple) opens up a box. In the box there are all three tables again with the relevant records for the resulting tuple *("Donald", "Professor A")*. In literature of Trio we find the predicate `lineage`. With this you should be able to get the same information in the tuple but in our tests it produces only an error.

What we observe Trio was tested on Ubuntu 20.04 LTS with Python 2.7 and PostgreSQL 12.2. Trio can handle joins very well and produces the expected results. But we are not allowed to use the keyword `JOIN` in a query.

Aggregations are denoted differently than usual. The functions `avg`, `count`, `max`, `min` and `sum` can be used by adding a "h", "l" or "e", e. g. `havg`, `lcount` or `emax`. If we query a table without uncertainty then the prefixes have no meaning and the normal aggregations are executed. It is different if you use aggregation functions on tables with uncertainty. Then we get results for the highest, the lowest and the expected possibility, respectively. However, we do not get any provenance for those results if you use aggregation functions.

Trio differentiates three types of tables. The first type is the certain table. These tables are displayed in blue. To create one normal table in Trio we use standard SQL.

The second type of tables are green uncertain trio tables. There we are allowed to make tuples with alternative values. Trio displays the column name with an additional star. To create an uncertain trio table we only need to add the keyword `UNCERTAIN` when creating the table. The following Command 2 shows how:

```
CREATE TRIO TABLE lec (
    l_id VARCHAR(10),
    course_nr VARCHAR(3),
    fullname VARCHAR(255),
    UNCERTAIN(fullname)
);
```

Command 2: Definition of relation LEC with an uncertain attribute *fullname* without confidence

As we can see in Figure 4, the column *fullname* in the green table is now marked with a star. To add a tuple into this trio table you add a single questionmark (`?`) at the end of the command 3:

```
INSERT INTO lec
VALUES (
    'L1.1', 001, 'Professor A'
)?;
INSERT INTO lec
VALUES (
    'L2', 002,
    ['Professor B' | 'Professor C']
)?;
```

Command 3: Inserting an record with an uncertain property

If we want to include an alternative value we have to use the vertical bar (`|`) between the alternative values and have to put the values in square brackets (`[]`).

The third type of tables are the uncertain trio tables with confidences. The tuples get a probability at the end. In TrioExplorer these tables are displayed in orange. To create an uncertain trio table with confidences you need to add the keyword `UNCERTAIN` when creating the table and add `WITH CONFIDENCES` at the end like in this command 4:

```
CREATE TRIO TABLE lec2 (
    l_id VARCHAR(10),
    course_nr VARCHAR(3),
    fullname VARCHAR(255),
    UNCERTAIN(fullname)
) WITH CONFIDENCES;
```

Command 4: Definition of relation LEC with an uncertain attribute *fullname* with confidence

As we can see in Figure 4, the column *fullname* in the orange table is now marked with a star. If we want to

include an alternative value we again have to use the vertical bar (|) between the alternative values and have to put the values in square brackets ([]). Attention, here we have to type the alternative values in a different way than for the uncertain trio table without confidences. To add the confidence value we add the probability value after the associated parantheses. We demonstrate this in Command 5.

```
INSERT INTO lec2
VALUES [
    ('L1.1', 001, 'Professor A'):1
];
INSERT INTO lec2
VALUES [
    ('L2', 002, 'Professor B'):0.7 |
    ('L2', 002, 'Professor C'):0.3
];
```

Command 5: Inserting an record with an uncertain property

The usage of foreign keys and unique commands are not supported. Moreover, for the `insert into` command you are only allowed to insert one tuple per command. Sadly `distinct` is not supported too. It only prints the following error message:

```
can only concatenate tuple (not
"list") to tuple
```

Trio has his own user interface, the TrioExplorer, shown in Figure 4. It opens in your browser and gives an overview about every existing table. Also we can query the result there. In the home tab we can view each table individually by selecting the table of our choice and click on *View*. Then we get a legend about the symbolic used in TrioExplorer. At the bottom of the left column we can see your command history. Here we can click on the queries to copy them right into the query box. If we want to store our result in an extra table we can create such a table. It is simply created by adding the usual SQL keywords `CREATE TABLE`.

```
CREATE TABLE result_query_3 AS
  SELECT s.firstname, l.fullname
  FROM students s, participants p, lec l
  WHERE s.student_id = p.student_id
  AND p.course_nr = l.course_nr;
```

Command 6: Creating a result table

The new table created by Command 6 has arrows to the existing source tables. We can even connect uncertain tables with certain tables. This is also shown in Figure 4. It shows among others the new table RESULT_QUERY_3 which was created by Command 6.

If you want to experience Trio by yourself without creating a new database, you can use the samples which

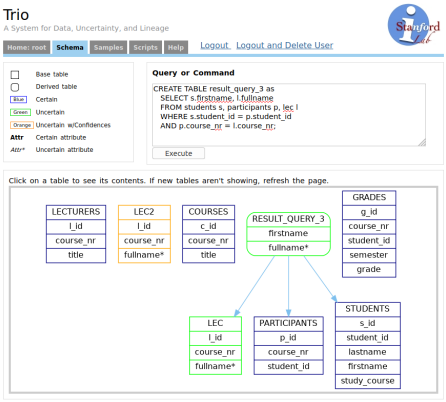

Figure 4: TrioExplorer: A schema overview in Trio with the result table of query 3 with the corresponding query

come with it. There is the sample of the *crime solver* which is ideal for testing Trio yourself. In our virtual machine there is this students example preconfigurated. To execute scripts in Trio we go to the scripts tab and load our file. The help tab gives you a little explanation about all the tabs before.

Conclusion Trio has a very clear user interface with the TrioExplorer. However, the user experience can be clouded by missing implementations and complicated documentations. There are no existing documentations about TrioPlus. We couldn't test the `DELETE` and `UPDATE` command which are described in the TrioQL documentation. It looks like they are not implemented yet. Also Trio cannot work with every data type, foreign keys and unique conditions we observed that aggregations are supported when using horizontal subqueries.

3.2 ORCHESTRA

What is ORCHESTRA? ORCHESTRA was originally developed as a conductor between biological databases. It should transfer data between inhomogenous databases. To reach this goal ORCHESTRA implements a peer-to-peer system. To use this data it provides the implementation of ***how***-provenance on tuple granularity. This tool can generate provenance graphs too. ORCHESTRA was developed in 2005 in Java. The source code is publicly available in the Google Code Archive[1] and is licensed under the Apache-2.0 license.

[1] `https://code.google.com/archive/p/penn-orchestra/`

Literature & Provenance ORCHESTRA, a tool introduced by *Green et al.*, is a tool that uses tuple-generating dependencies for schema mappings [Gre+10]. It is a system for data transfer or data exchange and was designed to exchange data between life sciences databases. Its goal is to track provenance information of exchanged data. To do that, it produces provenance graphs with tuple nodes, mapping nodes, and edges between them. Internally, ORCHESTRA is based on provenance semirings, which makes it the first tool that implemented ***how***-provenance. It also has its own query language ProQL to navigate through provenance graphs.

What we observe There are two versions of ORCHESTRA in the Google Code Archive. We tested version 0.1-SNAPSHOT on Debian 10 with DB2 and Java 8 because we could not make version 0.2-SNAPSHOT work. Thus some of our results may differ from the second version. First we had to resolve some dependency issues by modifying the `pom.xml`. Queries are written in Datalog and stored together with the table definitions plus trust conditions in XML files.

Let us take our running example Query 3. For the reason that mappings are stored as XML files we first have to translate our query into Datalog. In the head element we tell ORCHESTRA which attributes are used in the projection. In the body element the tables are listed that will be joined. We can replace the irrelevant attributes with underscores. The finished configuration snippet is listed in Command 7.

```
<mapping materialized="true" name="M3">
<head>
  <atom>
    STUDENTS.STUDENTSSCHEMA.
      STUDENT_LECTURER(
      FIRSTNAME, FULLNAME)
  </atom>
</head>
<body>
  <atom>
    STUDENTS.STUDENTSSCHEMA.STUDENTS(
      _, STUDENT_ID, _, FIRSTNAME, _)
  </atom>
  <atom>
    STUDENTS.STUDENTSSCHEMA.PARTICIPANTS(
      _, COURSE_NR, STUDENT_ID)
  </atom>
  <atom>
    STUDENTS.STUDENTSSCHEMA.LECTURERS(
      _, COURSE_NR, FULLNAME)
  </atom>
</body>
</mapping>
```

Command 7: Example of a mapping in ORCHESTRA

The result is listed in Table 5. As you can see, the rows have been sorted by FIRSTNAME and then by FULLNAME.

FIRSTNAME	FULLNAME
Donald	Lecturer A
Donald	Professor A
Donald	Professor B
Donald	Professor C
Elisabeth	Lecturer A
Elisabeth	Lecturer B
Elisabeth	Professor A
Elisabeth	Professor B
⋮	⋮

Table 5: Result of Query 3 in ORCHESTRA (first 8 rows)

ORCHESTRA is capable of generate provenance graphs. An example is displayed in Figure 5. First, using the dropdowns on the right we select the peer and the query. Then the result will be printed under the dropdown menus. When selecting a row, the provenance graph is drawn on the main area. The green box on the left represent the result tuple *("Donald", "Professor A")*. It is connected to three yellow rhombuses which stand for the mapping. Now we know we have three possible alternatives to derive this result row. Each mapping is connected to three blue boxes, containing the original tuples from the base tables. These tables are connected with a join. For instance, the top mapping uses `LECTURERS(L1.1, ...)`, `PARTICIPANTS(P1, ...)` and `STUDENTS(S1, ...)`. We observe that the student with the id 3 is used three times, once for each mapping node. Since ORCHESTRA works with Datalog an implicit duplicate elimination is performed. With this information we can calculate the provenance polynomial:

$$(S_1 \otimes P_1 \otimes L_{1.1}) \oplus (S_1 \otimes P_{20} \otimes L_7) \oplus (S_1 \otimes P_{24} \otimes L_9).$$

The two $\oplus$ symbols say that we have three alternatives which were combined via duplicate elimination. Each part is a join of three tuples, denoted by $\otimes$. You can even rearrange the positions of the nodes by drag-and-drop.

When comparing with the official screenshots from the project website the yellow `"+"` attracted our attention. Figure 6 shows another provenance

[2] https://www.cis.upenn.edu/~zives/orchestra/

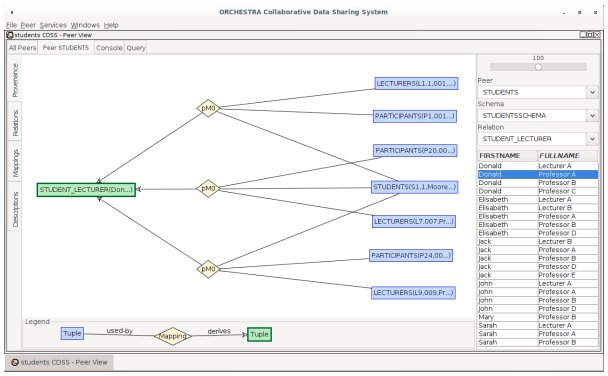

Figure 5: Screenshot of the GUI showing the query result with the corresponding provenance graph

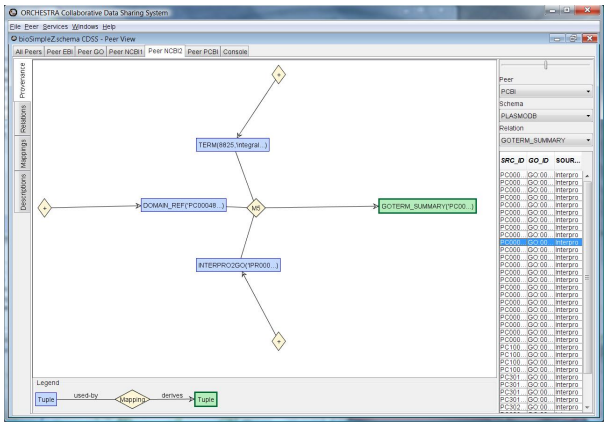

Figure 6: Screenshot from the ORCHESTRA project homepage[2]

Query 1 and Query 2 work as expected and produce similar results. But if we change the selection criteria then the tool produces result row that should not be there. After testing other variations of our queries we assume that selections only work properly if they are applied to key attributes. Query 4 could not be realized because ORCHESTRA does not support aggregations.

Conclusion ORCHESTRA is a tool for exchanging data in peer-to-peer systems. It implements the ***why***-provenance on tuple granularity. Duplicate eliminations are executed by default because ORCHESTRA uses Datalog as query language. Joins are working but aggregations are not. Provenance graphs belong to the functionalities of ORCHESTRA. This tool comes with its own GUI that can display them.

3.3 Perm

What is Perm? Perm stands for *Provenance Extension of the Relational Model* and is a provenance management system which supports provenance queries of different types. It was implemented in 2009 as a modified PostgreSQL server. Perm is based on version 8.3 and is licensed under the terms of the PostgreSQL license. The source code is available at GitHub[3]. The developers decided to extend the source code by a *Perm Module* which comes with a *Query Rewriter*. It takes a query written in *SQL-PLE* (*SQL Provenance Language Extension*) and converts it in standard SQL internally [Gla10]. When the command is executed the provenance information is put in additional columns. Thus Perm can make use of the optimizer of PostgreSQL. To calculate the ***why***-provenance the user adds the `PROVENANCE` keyword straight after the `SELECT` keyword. Query trees are used to solve the ***how***-provenance. The standalone *Perm Browser* client provides a simple-to-use GUI. It can also display query trees.

Literature & Provenance Perm is able to process ***how***-, ***why***-, ***where***- and **transformation** provenance [GA09]. It also introduced the query rewriter and explains how it is embedded in the architecture of Perm. The architecture is discussed later in this paragraph. The query rewriter takes a query q as input and transforms it to another query q^+ that produces the same result as q but generates provenance information for each row in additional columns. The provenance types come with different semantics and granularities to see in the following table.

Provenance Type	Semantics	Granularity
where	C-CS	Tuple
	Where	Attribute Value
why	PI-CS	Tuple
how	Polynomials	Tuple

Table 6: Provenance types supported by Perm with their semantics and granualities, based on [GMA13], p. 7

C-CS stands for Copy-Contribution-Semantics and PI-CS stands for Perm-Influence-Contribution-Semantics. Both semantics are based on ***data lineage***. In Perm however, the ***where***-provenance of attribute values can be found in the additional columns created by the query rewriter. If we look into the result table we can see every detail about the tuples.

The full architecture of Perm is shown in Figure 7. There we also see how this tool is embedded in PostgreSQL. We see that the Perm Module takes the

[3] `https://github.com/IITDBGroup/perm`

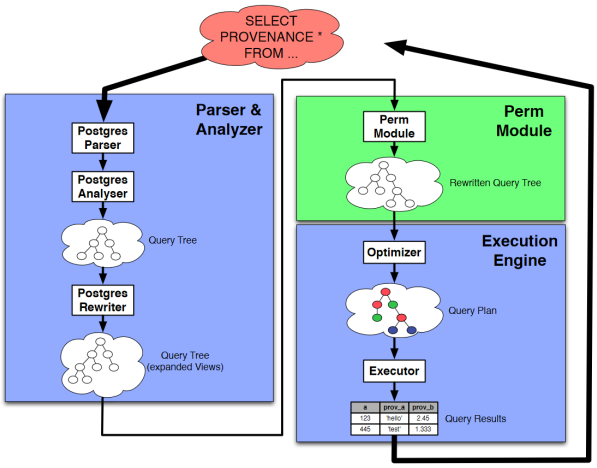

Figure 7: Perm Architecture [Gla10]

output from the PostgreSQL Rewriter and passes its result to the optimizer of PostgreSQL.

What we observe We tested Perm successfully on Ubuntu 18.04 LTS. All four test queries are working as expected. An example of the input is shown in Figure 8. In this paper we want to demonstrate Query 4 because we think it produces the most interesting result.

To instruct Perm to calculate the provenance of a SQL query we add the keyword `PROVENANCE` straight after the `SELECT` keyword like in Command 8. Now Perm will use the aforementioned Query Rewriter to calculate the new query and executes it.

```
SELECT PROVENANCE
    grade, count(*) AS c
FROM grades
WHERE course_nr = '002'
GROUP BY grade
ORDER BY grade;
```

Command 8: Query 4 mit `PROVENANCE`

The output of the query is listed in Table 7. There we can see that the row *(1.3, 2)* is listed twice. This is caused by the two records in GRADES that were grouped together to one row. Because Perm adds new columns PostgreSQL now thinks that these are two different tuples and therefore the duplicate elimination does not find a duplicate anymore. The column names are composed of the string `prov`, the name of the schema, the name of the source table and the name of the attribute, joined with underscores. Here we stored our tables in the default schema `public`. We observed that Perm repeated the underscores in our column names.

grade	c	(1)	(2)	(3)	(4)	(5)
1.0	1	G8	002	4	WS 15/16	1.0
1.3	2	G6	002	2	WS 14/15	1.3
1.3	2	G9	002	5	WS 15/16	1.3
2.0	1	G10	002	6	WS 15/16	2.0
2.3	1	G7	002	3	WS 14/15	2.3
3.3	1	G11	002	7	WS 15/16	3.3
3.7	1	G5	002	1	WS 15/16	3.7

(1) **prov_public_grades_g__id**
(2) **prov_public_grades_course__nr**
(3) **prov_public_grades_student__id**
(4) **prov_public_grades_semester**
(5) **prov_public_grades_grade**

Table 7: Result of Command 8 with Provenance in Perm

Let us show another keyword from the SQL-PLE syntax: `BASERELATION`. It causes a subquery to be treated like a relation from the database when retrieveing provenance data. The keyword is placed between the subquery and the corresponding alias.

In Command 9 we put Query 4 as inner query and sum up the number of grades that were given to students in course 002. The corresponding result is printed in Table 8. Again we need the keyword `PROVENANCE`.

```
SELECT PROVENANCE sum(gcount) FROM (
    SELECT grade, count(*) AS gcount
    FROM grades
    WHERE course_nr = '002'
    GROUP BY grade
    ORDER BY grade
) BASERELATION query4;
```

Command 9: Query 4 with `BASERELATION`

sum	prov_query4_grade	prov_query4_gcount
7	1.0	1
7	1.3	2
7	2.0	1
7	2.3	1
7	3.3	1
7	3.7	1

Table 8: Result table of Command 9

The expected result without the special keywords should be the single value 7. However, it has six rows now. But we can not see all column names of GRADES. Instead the attributes GRADE and GCOUNT are displayed. The subquery which is marked as base relation is not an actual table from our database. Hence Perm is

composing the attribute name using the name of the subquery between `prov` and the name of the projected attribute.

Note the position of the `PROVENANCE` keyword. If we place the keyword to the inner query rather than the outer query, then the result would be a single 9 because we would aggregate Table 7.

Perm has its own GUI named Perm Browser. A screenshot of the Perm Browser is in Figure 8. In the top left corner are some logos from the research group. Under the pictures is the Query textbox. There we type our query in SQL-PLE. If the `PROVENANCE` keyword is used in the Query textbox, the Provenance Query textbox below shows the rewritten query. Additional options are available below the second textbox. Back to the top; right of the logos there are two buttons and a dropdown menu. In the dropdown menu our history is stored. The "run" button starts the execution and the "show Rewrite" button fills the Provenance Query textbox. On the right hand side Perm presents the query algebra tree and the provenance query algebra tree with some buttons for zooming. At the bottom is the result table with the provenance attributes in an alternating color scheme.

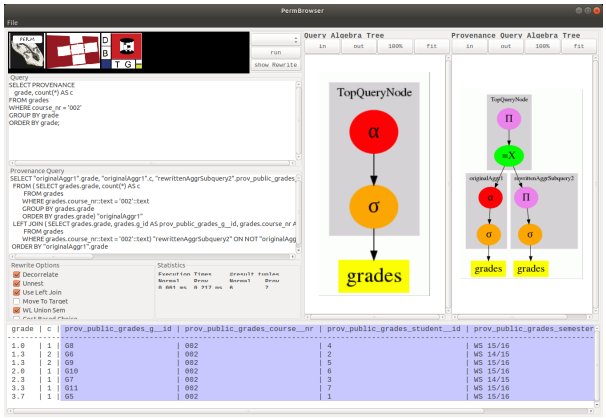

Figure 8: Screenshot of the *Perm Browser*

Conclusion Perm is a modified PostgreSQL server. It extends the SQL syntax in order to capture the provenance. The Perm Module is embedded in the architecture of PostgreSQL and contains a Query Rewriter. Provenance annotations are generated in additional rows when executing a command with the `PROVENANCE` keyword. The Perm Browser combines some of the functionality of Perm. You can view the rewritten query and the result as well as query trees.

3.4 GProM

What is GProM? GProM is a abbreviation for *Generic Provenance Middleware* and is the successor of Perm. It is, as the name suggests, a middleware which supports multiple database management systems like PostgreSQL. The source code is available at GitHub[4] and licensed under the Apache-2.0 license.

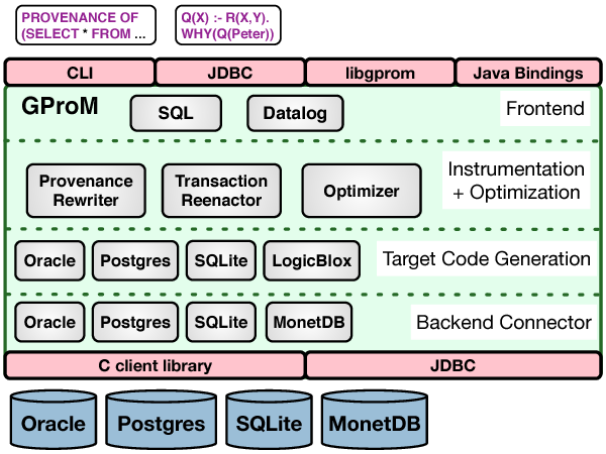

Figure 9: GProM Architecture [Ara+18]

Literature & Provenance According to [Ara+14], GProM uses annotation and query rewrite techniques for "computing, querying, storing, and translating the provenance of SQL queries, updates, transactions, and across transactions". It uses a relational database as backend. MonetDB, Oracle, PostgreSQL and SQLite are available backend systems. The architecture of GProM is illustrated in Figure 9. You can see the different steps from frontend to backend. Using the command-line interface or JDBC, for example, you can connect to GProM. The tool takes care of rewriting and optimizing queries. Then it will generate the code for the specific database backend and finally connects to it. Similar to other provenance tools, GProM uses a relational encoding to process provenance annotations. Incoming statements are translated into algebra graph models based on relational algebra graphs.

GProM is able to compute provenance of concurrent transactions as long as the underlying DBMS saves an audit log and features a time travel functionality. An audit log saves SQL statements, including an identifier and a timestamp. The time travel functionality stores snapshots of a relation so they can be restored if requested. According to the authors, GProM is the only provenance tool able to compute the provenance of concurrent transactions [Ara+18].

Originally, GProM was developed to compute the provenance of SQL queries [Lee+17], but has since been extended to also support Datalog queries. By providing

[4] `https://github.com/IITDBGroup/gprom`

a ***why*** or ***why-not*** question and a datalog query as input, GProM can compute the corresponding provenance of the result. This is done by unifying ***why***- and ***why-not***-provenance using provenance games. To ask why a certain tuple is not present in the output is to ask why this tuple is present in the complement of a result. GProM also includes query optimization based on heuristics and cost-based rules and is designed for extensibility: other provenance notations and models can be added whenever needed.

What we observe We used Ubuntu 20.4 LTS for our tests with GProM. Because of the architecture we needed a frontend and a backend. As backend we decided to use SQLite because it is easy to install. As for the frontend we used standard SQL for the queries and Datalog for the provenance game graphs.

All four test queries produce similar or even identical results to Perm. For comparison to other tools we already described Query 3 in detail (see Section 1.2). In short, we want to know which lecturer teaches which student.

(1)	(2)	(3)	(4)	(5)	...
Donald	Lecturer A	S1	1	Moore	...
Donald	Professor A	S1	1	Moore	...
Donald	Professor C	S1	1	Moore	...
Donald	Professor A	S1	1	Moore	...
Donald	Professor A	S1	1	Moore	...
Sarah	Lecturer A	S2	2	Morgan	...
⋮	⋮	⋮	⋮	⋮	⋱

(1) **FIRSTNAME**
(2) **FULLNAME**
(3) **PROV_STUDENTS_S__ID**
(4) **PROV_STUDENTS_STUDENT__ID**
(5) **PROV_STUDENTS_LASTNAME**

Table 9: Result of Query 3 with Provenance in GProM (first 6 rows)

The result is printed in Table 9. A huge number of extra columns is generated. Because it is such a large table we decided only to show here the very beginning of the table.

Additionally we observed that GProM prints a warning message:

```
Attribute <STUDENT_ID> appears more than
    once in [...]
```

It appears that the equially named column are not resolveable by GProM.

Since provenance game graphs are unique to GProM among out tools, we want to test them as well. As example we will use Query 3 again. This time we need to translate our SQL query into Datalog. The new query is listed in Command 10. When executing this in the command line interface, a picture is created. Figure 10 shows the output.

```
Res(firstname, fullname) :-
    students(s_id, student_id, lastname,
        firstname, study_course),
    participants(p_id, course_nr,
        student_id),
    lecturers(l_id, course_nr, fullname).
WHY(Res('Jack', 'Lecturer B')).
```

Command 10: Query 3 in Datalog

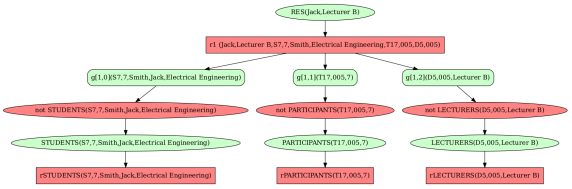

Figure 10: Provenance game graph for Query 3

Conclusion GProM is easy to install and well documented. Multiple database backends are supported. Queries can be written in SQL and Datalog. GProM's unique feature is the production of provenance game graphs. Most features are working as expected. For instance, inner joins lead to error messages. Unlike its predecessor Perm, GProM does not provide a graphical user interface.

3.5 ProvSQL

What is ProvSQL? ProvSQL is an extension for PostgreSQL that was first mentioned in 2017 [Sen17] and presented in 2018 [Sen+18]. The tool uses annotations to calculate provenance polynomials. Also, it is capable of displaying provenance graphs of results. ProvSQL does not modify the source code of PostgreSQL, therefore it is not required to use a specific or modified version of PostgreSQL besides the default versions. Furthermore all features are provided by functions that are used in SQL commands. It is available on GitHub[5] and published under the MIT License. As of today, ProvSQL is still being developed.

Literature & Provenance We can calculate provenance using various semirings. These are handled as described in Section 2.4.4. The user may use pre-defined provenance semirings like *formula* (the

[5] https://github.com/PierreSenellart/provsql/

how-semiring as described in [Sen17]) or the *counting* semiring. The definition of user-defined provenance semirings is possible. ProvSQL supports most common SQL keywords. However, aggregation is not supported. Senellart et al. explain that they have to implement semimodules in order to support aggregations [Sen+18]. In addition ProvSQL supports the calculation of ***where***-provenance.

What we observe Before ProvSQL can be used, the search path of PostgreSQL has to be updated to include the `provsql` schema, as well:

```
SET search_path TO public, provsql;
```

Afterwards, ProvSQL is ready to use. The first step should be the addition of provenance. This can be done by calling the `add_provenane` function on a relation, e.g. `SELECT add_provenance('students');`. The provenance annotations of ProvSQL are based on randomly generated UUIDs. One feature of the tool is the presentation of ***how***-provenance. The user can choose which attribute of the queried table the formulas should contain. This is handled by mappings which, however, have to be created manually by using the user-defined function (UDF) `create_provenance_mapping`. This function takes three arguments: the desired name of the mapping, the name of the table, and the name of the column holding the data. An example can be seen in Command 11. We can observe that calling this UDF creates a new table which gets the name of the mapping.

```
SELECT create_provenance_mapping(
    'students_id_mapping',
    'students',
    's_id'
);
```

Command 11: Calling the UDF `create_provenance_mapping`

Let us take Query 1. When executed, ProvSQL automatically adds a column `provsql` to the result which contains UUIDs for the values. To receive the provenance polynomial of a certain result, e. g. *"Electrical Engineering"*, we can use the user-defined function `formula` which takes the UUID of the result as well as the name of the provenance mapping as arguments. An example can be seen in Command 12. The result is listed in Table 10.

```
SELECT formula(
  '39371d93-587a-5b89-8e8a-89f58ef62f13',
  'students_id_mapping'
);
```

Command 12: Retrieving the provenance polynomial

formula
(S4 ⊕ S5 ⊕ S6)

Table 10: Result of command 12

In this table we see the column name `formula`. It is displayed because we called the corresponding function without giving it an alias. We have only one row in the result which shows us the provenance polynomial. Now we know that the tuples $S4$, $S5$ and $S6$ were combined because they have the same course and therefore are duplicates.

An advantage of ProvSQL is the graphical representation of provenance polynomials and their corresponding graphs. We want to demonstrate it using Query 3. Let us assume we already defined `students_id_mapping` as shown in Command 11. In the same way we create the mappings `participants_id_mapping` with the PARTICIPANTS relation and `lecturers_id_mapping` with the LECTURERS relation, respectively. Since the functions in ProvSQL take only one mapping we have to find a way to combine different mappings. ProvSQL does not provide commands for this task by itself. But because mappings are stored as relations we can merge them by using `UNION`. Command 13 shows how to create a view that brings all three mappings together.

```
CREATE VIEW database_mapping AS (
    (SELECT *
       FROM students_id_mapping)
    UNION
    (SELECT *
       FROM participants_id_mapping)
    UNION
    (SELECT *
       FROM lecturers_id_mapping)
);
```

Command 13: Combining multiple mappings using unions in a view

Now we can continue with the generation of the provenance graph. First we have to execute the original query, e. g. Query 3. Then we take the UUID from the row that we are interested in. Remember that ProvSQL adds a column of the name `provsql` automatically which contains the UUIDs. In this case we take the first *("Donald", "Professor A")* value. The corresponding UUID is *"aa2f7623-3635-5b3b-9909-ef5aea6b81e8"*. Next we add it as the first argument for

the `view_circuit` function. As the second parameter we add the `database_mapping` from Command 13 in order to display the different IDs. Command 14 shows the complete function call.

```
SELECT view_circuit(
  'aa2f7623-3635-5b3b-9909-ef5aea6b81e8',
  'database_mapping'
);
```

Command 14: Using the UDF `view_circuit` to generate the provenance circuit

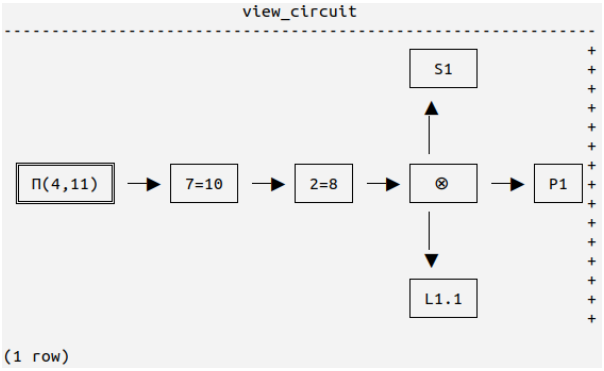

Figure 11: Provenance circuit of "aa2f7623-3635-5b3b-9909-ef5aea6b81e8"

ProvSQL prints the graph as an ASCII image. Figure 11 shows the result. The root of the graph represents the projection. Imagine you write down the column names of STUDENTS, PARTICIPANTS and LECTURERS and number them consecutively from one onwards. The numbers in the parentheses are the indices of our list. For this example, the 4 is the fourth column of STUDENTS, FIRSTNAME and the 11 is the second column of LECTURERS, FULLNAME. The expressions `7=10` and `2=8` are caused by the inner join. They use the same numbering as the projection and are referencing to the join criteria in the `WHERE` clause. Finally, the last four nodes are the tuples *S1*, *P1* and *L1.1* which were combined with the ⊗ operator. We see that these three tuples were combined in a join. ProvSQL shows the IDs in the graph because we put them in the mapping.

Conclusion ProvSQL is the latest tool in our collection. It does not depend on a specific version of PostgreSQL and is easy to install. The tool provides its interface via user-defined functions. The calculation of ***how***- and ***where***-provenance is possible. Various semirings can be used. ProvSQL is capable of generating *provenance circuits* which represent the semiring formulas. Most common SQL keywords are supported except aggregations.

3.6 Tioga

What is Tioga? Tioga is the oldest tool that matches our selection critera. It was developed in 1993 as a tool to detect forest fires and thus one of the first provenance tools. Unfortunate, the project website and the source code was not available anymore. Thereupon we tried to contact several authors. We received only a few answers; Tioga had not been published at that time. Then we tried the successor, Tioga 2. Later it was renamed in DataSplash. The project website[6] looked promising.

Literature & Provenance Stonebraker et al. describe Tioga in their article "Tioga: Providing Data Management Support for Scientific Visualization Applications" [Sto+93]. At the early time there existed programs which presented directed graphs through boxes and arrows. Tioga generalises this concept and calls the diagrams *recipes*. Analogue to a cooking recipe will the boxes represent the ingredients. Technically there are the symbolization of functions in Postgres. Various functions are working as the data passes through. This is comparible with the workflow provenance in Section 2.2. An recipe example is shown in Figure 12.

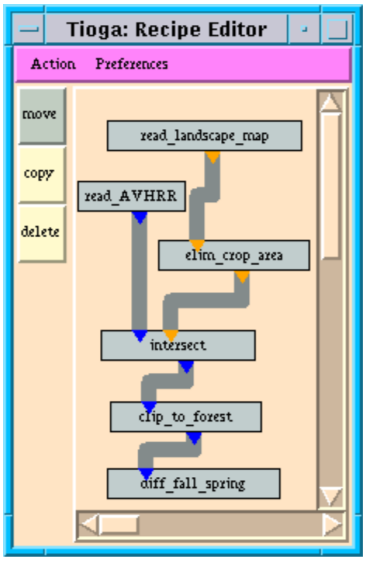

Figure 12: Recipe Editor in Tioga [Sto+93]

Tioga should be able to do ***how***-provenance [PRS18]. This should be on attribute level. Thus we couln'd test Tioga we cannot confirm the literature results of this paper. Furthermore we tried DataSplash, the successor of Tioga.

[6] `http://datasplash.cs.berkeley.edu/`

What we observe Thou we couldn't test Tioga because the source code is not available we decided to give DataSplash a try. The first thing we did was to check which Linux version was up to date, Debian 3. The installation of Debian 3 in a virtual machine was not easy due to its age. After that, we were able to successfully install DataSplash. But it was not properly working. The official sample looked like Figure 13. It was the best result we could get.

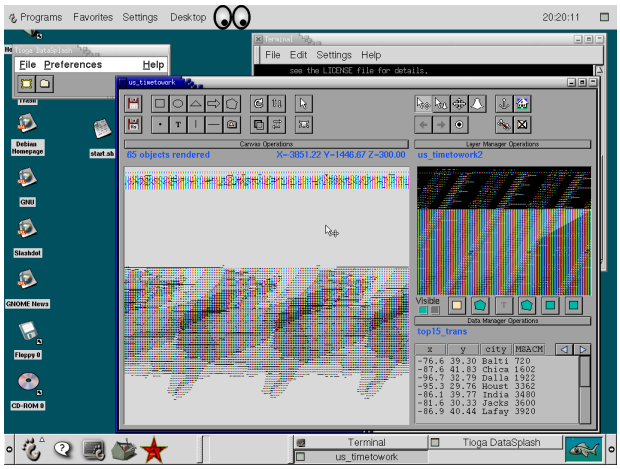

Figure 13: How DataSplash actually looks

We also tried Debian 4, Debian2, FreeBSD and Ubuntu 18.04 LTS. But we had either issues with the operability of the older operating systems in virtual machines or `segmentation faults` that we couldn't solve. On Ubuntu 18.04 LTS we discovered compilation errors and not executable binaries. Particularly the C-Library `libc.so.5 vs. libc.so.6` was not performable on this Ubuntu version.

Conclusion Tioga and DataSplash are very interessing programs. Especially because there are the first and oldest data provenance tools we have record off. Unfortunately the source code is not available for Tioga and DataSplash got a lot of problems that need much more effort to solve. Therefore, it was no longer suitable for us to continue with this project, and we decided to work on other tools like ProvC&B.

3.7 ProvC&B

What is ProvC&B? ProvC&B [Ile+14a] is a tool made by Ioana Ileana in the year 2014 implementing the similar named *Provenance-directed Chase&Backchase* algorithm [DH13].

Literature & Provenance In opposition to previous presented applications of provenance in section 2.3 ProvC&B is using the ***why***-provenance (see 2.4.3) to improve the performance of answering queries using views (AQuV) via CHASE and BACKCHASE (also called C&B). AQuV is a method to calculate answers to a given query while only using the ressources that are visible to a specific user in the form of views instead of using base relations. An increased need for privacy could be one motivation for using views here. The CHASE is an algorithm that, poorly explained, incorporates integrity constraints into specific objects. The BACKCHASE is just another CHASE that uses the result of the first CHASE as the new object in order to reobtain the structure of the previous chased object. The integrity constraints used in CHASE and BACKCHASE are just the views that are rewritten into dependencies. Therefore chasing a query with the views delivers a universal plan only containing views after cutting out the base relations. This universal plan is then chased backwards in order to obtain the base relations that are computable only using previously determined views. In order to find the minimal amount of views necessary to obtain the closest result to the original query we need to find mappings from the views to the base relations. To find those mappings we need to consider every subset of views and therefore the power quantity which is algorithmically expensive to check for mappings. Using the ***why***-provenance here to annotate such views and resulting relations undermines the necessity of computing and checking such mappings which results in reduced computational effort. Because of the nature of the CHASE used in the algorithm we can answer AQuV-problems even under additional integrity constraints. For a better understanding of the C&B and ProvC&B algorithm we would like to refer to [DH13].

What we observe Because ProvC&B is an Java project we were able to test this on an up to date windows operating system without noticing any differences. Unfortunatly, four problems quickly emerged. First of all, the Java code was poorly documented or not at all documented. This caused some understanding problems which leads us to the second problem. The code is relativly difficult to understand. We cannot gain enough knowledge about what this tool wants to do from the source code. The thrid problem is the input. We are not sure about it because there is no documentation about is at all. The paper [Ile+14a] uses same names for some input variables. Because of this we highly believe there are the same but we couldn't confirm this. Also some input variables could have to do something this benchmarking. The last and fourth problem for us were the output. It contains only 3 numbers. A guess is that one of the numbers is the execution time and the other

two numbers about benchmarking. But there was no way for us to find out about it.

Conclusion Even if the tooltest wasn't much of a success ProvC&B raises the utilization of provenance to another level. Using the ***why***-provenance actively to decrease computational effort is an indicator that provenance as a research topic will have more surprises in store than just improving understandibility, reproducibility or quality.

4 Conclusion

In this paper, we first investigated different kinds of provenance. In Section 2, we defined the term "provenance" and took a look at the different applications and types of provenance, especially data provenance. This type of provenance consists of ***where***-, ***why***-, ***how***-, and ***why-not***-provenance. We saw that ***why*** can be derived from ***where*** and ***how*** can be derived from ***why***. Additionally, we investigated ***data lineage*** which can be considered as predecessor of ***why***-provenance.

In Section 3, we benchmarked seven provenance tools: Trio, ORCHESTRA, Perm, GProM, ProvSQL, Tioga, and ProvC&B. Unfortunately, we were unable to test Tioga and ProvC&B due to technical difficulties. A brief summary of our test results can be seen in Table 11.

We conclude that Trio has a clear user interface provided by TrioExplorer, even though some implementations are missing. We were unable to find any documentation of TrioPlus, the command-line interface version of the tool. Trio is able to compute three of our four test queries, lacking support for Query 1 (duplication elimination). Query 4 however, only works when using Trio's own horizontal subqueries. The tool can handle ***why***- and ***where***-provenance. Unfortunately, Trio lacks support for ***how***-provenance. Also, Trio cannot handle `DELETE` and `UPDATE` commands, although they are described in the documentation of TrioQL, Trio's own query language. Aggregations are only supported when using horizontal subqueries.

ORCHESTRA is able to handle all queries except Query 4 (aggregations). Aggregations are not supported at all, joins, however, work fine. Besides that, ORCHESTRA is able to compute our three types of data provenance, ***where*** ***why*** and ***how***. The ***why***-provenance is implemented on tuple granularity. Since ORCHESTRA uses Datalog as query language, duplication elimination is supported by nature. The tool uses its own GUI that can display provenance graphs for ***how***-provenance.

Perm is built on top of PostgreSQL and extends the syntax of SQL for computing provenance by using a special `PROVENANCE` statement. It comes with a GUI called Perm Browser and a query rewriter that translates special provenance queries to common SQL queries. Perm is the only tool besides – with some limitations – GProM that can process all of our four queries. It is able to compute ***why***- and ***where***-provenance but, unfortunately, lacks support for ***how***-provenance.

When we tested GProM, we discovered that it is well documented and easy to install. It supports four different database backends and SQL as well as Datalog as frontend languages. What makes GProM unique is its ability to generate provenance game graphs, hence it is able to handle ***why***- and ***why-not***-provenance. GProM is able to compute all of our four queries, although Query 2 has to be rewritten because GProM cannot handle the `JOIN` keyword. GProM is one out of two tools we have tested that is still being actively developed.

ProvSQL is the second tool tested that is still maintained by its developers. It's an extension for PostgreSQL and therefore does not require any specific version of it. ProvSQL works by using user-defined functions for provenance computation. It is noteworthy that ProvSQL can handle ***where***- and ***why***- as well as ***how***-provenance and is also able to produce provenance circuits for ***how***-provenance, based on provenance semirings. The tool can process Query 1 to 3 but lacks support for Query 4 because aggregations are not supported yet.

Unfortunately, we were not able to test Tioga and DataSplash. The source code of the former is not available (anymore) and due to technical problems, we were unable to test the latter. Although the tool tests were not very successful, we think that Tioga and DataSplash are quite interesting tools.

Besides Tioga, we were also unable to test ProvC&B due to poorly documented code, unknown inputs and unreadable outputs. However, we believe that ProvC&B might be a powerful tool because it uses ***why***-provenance not for tracking origins of data or understandability but for optimizing an algorithm. ProvC&B therefore is an example for the importance of research on data provenance.

Table 11 shows a summary of our test results. As can be seen, two out of the five working tools – Perm and GProM – can handle all of our four queries. Two other of them – ORCHESTRA and ProvSQL – can compute all three main kinds of data provenance. Besides that, the table shows if a tool is still under active development as well as its publishing license. Interestingly, the tools that are able to calculate the ***how***-provenance cannot handle Query 4which involves aggregations. ***How***-provenance for aggregations is still being researched and therefore, tools that can deal with ***how***-provenance, might have exceeded its limit for now.

Tool	Query				Provenance types			Under active development?	License
	1	**2**	**3**	**4**	***How***	***Why***	***Where***		
Trio	✗	✓	✓	✓[7]	✗	✓	✓	✗	BSD License
ORCHESTRA	✓	✓	✓	✗	✓ (Graph)	✓	✓	✗	Apache License, Version 2.0
Perm	✓	✓	✓	✓	✗	✓	✓	✗	PostgreSQL License
GProM	✓	✓[8]	✓	✓	✗	✓	✓	✓	Apache License, Version 2.0
ProvSQL	✓	✓	✓	✗	✓	✓	✓	✓	MIT License

Table 11: Tool comparison table

5 References

[Sto+93] Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson, and Jiang Wu. "Tioga: Providing Data Management Support for Scientific Visualization Applications". In: *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*. Ed. by Rakesh Agrawal, Seán Baker, and David A. Bell. Morgan Kaufmann, 1993, pp. 25–38. URL: http://www.vldb.org/conf/1993/P025.PDF.

[BKT00] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. "Data Provenance: Some Basic Issues". In: *Foundations of Software Technology and Theoretical Computer Science, 20th Conference, FST TCS 2000 New Delhi, India, December 13-15, 2000, Proceedings*. Ed. by Sanjiv Kapoor and Sanjiva Prasad. Vol. 1974. Lecture Notes in Computer Science. Springer, 2000, pp. 87–93. DOI: 10.1007/3-540-44450-5_6. URL: https://doi.org/10.1007/3-540-44450-5%5C_6.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. "Why and Where: A Characterization of Data Provenance". In: *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings*. Ed. by Jan Van den Bussche and Victor Vianu. Vol. 1973. Lecture Notes in Computer Science. Springer, 2001, pp. 316–330. DOI: 10.1007/3-540-44503-X_20. URL: https://doi.org/10.1007/3-540-44503-X%5C_20.

[Agr+06] 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: *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*. Ed. by Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim. ACM, 2006, pp. 1151–1154. URL: http://dl.acm.org/citation.cfm?id=1164231.

[Ben+06] Omar Benjelloun, Anish Das Sarma, Chris Hayworth, and Jennifer Widom. *An Introduction to ULDBs and the Trio System*. Technical Report 2006-7. Stanford InfoLab, 2006. URL: http://ilpubs.stanford.edu:8090/793/.

[Mut+07] Michi Mutsuzaki, Martin Theobald, Ander de Keijzer, Jennifer Widom, Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Raghotham Murthy, and Tomoe Sugihara. "Trio-One: Layering Uncertainty and Lineage on a Conventional

[7] Works only if using horizontal subqueries.

[8] Works only if not using the keyword JOIN

DBMS (Demo)". In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.cidrdb.org, 2007, pp. 269–274. URL: http://cidrdb.org/cidr2007/papers/cidr07p30.pdf.

[CCT09] James Cheney, Laura Chiticariu, and Wang Chiew Tan. "Provenance in Databases: Why, How, and Where". In: *Found. Trends Databases* 1.4 (2009), pp. 379–474. DOI: 10.1561/1900000006. URL: https://doi.org/10.1561/1900000006.

[GA09] Boris Glavic and Gustavo Alonso. "Perm: Processing Provenance and Data on the same Data Model through Query Rewriting". In: *Proceedings of the 25th IEEE International Conference on Data Engineering*. 2009, pp. 174–185.

[Gla10] Boris Glavic. "Perm: Efficient Provenance Support for Relational Databases". PhD thesis. University of Zurich, 2010.

[Gre+10] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. "Provenance in ORCHESTRA". In: *IEEE Data Eng. Bull.* 33.3 (2010), pp. 9–16.

[DH13] Alin Deutsch and Richard Hull. "Provenance-Directed Chase&Backchase". In: *In Search of Elegance in the Theory and Practice of Computation: Essays Dedicated to Peter Buneman*. Ed. by Val Tannen, Limsoon Wong, Leonid Libkin, Wenfei Fan, Wang-Chiew Tan, and Michael Fourman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 227–236. ISBN: 978-3-642-41660-6. DOI: 10.1007/978-3-642-41660-6_11. URL: https://doi.org/10.1007/978-3-642-41660-6_11.

[GMA13] Boris Glavic, Renée J. Miller, and Gustavo Alonso. "Using SQL for Efficient Generation and Querying of Provenance Information". In: *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman* (2013), pp. 291–320.

[Ara+14] Bahareh Arab, Dieter Gawlick, Venkatesh Radhakrishnan, Hao Guo, and Boris Glavic. "A Generic Provenance Middleware for Database Queries, Updates, and Transactions". In: *Proceedings of the 6th USENIX Workshop on the Theory and Practice of Provenance*. 2014.

[Ile+14a] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. "Complete yet Practical Search for Minimal Query Reformulations under Constraints". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 1015–1026. ISBN: 9781450323765. DOI: 10.1145/2588555.2593683. URL: https://doi.org/10.1145/2588555.2593683.

[Ile+14b] Ioana Ileana, Bogdan Cautis, Alin Deutsch, and Yannis Katsis. "Complete yet practical search for minimal query reformulations under constraints". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. Ed. by Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu. ACM, 2014, pp. 1015–1026. DOI: 10.1145/2588555.2593683. URL: https://doi.org/10.1145/2588555.2593683.

[Wil+16] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao,

and Barend Mons. “The FAIR Guiding Principles for scientific data management and stewardship”. In: *Scientific Data* 3.1 (Mar. 15, 2016), p. 160018. ISSN: 2052-4463. DOI: 10.1038/sdata.2016.18. URL: https://doi.org/10.1038/sdata.2016.18.

[GT17] Todd J. Green and Val Tannen. “The Semiring Framework for Database Provenance”. In: *PODS*. ACM, 2017, pp. 93–99.

[HDB17] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. “A survey on provenance: What for? What form? What from?” In: *VLDB J.* 26.6 (2017), pp. 881–906. DOI: 10.1007/s00778-017-0486-1. URL: https://doi.org/10.1007/s00778-017-0486-1.

[Lee+17] Seokki Lee, Sven Köhler, Bertram Ludäscher, and Boris Glavic. “A SQL-Middleware Unifying Why and Why-Not Provenance for First-Order Queries”. In: *ICDE*. IEEE Computer Society, 2017, pp. 485–496.

[Sen17] Pierre Senellart. “Provenance and Probabilities in Relational Databases: From Theory to Practice”. In: *SIGMOD Record* 46.4 (Dec. 2017).

[Ara+18] Bahareh Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. “GProM - A Swiss Army Knife for Your Provenance Needs”. In: *IEEE Data Engineering Bulletin* 41.1 (2018), pp. 51–62.

[DFG18] Daniel Deutch, Nave Frost, and Amir Gilad. “Provenance for Non-Experts”. In: *IEEE Data Eng. Bull.* 41.1 (2018), pp. 3–14. URL: http://sites.computer.org/debull/A18mar/p3.pdf.

[PRS18] Beatriz Pérez, Julio Rubio, and Carlos Sáenz-Adán. “A systematic review of provenance systems”. In: *Knowl. Inf. Syst.* 57.3 (2018), pp. 495–543. DOI: 10.1007/s10115-018-1164-3. URL: https://doi.org/10.1007/s10115-018-1164-3.

[Sen+18] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. “ProvSQL: Provenance and Probability Management in PostgreSQL”. In: *Proc. VLDB*. Demonstration. Rio de Janeiro, Brazil, Aug. 2018, pp. 2034–2037.

A Appendix

A.1 Tables of the university database

s_id	student_id	lastname	firstname	study_course
S_1	1	Moore	Donald	Computer Science for Teaching
S_2	2	Morgan	Sarah	Mathematics
S_3	3	Wood	Jack	Electrical Engineering
S_4	4	Harrison	Elisabeth	Computer Science
S_5	5	Williams	John	Computer Science
S_6	6	William	Mary	Computer Science
S_7	7	Smith	Jack	Electrical Engineering
S_8	8	John	Jennifer	Theoretical Computer Science

Table 12: Table STUDENTS

c_id	course_nr	title
C_1	001	Database Systems and Data Science
C_2	002	Operating Systems
C_3	003	Artificial Intelligence
C_4	004	Probability Theory
C_5	005	Algorithms and Data Structures
C_6	006	Object-Oriented Programming
C_7	007	Law and Computer Science
C_8	008	Data Warehouses and Business Intelligence
C_9	009	Networks and Cybersecurity

Table 13: Table COURSES

l_id	course_nr	fullname
$L_{1.1}$	001	Professor A
$L_{1.2}$	001	Lecturer A
L_2	002	Professor B
L_3	003	Professor C
L_4	004	Professor D
L_5	005	Lecturer B
L_6	006	Professor D
L_7	007	Professor A
L_8	008	Professor E
L_9	009	Professor A

Table 14: Table LECTURERS

p_id	course_nr	student_id
P_1	001	1
P_2	001	2
P_3	001	4
P_4	001	5
P_5	002	1
P_6	002	2
P_7	002	3
P_8	002	4
P_9	002	5
P_{10}	002	6
P_{11}	002	7
P_{12}	003	1
P_{13}	004	3
P_{14}	004	5
P_{15}	004	7
P_{16}	005	4
P_{17}	005	7
P_{18}	006	4
P_{19}	006	5
P_{20}	007	1
P_{21}	007	3
P_{22}	007	5
P_{23}	008	3
P_{24}	009	1
P_{25}	009	4
P_{26}	009	5

Table 15: Table PARTICIPANTS

g_id	course_nr	student_id	semester	grade
G_1	001	1	SS 16	2.0
G_2	001	2	SS 16	1.7
G_3	001	4	SS 16	1.7
G_4	001	5	SS 16	3.0
G_5	002	1	WS 15/16	3.7
G_6	002	2	WS 14/15	1.3
G_7	002	3	WS 14/15	2.3
G_8	002	4	WS 15/16	1.0
G_9	002	5	WS 15/16	1.3
G_{10}	002	6	WS 15/16	2.0
G_{11}	002	7	WS 15/16	3.3
G_{12}	003	1	WS 16/17	1.0
G_{13}	004	3	WS 16/17	1.3
G_{14}	004	5	WS 16/17	3.0
G_{15}	005	4	SS 17	2.7
G_{16}	005	7	SS 17	1.7
G_{17}	006	4	SS 17	2.7
G_{18}	006	5	SS 17	4.0
G_{19}	007	1	SS 16	2.3
G_{20}	007	3	SS 16	1.7
G_{21}	009	1	SS 16	3.3
G_{22}	009	5	SS 15	5.0
G_{23}	009	5	SS 16	2.7

Table 16: Table GRADES

A.2 Queries

```
SELECT DISTINCT study_course FROM students;
```

Command 15: Query 1

study_course	*how*	*why*	*where*	*lineage*
Computer Science	$S_3 \oplus S_7$	$\{\{S_3\},\{S_7\}\}$	STUDENTS	S_3, S_7
Computer Science for Teaching	S_1	$\{\{S_1\}\}$	STUDENTS	S_1
Electrical Engineering	$S_4 \oplus S_5 \oplus S_6$	$\{\{S_4\},\{S_5\},\{S_6\}\}$	STUDENTS	S_4, S_5, S_6
Mathematics	S_2	$\{\{S_2\}\}$	STUDENTS	S_2
Theoretical Computer Science	S_8	$\{\{S_8\}\}$	STUDENTS	S_8

Table 17: Result of Query 1 with Provenance

```
SELECT
    s.s_id, s.lastname, s.firstname,
    p.p_id, p.course_nr
FROM
    students s INNER JOIN
    participants p ON s.student_id = p.student_id
```

```
WHERE
    p.course_nr = '005'
;
```

Command 16: Query 2

s_id	lastname	firstname	p_id	course_nr	*how*	*why*	*where*	*lineage*
S_4	Harrison	Elisabeth	P_{16}	005	$S_4 \otimes P_{16}$	$\{\{S_4, P_{16}\}\}$	STUDENTS, PARTICIPANTS	S_4, P_{16}
S_7	Smith	Jack	P_{17}	005	$S_7 \otimes P_{17}$	$\{\{S_7, P_{17}\}\}$	STUDENTS, PARTICIPANTS	S_7, P_{17}

Table 18: Result of Query 2 with Provenance

```
SELECT s.firstname, l.fullname
FROM students s, participants p, lecturers l
WHERE s.student_id = p.student_id
AND p.course_nr = l.course_nr;
```

Command 17: Query 3

firstname	fullname	*how*	*why*	*where*	*lineage*
Donald	Lecturer A	$S_1 \otimes P_1 \otimes L_{1.2}$	$\{\{S_1, P_1, L_{1.2}\}\}$	STUDENTS, LECTURERS	$S_1, P_1, L_{1.2}$
Donald	Professor A	$S_1 \otimes P_1 \otimes L_{1.1}$	$\{\{S_1, P_1, L_{1.1}\}\}$	STUDENTS, LECTURERS	$S_1, P_1, L_{1.1}$
Sarah	Lecturer A	$S_2 \otimes P_2 \otimes L_{1.2}$	$\{\{S_2, P_2, L_{1.2}\}\}$	STUDENTS, LECTURERS	$S_2, P_2, L_{1.2}$
Sarah	Professor A	$S_2 \otimes P_2 \otimes L_{1.1}$	$\{\{S_2, P_2, L_{1.1}\}\}$	STUDENTS, LECTURERS	$S_2, P_2, L_{1.1}$
Elisabeth	Lecturer A	$S_4 \otimes P_3 \otimes L_{1.2}$	$\{\{S_4, P_3, L_{1.2}\}\}$	STUDENTS, LECTURERS	$S_4, P_3, L_{1.2}$
Elisabeth	Professor A	$S_4 \otimes P_3 \otimes L_{1.1}$	$\{\{S_4, P_3, L_{1.1}\}\}$	STUDENTS, LECTURERS	$S_4, P_3, L_{1.1}$
John	Lecturer A	$S_5 \otimes P_4 \otimes L_{1.2}$	$\{\{S_5, P_4, L_{1.2}\}\}$	STUDENTS, LECTURERS	$S_5, P_4, L_{1.2}$
John	Professor A	$S_5 \otimes P_4 \otimes L_{1.1}$	$\{\{S_5, P_4, L_{1.1}\}\}$	STUDENTS, LECTURERS	$S_5, P_4, L_{1.1}$
...	...	...	...	...	...

Table 19: Result of Query 3 (first 8 rows) with Provenance

firstname	fullname
Donald	Lecturer A
Donald	Professor A
Sarah	Lecturer A
Sarah	Professor A
Elisabeth	Lecturer A
Elisabeth	Professor A
John	Lecturer A
John	Professor A
Donald	Professor B
Sarah	Professor B
Jack	Professor B
Elisabeth	Professor B
John	Professor B
Mary	Professor B
Jack	Professor B
Donald	Professor C
Jack	Professor D
John	Professor D
Jack	Professor D
Elisabeth	Lecturer B
Jack	Lecturer B
Elisabeth	Professor D
John	Professor D
Donald	Professor A
Jack	Professor A
John	Professor A
Jack	Professor E
Donald	Professor A
Elisabeth	Professor A
John	Professor A

Table 20: Full Result of Query 3

```
SELECT grade, count(*) AS gcount
FROM grades
WHERE course_nr = '002'
GROUP BY grade
ORDER BY grade;
```

Command 18: Query 4

grade	gcount	*how*	*why*	*where*	*lineage*
1.0	1	G_8	$\left\{\{G_8\}\right\}$	GRADES	G_8
1.3	2	$G_6 \oplus G_9$	$\left\{\{G_6, G_9\}\right\}$	GRADES	G_6, G_9
2.0	1	G_{10}	$\left\{\{G_{10}\}\right\}$	GRADES	G_{10}
2.3	1	G_7	$\left\{\{G_7\}\right\}$	GRADES	G_7
3.3	1	G_{11}	$\left\{\{G_{11}\}\right\}$	GRADES	G_{11}
3.7	1	G_5	$\left\{\{G_5\}\right\}$	GRADES	G_5

Table 21: Result of Query 4 with Provenance