Bulletin of the Technical Committee on

Data Engineering

March 2018 Vol. 41 No. 1

IEEE Computer Society

Letters

Letter from the Editor-in-Chief . . . . *David Lomet* 1
Letter from the Special Issue Editor . . . . *Tova Milo* 2

Special Issue on Applications of Provenance

Provenance for Non-Experts . . . . *Daniel Deutch, Nave Frost, Amir Gilad* 3
Provenance and the Different Flavors of Reproducibility . . . . *Juliana Freire, Fernando Seabra Chirigati* 15
Data Citation: A New Provenance Challenge . . . . *Abdussalam Alawini, Susan Davidson, Gianmaria Silvello, Val Tannen, Yinjun Wu* 27
Provenance Analysis for Missing Answers and Integrity Repairs . . . . *Jane Xu, Waley Zhang, Abdussalam Alawini, Val Tannen* 39
GProM - A Swiss Army Knife for Your Provenance Needs . . . . *Bahareh S. Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, Qitian Zeng* 51
Supporting Data Provenance in Data-Intensive Scalable Computing Systems . . *Matteo Interlandi, Tyson Condie* 63
Data Center Diagnostics with Network Provenance . . . . *Ang Chen, Chen Chen, Lay Kuan Loh, Yang Wu, Andreas Haeberlen, Limin Jia, Boon Thau Loo, Wenchao Zhou* 74

Conference and Journal Notices

ICDE 2018 Conference 86
TCDE Membership Form . . . . back cover

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is `http://tab.computer.org/tcde/index.html`.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at `http://tab.computer.org/tcde/bull_about.html`.

Letter from the Editor-in-Chief

ICDE 2018

Repeating my comment from the last issue-

The IEEE International Conference on Data Engineering will be held in April 14-19, 2018 in Paris, France. This is the flagship conference of the Computer Society's Technical Committee on Data Engineering. It is a great conference, at a great location. What could possibly be better than April in Paris at ICDE! I am attending and hope to see you there.

About the Bulletin

This March current issue marks the end of editorial tenure for the Bulletin's current set of editors. So it is once again time for me to pat myself on the back. This current set, Tim Kraska, Tova Milo, and Haixun Wang, continue my outstanding success (he says modestly) in finding and choosing great editors. All three have done truly fine jobs at producing issues that bring to our readers surveys of the latest work in very current areas. The success of the Bulletin depends on great issue editors. I want to thank Tim, Tova, and Haixun for being exactly that with the fine jobs that they have done. There was unexpected "scrambling" over the past two years, so I want to thank them also for their flexibility in coping with this.

The Current Issue

Don't you get tired of someone shouting "FAKE NEWS!". Or perhaps even worse, being exposed to fake news before it has been labeled as such? Our political conversations seem increasingly to include many variations of "fake news" and discussions about which news is fake. "Sad." So where am I going with this?

The database world has been working on a key aspect of this problem for many years. The technical area is called "data provenance". And it addresses the problem of where information comes from and how it impacts the subsequent processing of data and the reported results. The June, 2010 issue was the last one on provenance. And seven years is a long time in an active technical area, especially an area as important as this.

The current issue is focused on the applications of provenance. Without delving into the current political scene, a reader will clearly see how extensively provenance management can be used. As we gain more insight into its application scenarios, we also gain more insight into how to manage provenance. This symbiotic relationship is driving the field forward. Tova Milo, our Bulletin editor for the issue, has done an excellent job in bringing the issue together, making it a great place to learn about and track progress in the field. The result is an issue well worth reading.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Over the past years there has been a growing recognition of the importance of provenance in data management. Besides it traditional use for query results explanations, new applications of provenance range from query optimization, to distributed data management, human-machine interaction, and experiments reproducibility. In this issue, we have a slate of very interesting articles discussing the different roles of provenance in information management in a variety of domains.

We start with "Provenance for non-experts", Daniel Deutch, Nave Frost and Amir Gilad. The paper considers the flourish of data-intensive systems that are geared towards direct use by non-experts, such as Natural Language question answering systems and query-by-example frameworks. It highlights the importance of incorporating provenance in building such user interfaces.

The second paper, "Provenance and the Different Flavors of Reproducibility", Juliana Freire and Fernando Seabra Chirigati, considers the important problem of experiments reproducibility. It provides an overview of the different types of provenance and how they influence reproducibility. The goal here is to help researchers find the most appropriate provenance capture for their experiment, based on the level of reproducibility they need to attain.

The next paper "Data Citation: A New Provenance Challenge", Abdu Alawini, Susan Davidson, Gianmaria Silvello, Val Tannen and Yinjun Wu, proposes a provenance-based novel framework of the citation of query results. The proposed solution is to specify citations for a small set of frequent queries citation views and then use these views to construct a citation for general queries.

The fourth paper, "Provenance Analysis for Missing Answers and Integrity Repairs", Jane Xu, Waley Zhang, Abdussalam Alawini, and Val Tannen, points that prior approaches for provenance used positive provenance and were thus not directly usable for explaining missing answers or failure of integrity constraints. The paper addressee this shortcoming by offering provenance-based explanations via (minimal) repairs, applicable for debugging, repairing, and cleaning databases.

In "GProM - a Swiss Army Knife for Your Provenance Needs", Bahareh Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu and Qitian Zeng, provided an overview of GProM, a novel generic provenance middleware for relational databases. The system supports diverse provenance and annotation management tasks through query instrumentation.

Next, "Supporting Data Provenance in DISC Systems", Matteo Interlandi and Tyson Condie, uses data provenance as a key building block to provide debugging support for data processing pipelines. Specifically, the paper reports experience in building Titian: a data provenance system targeting the Apache Spark framework.

Finally, we conclude with "Data Center Diagnostics with Network Provenance", Ang Chen, Chen Chen, Lay Kuan Loh, Yang Wu, Andreas Haeberlen, Limin Jia, Boon Thau Loo and Wenchao Zhou. Diagnosing problems in data centers are a challenging problem due to their complexity and heterogeneity. The promising approach described in the paper leverages provenance, which provides the fundamental functionality that is needed for performing fault diagnosis and debugging.

I hope that you enjoy the issue as much as I enjoyed putting it together!

Tova Milo
Tel Aviv University

Provenance for Non-Experts

Daniel Deutch, Nave Frost, Amir Gilad
Tel Aviv University

Abstract

The flourish of data-intensive systems that are geared towards direct use by non-experts, such as Natural Language question answering systems and query-by-example frameworks calls for the incorporation of provenance management. Provenance is arguably even more important for such systems than for "classic" database application. This is due to the elevated level of uncertainty associated with the typical ambiguity of user specification (e.g. phrasing questions in Natural Language or through examples). Existing provenance solutions are not geared towards the non-experts, and the typical complexity and size of their instances render them ill-suited for this goal. We outline in this paper our ongoing research and preliminary results, addressing these challenges towards developing provenance solutions that serve to explain computation results to non-expert users.

1 Introduction

In the context of data-intensive systems, data provenance captures the way in which data is used, combined and manipulated by the system. Provenance information can for instance be used to reveal whether data was illegitimately used, to assess the trustworthiness of a computation result, or to explain the rationale underlying the computation. As data-intensive systems constantly grow in use, in complexity and in the size of data they manipulate, provenance tracking becomes of paramount importance. In its absence, it is next to impossible to follow the flow of data through the system. This in turn is extremely harmful for the quality of results, for enforcing policies, and for the public trust in the systems.

The focus of the present paper is on provenance tracking and presentation, geared towards the *non-expert* user. There is a large body of research and development on database interfaces for non-experts, such as Natural Language Interfaces [25, 26, 32, 39, 40] or exploratory systems such as query-by-example frameworks [1, 12, 29, 33, 36, 37, 42, 45]. Provenance is arguably even more important for such systems than it is for "classic" database applications, since uncertainty is far greater: there may be errors in the way the user has phrased the question and in the way the system has understood what she asked; even in the absence of errors, there are usually many valid queries that are consistent with the user's input. To this end, there is typically a phase of interaction, where the system refines its set of possible queries based on user feedback; provenance could potentially be highly useful for this phase as well as for the end result: it could allow users to understand what is the inferred query or set of candidate queries and whether they fit their intended semantics.

One may consider the use of existing provenance solutions for this purpose. After all, many of the above mentioned interfaces for non-experts eventually compile the user input into a formal query. Can we use a

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

provenance model designed for the underlying query language? The challenge is in that existing provenance solutions are not suitable for presentation to non-experts but rather focus on provenance storage through an internal representation and then its use for analysis. Specifically, the resulting provenance is typically both too complex and too large-scale to allow for its direct presentation to non-experts. This requires the development of dedicated solutions.

This paper outlines our recent and ongoing work on provenance tracking and presentation that are geared towards non-expert users. We discuss three application domains, as follows. The first two are database interfaces for non-experts of different flavours, namely Natural Language Interfaces and query-by-example frameworks. The third application domain is very different, involving explanations for results of Machine Learning models. We next identify several overarching principles that apply across domains and allow to reduce the complexity and scale of provenance, towards its presentation to non-experts.

- We claim that in order to be understandable to non-expert users, there should be a tight **coupling between the way provenance information is presented, and the standard user interaction with the system**. For example, if the user asks questions in Natural Language, then it is natural to present provenance information in a similar form; in a completely different domain, if one uses an image tagging application, then it is natural to depict provenance information as a layer on top of the image. This principle was proven successful in the context of (coarse-grained) workflow provenance [19, 38], where provenance is represented and shown in a graphic manner that is very similar to the way developers design the workflow itself; we claim that it is a key principle in the context of provenance presentation for non-experts as well.

- The sheer size of provenance calls for solutions that **summarize** it. Provenance summarization has been studied in multiple contexts, where summaries may be "lossless", i.e. involve no loss of information, as is the case with the work on factorized representation [4, 24, 31], or "lossy" as in [2, 13, 34]. In the context of non-experts, summarization needs not only to be concise but also to "make sense" to the non-expert. For instance, we demonstrate below that in the context of NL provenance, some of the possible summarizations may naturally be translated to semantically meaningful sentences, while others may not.

- Another approach for addressing provenance size is to track and present only portions of it, in a **selective** manner. Depending on the intended use, full provenance information may be unnecessary: for instance, if provenance is tracked with the goal of allowing users to distinguish between candidate formal queries generated by the system, then showing provenance for a "representative" sample of query outputs may suffice. Here again, the choice of samples may be based on the user interaction: for instance, we may show provenance for the examples that the user has given (and is thus familiar with), to demonstrate the logic captured by the inferred query; additional examples that show further diverse facets of the query may be selected as well.

In the rest of this paper we overview our results achieved so far in this area. In Section 2 we overview our solution from [13, 14] on Natural Language Provenance. In Section 3 we highlight the potential of incorporating provenance to query-by-example frameworks [15, 16]. In Section 4 we discuss explaining to non-experts the results of Machine Learning models, and outline an approach for achieving that. We conclude in Section 5.

2 Natural Language Explanations

Developing Natural Language (NL) interfaces for database systems has been the focus of multiple lines of research (see e.g. [3, 25, 26, 40]). Users who view results computed by such systems may also be interested in the explanations for these results, i.e. *why* does each answer qualify to the query criteria. Such explanations could greatly enrich the answers, as well as provide means for the user to understand what is the underlying formal query compiled by the system and whether it matches the user intention.

As an example, consider the Microsoft Academic Search database (`http://academic.research.microsoft.com`) and consider the NL query in Figure 1a. A state-of-the-art NL query engine, `NaLIR` [26], is able to transform this NL query into the SQL query also shown (as a Conjunctive Query) in Figure 1b. When evaluated using a standard database engine, the query returns the expected list of organizations. However, the answers (organizations) in the query result lack *justification*, which in this case would include the authors affiliated with each organization and details of the papers they have published (their titles, their publication venues and publication years). Such additional information can lead to a richer answer than simply providing the names of organizations: it allows users to also see relevant details of the qualifying organizations. Provenance information is also valuable for validation of answers: a user who sees an organization name as an answer is likely to have a harder time validating that this organization qualifies as an answer, compared to a setting where she is shown the full details of publications. Our approach is to present *provenance information for answers of NL queries, again as sentences in natural language*. Continuing our running example, Figure 1c shows one of the answers outputted by our system in response to the NL query in Figure 1a.

This solution [14] consists of the following key components.

"Return the organization of authors who published papers in database conferences after 2005"

(a) NL Query

```
query(oname) :-
org(oid, oname),
conf(cid, cname),
pub(wid, cid, ptitle, pyear),
author(aid, aname, oid),
domainConf(cid, did),
domain(did, dname),
writes(aid, wid),
dname = 'Databases',
pyear > 2005
```

(b) CQ Q

"Tel Aviv University is the organization of Tova Milo who published 'OASSIS...' in SIGMOD in 2014"

(c) Answer For a Single Assignment

Figure 1: NL Query, CQ Q, and an example NL Answer

Provenance Tracking Based on the NL Query Structure A first key idea in our solution is to leverage the *NL query structure* in constructing NL provenance. In particular, we modify `NaLIR` so that we store exactly which parts of the NL query translate to which parts of the formal query. Then, we evaluate the formal query using a provenance-aware engine (we use `SelP` [17]), further modified so that it stores which parts of the query "contribute" to which parts of the provenance. By composing these two mappings (text-to-query-parts and query-parts-to-provenance) we infer which parts of the NL query text are related to which provenance parts. Finally, we use the latter information in an "inverse" manner, to translate the provenance to NL text. We show the construction by example and refer the reader to [14] for further details.

Example 1: Re-consider our running example query and consider the database in Table 1. The assignments to the query are represented in Figure 2 as a DNF expression. Each of the 6 clauses stands for a different assign-

```
(oname,TAU)∧(aname,Tova M.)∧(ptitle,OASSIS...)∧(cname,SIGMOD)∧(pyear,14')∨
(oname,TAU)∧(aname,Tova M.)∧(ptitle,Querying...)∧(cname,VLDB)∧(pyear,06')∨
(oname,TAU)∧(aname,Tova M.)∧ (ptitle,Monitoring..)∧(cname,VLDB)∧(pyear,07')∨
(oname,TAU)∧(aname,Slava N.)∧(ptitle,OASSIS...)∧(cname, SIGMOD)∧(pyear,14')∨
(oname,TAU)∧(aname,Tova M.)∧(ptitle,A sample...)∧(cname,SIGMOD)∧(pyear,14')∨
(oname,UPENN)∧(aname,Susan D.)∧(ptitle,OASSIS...)∧(cname,SIGMOD)∧(pyear,14')
```

Figure 2: Value-level Provenance

oid	oname
1	UPENN
2	TAU

Rel. org

aid	aname	oid
3	Susan D.	1
4	Tova M.	2
5	Slava N.	2

Rel. $author$

wid	cid	ptitle	pyear
6	10	“OASSIS...”	2014
7	10	“A sample...”	2014
8	11	“Monitoring...”	2007
9	11	“Querying...”	2006

Rel. pub

aid	wid
4	6
3	6
5	6
4	7
4	8
4	9

Rel. $writes$

cid	cname
10	SIGMOD
11	VLDB

Rel. $conf$

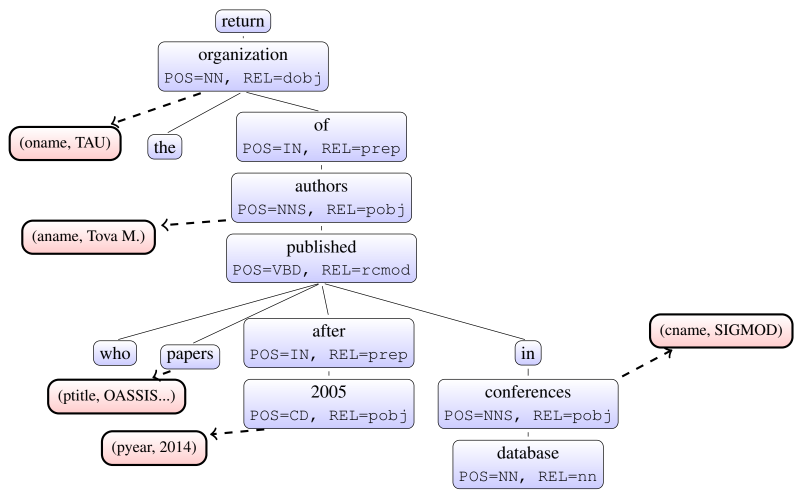
cid	did
10	18
11	18

Rel. $domainConf$

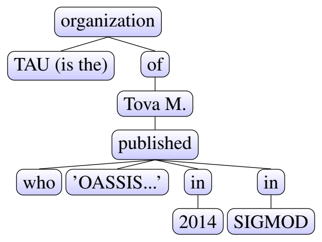
did	name
18	Databases

Rel. $domain$

Table 1: DB Instance



(a) Query Tree



(b) Answer Tree

Figure 3: Question and Answer Trees

ment, and the atoms are pairs of the form (var, val) so that var is assigned val in the particular assignment. We only record variables to which a query word was mapped (these are the relevant variables for formulating the answer). For instance, the node “organization” was mapped to the variable $oname$ which was assigned the values “TAU” and “UPENN”. If we were to replace the value in the organization node by the value “TAU” mapped to it, the word “organization” would not appear in the answer although it is needed to produce a coherent sentence such as the one depicted in Figure 1c. In the absence of this word, it is unclear how to connect “Tova M.” and “TAU”. To this end, we rely on the information encoded in the dependency tree to convert it to an answer tree based on the relationships and part-of-speech of the words in the sentence. Finally, the conversion of the answer tree in Figure 3b to a sentence is done by replacing the words of the NL query with the values mapped to them, e.g., the word “authors” in the NL query (Figure 1a) is replaced by “Tova M.” and the word “papers” is replaced by “OASSIS...”. The word “organization” is not replaced (as it remains in the answer tree) but rather the words “TAU is the” are added prior to it, using the part-of-speech and relationship of the word. Completing this process, we obtain the answer shown in Figure 1c.

Factorization A second key idea is related to the provenance size. In typical scenarios, a single answer may have multiple explanations (multiple authors, papers, venues and years in our example). A naïve solution is

```
[TAU] ·
     ([Tova M.] ·
        ([VLDB] ·
            ([2006] · [Querying...]
A  B      + [2007] · [Monitoring...]))            B  A
        + [SIGMOD] · [2014] ·
              ([OASSIS...] + [A Sample...]))
     + [Slava N.] · [OASSIS...] · [SIGMOD] · [2014])
+ [UPENN] · [Susan D.] · [OASSIS...] ·
  [SIGMOD] · [2014]
```

(a) f_1

```
[TAU] ·
    ([SIGMOD] · [2014] ·
        ([OASSIS...] ·
            ([Tova M.] + [Slava N.]))
      + [Tova M.] · [A Sample...])
  + [VLDB] · [Tova M.] ·
        ([2006] · [Querying...]
      + [2007] · [Monitoring...])
+ [UPENN] · [Susan D.] ·  [OASSIS...] ·
  [SIGMOD] · [2014]
```

(b) f_2

Figure 4: Provenance Factorizations

to formulate and present a separate sentence corresponding to each explanation. The result will however be, in many cases, very long and repetitive. As observed already in previous work [11, 31], different assignments (explanations) may have significant parts in common, and this can be leveraged in a *factorization* that groups together multiple occurrences. In our example, we can e.g. factorize explanations based on author, paper name, conference name or year. Importantly, we impose a novel constraint on the factorizations that we look for (which we call *compatibility*), intuitively capturing that their structure is consistent with a partial order defined by the parse tree of the question. This constraint is helpful in translating the factorization back to an NL answer whose structure is similar to that of the question; again, we refer the reader to [14] for details and only show an example.

Example 2: Re-consider the provenance expression in Figure 2. Two possible factorizations are shown in Figure 4, keeping only the values and omitting the variable names for brevity (ignore the A,B brackets for now). In both cases, the idea is to avoid repetitions in the provenance expression, by taking out a common factor that appears in multiple summands. Different choices of which common factor to take out lead to different factorizations.

Consider factorization f_2 from Figure 4. "TAU" should be at the beginning of the sentence and followed by the conference names "SIGMOD" and "VLDB". The second and third layers of f_2 are composed of author names ("Tova M.", "Slava N."), paper titles ("OASSIS", "A sample...", "Monitoring...") and publication years (2007, 2014). Changing the original order of the words such that the conference name "SIGMOD" and the publication year "2014" will appear before "Tova M." breaks the sentence structure in a sense. It is unclear how to algorithmically translate this factorization into an NL answer, since we need to patch the broken structure by adding connecting phrases. One hypothetical option of patching f_2 and transforming it into an NL answer is depicted below. The bold parts of the sentence are not part of the factorization and it is not clear how to generate and incorporate them into the sentence algorithmically. Even if we could do so, it appears that the resulting sentence would be quite convoluted:

TAU is the organization **of authors who published in**
SIGMOD 2014
'OASSIS...' **which was published by**
Tova M. and Slava N.
and Tova M. published 'A sample...'
and Tova M. published in VLDB
'Querying...' in 2014
and 'Monitoring...' in 2007.
UPENN is the organization of Susan D. who published
'OASSIS...' in SIGMOD in 2014

Observe that the resulting sentence is not clear, even though it was obtained from a shorter factorization f_2; the intuitive reason is that the structure of f_2 is very different from that of the NL question, and thus is not guaranteed to admit a structure that is coherent in Natural Language. Interestingly, the sentence we would obtain in such a way also has an edit distance from the question [18] that is shorter than that of our answer, demonstrating that edit distance is not an adequate measure here.

Instead, we propose using the structure of the original NL query to construct a factorization that, while reducing the size of the original expression, maintains the hierarchy of the words in the dependency tree. f_1 in Figure 4 maintains the word hierarchy implied by the dependency tree (Figure 3a), and thus can be converted into an answer tree and the sentence:

```
TAU is the organization of
    Tova M. who published
        in VLDB
           'Querying...' in 2006 and
           'Monitoring...' in 2007
        and in SIGMOD in 2014
            'OASSIS...' and  'A sample...'
    and Slava N. who published
        'OASSIS...' in SIGMOD in 2014.
UPENN is the organization of Susan D. who published
'OASSIS...' in SIGMOD in 2014.
```

Summarization We *summarize* explanations by replacing details of different parts of the explanation by their synopsis, e.g. presenting only the number of papers published by each author, the number of authors, or the overall number of papers published by authors of each organization. Such summarizations incur by nature a loss of information but are typically much more concise and easier for users to follow. Here again, while provenance summarization has been studied before (e.g. [2, 34]), the desiderata of a summarization needed for NL sentence generation are different, rendering previous solutions inapplicable here. We observe a tight correspondence between factorization and summarization: every factorization gives rise to multiple possible summarizations, each obtained by counting the number of sub-explanations that are "factorized together".

```
(A) [TAU] · Size([Tova M.],[Slava N.]) · Size([VLDB],[SIGMOD]) ·
      Size([Querying...],[Monitoring...],
        [OASSIS...],[A Sample...]) · Range([2006],[2007],[2014])
(B) [TAU]·(
      [Tova M.] ·
       Size([VLDB],[SIGMOD]) ·
       Size([Querying...],[Monitoring...],
        [OASSIS...],[A Sample...]) · Range([2006],[2007],[2014])
      [Slava N.] · [OASSIS...] · [SIGMOD] · [2014])
```

(a) Summarized Factorizations

(A) *"TAU is the organization of 2 authors who published 4 papers in 2 conferences in 2006 - 2014"*

(B) *"TAU is the organization of Tova M. who published 4 papers in 2 conferences in 2006 - 2014 and Slava N. who published 'OASSIS...' in SIGMOD in 2014"*

(b) Summarized Sentences

Example 3: Reconsider Example 2; if there are many authors from TAU then even the compact representation of the result could be very long. In such cases we need to summarize the provenance in some way that will preserve the "essence" of all assignments without actually specifying them, for instance by providing only the number of authors/papers for each institution.

Re-consider the factorization f_1 from Figure 4. We can summarize it in multiple levels: the highest level of authors (summarization "A"), or the level of papers for each particular author (summarization "B"), or the level of conferences, etc. Note that if we choose to summarize at some level, we must summarize all levels below it (e.g. if we summarize for "Tova M." at the level of conferences, we cannot specify the papers titles and publication years).

Figure 5a presents the summarizations of sub-trees for the "TAU" answer, where "size" is a summarization operator that counts the number of distinct values and "range" is an operator over numeric values, summarizing them as their range. The summarized factorizations are further converted to NL sentences which are shown in Figure 5b. Summarizing at a higher level results in a shorter but less detailed summarization.

Our work in [14] is restricted to Conjunctive Queries, and extensions to more expressive forms of queries are the subject of an ongoing investigation. We next turn to briefly describe another aspect of explanations, namely explaining non-answers.

Why-not Explanations Our work so far has focused on explaining query answers in NL; an equally important type of insight that users may wish to gain concerns expected answers that do not appear in the query result set. Such insight may be useful for identifying errors in the inferred query as well as omissions in the database.

Provenance for non-answers has been extensively explored (e.g. [5–7, 10, 22]), but similarly to the positive case, a direct use of such models is unsuitable for non-experts. For instance, the work of [10] defines why-not provenance in terms of the query operators; the non-expert may be unfamiliar with formal query languages. Similarly, the work of [5] defines a notion of provenance polynomials, which may be too complex for presentation to non-experts, etc.

Again, a plausible approach (that we follow in our work-in-progress) is to leverage the structure of the NL query for presentation of why-not provenance. For example, if we leverage the work of [10] to identify "picky" operators in the query (i.e. parts of the query that were responsible for the tuple omission), then in some cases we may further track the words in the NL query that have led to the generation of each operator. Then, we can present why-not provenance by *highlighting* these words. Another approach is to suggest various alternative questions which are in the same spirit of the user's question, with a strike-through line over the word mapped to the picky operator.

Example 4: Reconsider our running example and assume that all "Hogwarts" authors who published in database conferences have done so before 2005, so all tuples that contain "Hogwarts" do not qualify due to the selection operator $pyear > 2005$. Instead of showing the selection operator itself, we can highlight the words that were mapped to this operator: "after 2005". This will give the user an idea of what has to change in the query in order to get "Hogwarts" in the result set. Another scenario may be that no author associated to "Hogwarts" has published in a database conference. In that case, we could highlight the words "database conferences" in the original NL query. Alternatively, we could suggest the question "Return the organization of authors who published papers in database conferences after 2005".

These approaches work fairly well for operators that have words in the NL query directly mapped to them. This is not always the case. For instance, for the query of Figure 1b, the triple join operator used to connect table *author* with the table *pub* through the connecting table *writes* is not specifically mentioned in the NL query, and was inferred automatically by the NL interface. Handling such cases requires further solutions whose investigation is left for future work.

3 Explanations for Query-By-Example Frameworks

Query-by-example interfaces have been extensively studied [8, 30, 44] to allow non-expert users to query a database without requiring extensive technical knowledge. These interfaces allow users to specify output examples of the desired query and then try to automatically infer it. The challenge is naturally that the size of the search space, i.e. the number of queries that are consistent with the given examples, is typically very large. Existing solutions aim at addressing this via an interactive process; for instance, users are shown at each step positive and negative examples based on generated candidate queries, and their feedback is used to focus on a subset of these queries [8]. We claim that provenance may be very helpful in narrowing the search space in query-by-example frameworks, and may be used for this purpose in two different manners, as follows.

Query-by-provenance In [15, 16] we have proposed a framework for the inference of queries from output examples *and their explanations*. The idea is that explanations that are attached to examples may be viewed as their provenance with respect to the (unknown) ground truth query (we show in [16] an interface that allows non-experts to provide explanations).

Example 5: Consider a database of academic authors and their papers, and also consider an intended query that should return all pairs of authors who wrote a common paper. Examples of the output, i.e. pairs of authors, may share characteristics such as country of residence. This in turn may lead to the inference of a query that is very different from the intended one. In contrast, if the user also provides a co-authored paper as an explanation, this may potentially greatly restrict the search space.

This leads to three concrete questions: (1) how do we formalize the consistency of a query with examples and explanations? (2) How many different queries are consistent with a set of examples and their explanations/ (3) What is the complexity of finding a consistent query and/or enumerating all of them? In [15] we provide initial answers to these questions. We formally define a generic notion of query-by-provenance that applies to any *provenance semiring* [21]. We show that the answers to the questions (2) and (3) above greatly depend on the choice of semiring. For instance, the number of queries for a given provenance polynomial (NX) may be exponential in the sum of the arities of the relations participating in a monomial. On the other hand, if the provenance is given in the Why(X) semiring (corresponding to the why-provenance of [9]), there may be infinitely many consistent queries. Yet, we show in [15] a "small-world" property, namely that there exists a consistent query whose size is polynomial in the number of attributes in the output example, the number of distinct relation names in the provenance, and the largest cardinality of a set in the provenance.

Using Provenance to Choose Between Candidate Queries As mentioned above, even in the presence of explanations, there may be a large number of consistent candidate queries, and further user feedback is required to choose a "correct" one, i.e. one that matches their intention. Provenance may be highly useful in this respect as well; it is common practice to procure feedback for possible answers that would allow differentiating between candidate queries (i.e. ask about the correctness of answers of one query that are non-answers of another one). But it is not always trivial for users to state whether or not a proposed result should appear in the output; having the system explain the rationale for computing this answer (i.e. present its provenance with respect to the candidate query being considered), can be highly useful.

Example 6: Consider an ontology of authors and papers, and a query asking for all authors with Erdős number 2. Examples for the query output, i.e. example authors, will likely be un-indicative of the actual intended query, since the authors may share many other characteristics. The *provenance* of an example author with respect to the intended query will be one of her co-authorship paths to Erdős of length 2, which reveals much more information on the inferred query and allows for its validation by the user.

Following our general principle, both the presentation of provenance and the procurement of explanations take a form that follows the standard interaction of the user with the system. For instance, users attach explanations to examples that they would give regardless of provenance, and so they can use the rationale they have already employed in choosing the examples, to form the explanation. Provenance for a sample answer produced by a candidate query can be shown by e.g. the relevant path in an RDF setting (in our ongoing work), and the candidate answers themselves should be carefully chosen to "resemble" the user-provided examples. Employing these principles allows for bridging the gap between formal provenance models and non-expert users.

4 Machine Learning Results Explanation

The two example domains shown above are ones where the underlying models are based on some query language for which provenance may be tracked. Can we leverage the developed ideas for additional settings? In this section we briefly describe our work-in-progress, exploring the possibility of generating explanations for non-experts that shed light on the result of Machine Learning (ML) models.

Despite its wide adoption, one drawback of ML is the complexity level of its models which makes it difficult to understand the reasons for a given result. Such understanding is crucial to assess (and potentially improve) the quality of a model and identify errors in its output. Recognizing this need, there is a large body of work on understandable Machine Learning. One approach in this respect is to explain the model as a whole, e.g. by approximating its specification through a simpler model, such as rules (see e.g. [23]). A second approach is not to explain the model as a whole but rather to present the reasoning underlying individual predictions [35]. This latter approach is close in spirit to provenance, and will be the focus of our short discussion.

We start with a simple model of explanations based on minimal changes, and gradually refine it.

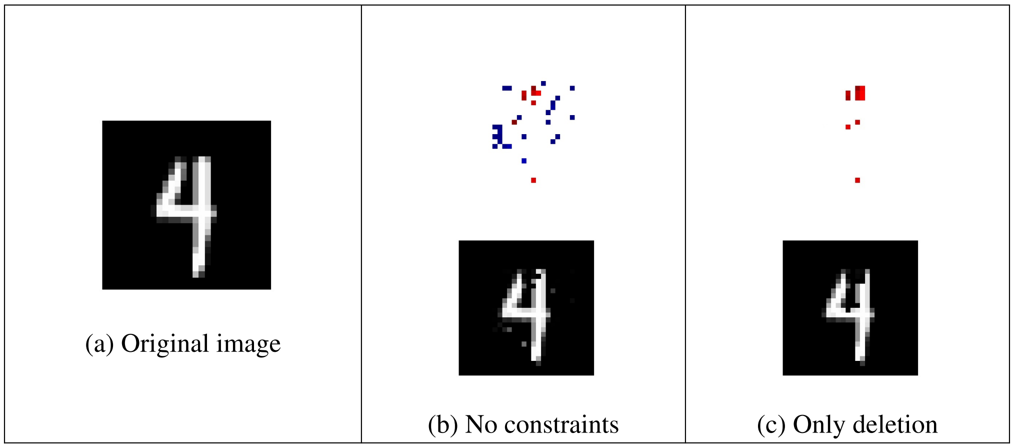
Example 7: Consider a simple linear regression model for loan applications, where the model takes into account two binary parameters, $income$ and $debt$. For each loan request, the model will return a score based on the following formula $y = 0.5 + \frac{income}{2} - \frac{debt}{2}$. Assume that loans are approved only if they have model score of 1. Further consider an applicant with $x = \{income : 1, debt : 1\}$. The model returns the score of 0.5, and the application is consequently denied. Naturally, the applicant may be interested in understanding the denial reason, and may ask e.g. **(1)** what factors have influenced the decision? and **(2)** what changes in the application may lead to its acceptance next time?

A possible explanation model, in the spirit of why-not provenance, is that of *minimal changes* to the data that would change the model decision. In this example, to answer the second question, we may look for minimal changes that lead to loan acceptance (in our example, this involves changing the profile to $x = \{income : 1, debt : 0\}$). This also partly answers the first question, as we understand that the $debt$ parameter has a major influence on the model result. We may further look for changes that would cause to model the *decrease* its confidence in the application. By doing so, we would find out that changing the income value such that $x = \{income : 0, debt : 1\}$ would reduce the model score to 0. Combined, we observe that both $income$ and $debt$ parameters have a significant influence on the model decision.

In the context of database queries the related problems of explanations using minimal changes has been studied in [27, 28, 43]. However, in the domain of machine learning the subject of minimal modifications in the input that change the model output has been studied in a different context than that of explanations, namely adversarial examples (see e.g. [20, 41]) where the goal is to "fool" the model. For explanations, we may need a more refined model: in our example, it does not make sense to recommend the applicant to change features she can not control such as her age. Hence, we may be interested in the minimal changes that lead to the satisfaction of some pre-specified constraints.

Example 8: Figure 6a presents an image of handwritten 4 that was tagged as 9 by a simple Random Forest model which we have trained using the MNIST data set. Two examples for minimal changes to the image that lead to a decision of "4" are shown in Figures 6b and 6c. In Figure 6b there were no constraints on the allowed modifications, and we can see that many pixels have changed. Some of them are not even close to the digit, and do not serve as a convincing explanation. In Figure 6c we present the result of the minimal sequence of *pixel deletions* (the constraint is that only deletions are allowed) that lead to the tag "4". We can see that most of the deletion were of pixels in the top of the image, providing further insight.

Can we gain further insights by *summarizing* explanations for multiple instances that share some property? Our initial results indicate that this is the case.

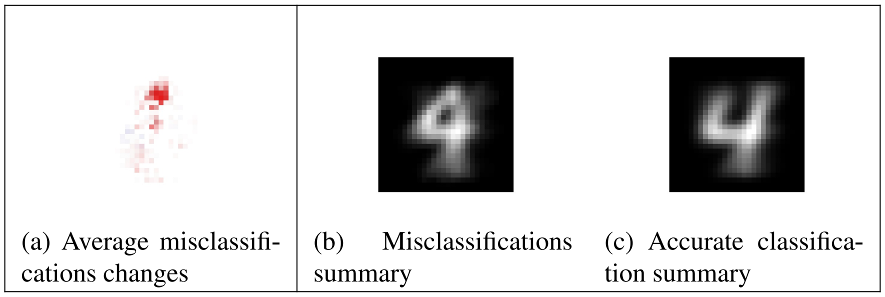

(a) Original image

(b) No constraints

(c) Only deletion

Figure 6: Change constraints

Example 9: In Figure 7a we depict the *average* changes over multiple instances that were tagged "9" but for which the ground truth tag is "4". Now we can clearly see that deleting the top pixels of the digit is a general "fix", and so their existence is a valid explanation for the misclassification. We can verify the correctness of this insight by Figures 7b and 7c that present the average of 4 images that were tagged as 9 and 4 respectively by our model. Observe that the upper part of the digits classified as 9 is closed and that of digits classified as 4 is open. The gained insight is that the model learned how to recognize images of 4 with open top, but images of 4 with close top are misclassified as 9.


(a) Average misclassifications changes

(b) Misclassifications summary

(c) Accurate classification summary

Figure 7: Summarized explanations

5 Conclusion

We have discussed the usefulness of provenance presentation to non-experts, and have demonstrated it in three contexts: NL database interfaces, query-by-example frameworks, and Machine Learning. We have further described preliminary solutions for transforming provenance into user-friendly explanations, highlighting some common principles. Further scaling up the solutions, applying them to additional frameworks and applications, and improving the clarity of explanations are all subjects of our ongoing investigation.

Acknowledgments This research has been partially supported by the Israeli Science Foundation (ISF), by ICRC - The Blavatnik Interdisciplinary Cyber Research Center and by Intel. The different contributions of Nave Frost and Amir Gilad are each part of their respective PhD research conducted at Tel Aviv University.

References

[1] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Playful query specification with dataplay. *PVLDB*, 5(12), 2012.

[2] E. Ainy, P. Bourhis, S. B. Davidson, D. Deutch, and T. Milo. Approximated summarization of data provenance. In *CIKM*, 2015.

[3] Y. Amsterdamer, A. Kukliansky, and T. Milo. A natural language interface for querying general and individual knowledge. *PVLDB*, 8(12), 2015.

[4] N. Bakibayev, D. Olteanu, and J. Zavodny. FDB: A query engine for factorised relational databases. *PVLDB*, 5(11), 2012.

[5] N. Bidoit, M. Herschel, and A. Tzompanaki. Efficient computation of polynomial explanations of why-not questions. In *CIKM*, 2015.

[6] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *TaPP*, 2014.

[7] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based why-not provenance with nedexplain. In *EDBT*, 2014.

[8] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive inference of join queries. In *EDBT*, 2014.

[9] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[10] A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, 2009.

[11] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, 2008.

[12] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. ICDT, 2010.

[13] D. Deutch, N. Frost, and A. Gilad. Nlprov: Natural language provenance (demo). *PVLDB*, 9(13), 2016.

[14] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. *PVLDB*, 10(5), 2017.

[15] D. Deutch and A. Gilad. Learning queries from examples and their explanations. *CoRR*, 2016.

[16] D. Deutch and A. Gilad. Qplain: Query by explanation (demo). In *ICDE*, 2016.

[17] D. Deutch, A. Gilad, and Y. Moskovitch. Selective provenance for datalog programs using top-k queries. *PVLDB*, 8(12), 2015.

[18] M. Emms. Variants of tree similarity in a question answering task. In *LD*, 2006.

[19] I. Foster, J. Vockler, M. Wilde, and A. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *SSDBM*, 2002.

[20] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, 2014.

[21] T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[22] M. Herschel. A hybrid approach to answering why-not questions on relational query results. *J. Data and Information Quality*, 5(3), 2015.

[23] J. Huysmans, B. Baesens, and J. Vanthienen. Using rule extraction to improve the comprehensibility of predictive models. 2006.

[24] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, 2010.

[25] D. Küpper, M. Storbel, and D. Rösner. Nauda: A cooperative natural language interface to relational databases. SIGMOD, 1993.

[26] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1), 2014.

[27] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. *IEEE Data Eng. Bull.*, 2010.

[28] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB Endowment*, 2010.

[29] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. In *VLDB*, 2014.

[30] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5), 2014.

[31] D. Olteanu and J. Závodný. Factorised representations of query results: Size bounds and readability. ICDT, 2012.

[32] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, 2003.

[33] F. Psallidas, B. Ding, K. Chakrabarti, and S. Chaudhuri. S4: Top-k spreadsheet-style search for query discovery. SIGMOD, 2015.

[34] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *PVLDB*, 1(1), 2008.

[35] M. T. Ribeiro, S. Singh, and C. Guestrin. “why should i trust you?”: Explaining the predictions of any classifier. KDD, 2016.

[36] T. Sellam and M. L. Kersten. Meet charles, big data query advisor. CIDR, 2013.

[37] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *SIGMOD*, 2014.

[38] Y. L. Simmhan, B. Plale, and D. Gannon. Karma2: Provenance management for data-driven workflows. *Int. J. Web Service Res.*, 2008.

[39] D. Song, F. Schilder, and C. Smiley. Natural language question answering and analytics for diverse and interlinked datasets. NAACL, 2015.

[40] D. Song, F. Schilder, C. Smiley, C. Brew, T. Zielund, H. Bretz, R. Martin, C. Dale, J. Duprey, T. Miller, and J. Harrison. TR discover: A natural language interface for querying and analyzing interlinked datasets. In *ISWC*, 2015.

[41] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *CoRR*, 2013.

[42] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, 2009.

[43] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proceedings of the VLDB Endowment*, 2013.

[44] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.

[45] M. M. Zloof. Query by example. In *AFIPS NCC*, 1975.

Provenance and the Different Flavors of Computational Reproducibility

Juliana Freire
New York University
juliana.freire@nyu.edu

Fernando Chirigati
New York University
fchirigati@nyu.edu

Abstract

While reproducibility has been a requirement in natural sciences for centuries, computational experiments have not followed the same standard. Often, there is insufficient information to reproduce computational results described in publications, and in the recent past, this has led to many retractions. Although scientists are aware of the numerous benefits of reproducibility, the perceived amount of work to make results reproducible is a significant disincentive. Fortunately, much of the information needed to reproduce an experiment can be obtained by systematically capturing its provenance. In this paper, we give an overview of different types of provenance and how they can be used to support reproducibility. We also describe a representative set of provenance tools and approaches that make it easy to create reproducible experiments.

1 Introduction

The need to reproduce experiments to verify and extend them is not new in science. Revisiting and reusing past results – or as Newton once said, "standing on the shoulders of giants" – is the standard paradigm of all sciences. Unfortunately, achieving reproducibility has proved elusive for computational experiments, which, due to the explosion in the volume of available data and widely accessible computing infrastructure, have become an integral component of science in many different domains.

Scientific papers published in conferences and journals present a large number of tables, plots, and beautiful pictures that summarize the obtained results, but that loosely describe the steps taken to derive them [16,38]. Not only can the methods and the implementation be complex, but their configuration may require setting myriad parameters. Consequently, reproducing the results from scratch is both time-consuming and error-prone at best, and sometimes impossible.

Reproducibility of computational experiments across platforms and time brings a range of benefits to science. First, reproducibility enables reviewers to test the outcomes presented in papers. This is specially important given the growing concern that many spurious research findings are published in respected venues [5,12,29], which is reflected in the increasing number of paper retractions [44, 58]. Second, it allows new methods to be objectively compared against methods presented in reproducible publications. Third, researchers are able to build on top of previous work directly. Last but not least, recent studies indicate that reproducibility increases impact, visibility, and research quality [3, 7, 26, 36, 50, 59] and helps defeat self-deception [46].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

While many scientists recognize the importance of reproducibility, they are often held back by the complexities involved in putting it into practice. They must describe and encapsulate the entire experiment, i.e., the set of steps followed to obtain a result, including data, parameters, source code, dependencies, and environment, so that the results can be properly verified and analyzed. If the experiment is not systematically documented and made reproducible from the start, it may be difficult and time-consuming to retrospectively track all the necessary components, and important aspects may be mistakenly omitted. As an example, some numerical models, if not fully described, may lead to different implementations that are mathematically equivalent, but numerically disparate, thus hampering reproducibility [14]. Even when a complete description is available, others may have difficulties to reproduce the results: there may be no instructions about how to execute the code and explore it further; the experiment may not run on a certain operating system; there may be missing libraries; library versions may be different; and several issues may arise while trying to install all the required chains of dependencies.

Fortunately, much of the information needed to reproduce an experiment can be obtained by capturing its *provenance*. For a given experiment, computational provenance provides information about how and where that experiment was carried out, what input data was used, and which outputs were produced. In other words, provenance *embodies* the association between computation and results [17]: it helps determine both the data and the sequence of steps that generated the findings, which is essential to make results reproducible. Note, however, that different *types of provenance* can be captured, and these enable different *levels of reproducibility*. For instance, some systems, such as VisTrails [19] and Kepler [39], capture workflow provenance, i.e., the dataflow of an experiment, including its main programs and all the data used and derived. But these systems do not detect the library dependencies for the different programs. As a result, they only attain reproducibility given that the computational environment is unchanged (e.g., machine and version of dependencies are the same). Other systems, such as ReproZip [51], are able to capture provenance at the operating system level, including library dependencies, which allows experiments to be reproduced even in different machines. Nonetheless, different from workflow systems, ReproZip may not capture the dataflow in an human-interpretable format, which may make it harder to extend and modify the original dataflow for other purposes.

In this paper, we provide an overview of the different types of provenance and how they influence reproducibility. Our main goal is to help researchers understand how the different types of provenance affect the level of reproducibility and their corresponding trade-offs, to guide them in selecting a suitable approach for their experiments. In Section 2, we start by defining *computational reproducibility*, and use this definition to detail the different levels of reproducibility one can achieve. Next, we present all the different types of provenance and how they related to reproducibility in Section 3, where we also describe tools that can be used to capture these different types. We conclude in Section 4, where we discuss challenges and open problems.

2 What is Reproducibility?

There are different definitions for the term *reproducibility*, which have been used inconsistently across scientific disciplines [2]. In this paper, we focus on the *computational aspects* of reproducibility and introduce definitions that capture the necessary components to implement computational reproducibility in practice.

2.1 Computational Reproducibility

To understand what is needed to reproduce an experiment, we must first define what a *scientific experiment* is. We borrow the definition from Rokem and Chirigati [52]:

Definition 1 (Scientific Experiment): A scientific experiment, or simply experiment, is any procedure carried out to validate or refute a hypothesis, which involves the use of computational assets, including computer programs and digital data that is consumed (input data) and produced (output data).

Such procedure can be represented as a *dataflow*: a sequence of steps that are connected by the flow of data, where the output data of a step is used as input data for the following step. In this case, a step is represented by a computer program or a sequence of programs, and it transforms the data it consumes as part of the procedure. Below, we define what it means for an experiment to be reproducible [17].

Definition 2 (Reproducible Experiment): An experiment composed by a sequence of steps S that has been developed at time T, on environment (hardware and operating system) E, and on data D is *reproducible* if it can be executed with a sequence of steps S' (different or the same as S) at time $T' \geq T$, on environment E' (different or the same as E), and on data D' (different or the same as D) with consistent results.

Consistency here implies that the results can validate the original claims of the research, i.e., the same conclusions derived from the original results can be obtained from the new results. Clearly, this depends on the problem being addressed and the overall goal. For instance, it is unlikely that an experiment that compares the performance of two database systems will produce the exact running times across runs, in particular, if different machines are used (i.e., $E' \neq E$). However, if the new numbers reflect the same trends originally observed and validated, then we say that the results are consistent, and therefore, reproducible.

Note that our definition includes both *exact reproducibility* and *approximate reproducibility* [52]. Exact reproducibility, also known as *repeatability*, entails reproducing the exact same results (meaning, the exact same numbers) as originally published, which requires having $S' = S$, $E' = E$, and $D' = D$. Approximate reproducibility, on the other hand, involves producing results that are similar to (but not that same as) the original ones, which entails having different data, a modified sequence of steps, a different environment, or a combination of these. In this case, the reproduced results may not necessarily be the same, but they must be consistent with the original results. For experiments that are intrinsically difficult to replicate (e.g., experiments that require a specific hardware), guaranteeing approximate reproducibility is essential.

The PRIMAD Model. The PRIMAD model [18] extends our definition of reproducibility to include non-computational aspects as well. It provides a flexible model to define reproducibility: different levels (or modes) of reproducibility can be defined by modifying a set of variables while reproducing an experiment. We consider the following variables:

- **(P) Platform:** the computational environment, including operating system, hardware architecture, and library dependencies.
- **(R) Research Objectives:** the main goal or purpose of the experiment, i.e., the problem that the experiment is trying to address.
- **(I) Implementation:** how the experiment and its corresponding sequence of steps is implemented (e.g., source code and binaries).
- **(M) Methods:** the methods and algorithms that the experiment implements and uses to achieve the research goals.
- **(A) Actors:** the main users of the experiment.
- **(D) Data:** the input data files, intermediate data, and parameters for the experiment.

These variables are used to describe which aspects of the experiment can be changed while still attaining reproducible results. The conditions under which the experiment is reproducible are defined by qualifying the different variables: we tag a variable X with the prime symbol to indicate that X can be changed and the experiment is still reproducible. If untagged variables are changed, reproducibility cannot be guaranteed. For instance, if an experiment is claimed to be $P'RIMA'D'$, it means that, if researchers, who are not the original

Level	Data		Platform	Implementation	
	Parameters	Data Files		Binaries	Source Code
Repeatable	–	–	–	–	–
Re-runnable	X	X	–	–	–
Portable	–	–	X	–	–
Extendable	–	–	–	X	X
Modifiable	–	–	–	–	X

Table 2: Levels of reproducibility based on the different variables. The symbol "X" denotes a change in the corresponding variable, while "–" denotes no change from the original setting. Adapted from [18].

authors of the experiment (A'), reproduce the experiment with different data (D') on a different platform (P'), they will obtain results that are consistent with the ones originally published.

Note that P, I, and D are equivalent to E, S, and D, respectively, from Definition 2. The other variables from the PRIMAD model correspond to the non-computational aspects of reproducibility. While we do not consider these aspects in this paper, we note that they are vastly important and useful to complement the computational components.

Additional Dimensions of Reproducibility. Besides the PRIMAD variables, there are other dimensions that are important for qualifying the level of reproducibility of an experiment. One of these dimensions is *coverage* [17], which takes into account how much of the experiment can be reproduced, i.e., if the experiment can be partially or fully reproduced. Many experiments cannot be fully reproduced, e.g., experiments that rely on data derived by third-party Web services or special hardware. But such experiments can, sometimes, be partially reproduced. For example, if an experiment uses data that is derived by special, proprietary hardware, the data derivation may not be reproducible. However, the downstream analyses that use these data may be reproduced by others if the data is made available.

Another important dimension is *transparency*, which considers *how much information* is made available for an experiment. This dimension affects the variables in different ways. For instance, in terms of implementation, one can provide the original binaries used in the experiment. While this allows the results to be reproduced, it limits reusability: these binaries can only be executed in a compatible platform, and the implementation cannot be modified. On the other hand, if the source code is provided, the implementation can be better inspected and reused. In terms of data, if the data cleaning process is made available, in addition to the final, cleaned input files, it is easier to clean other datasets and explore how the experiment behaves with these.

Finally, *longevity* relates to the ability to reproduce experiments (long) after they were created. Supporting longevity is challenging because software environments evolve, i.e., libraries, operating systems, and data used change over time.

2.2 Levels of Reproducibility

To assess the reproducibility of an experiment, we need to understand which of its different components are made available. In what follows, we categorize the different *levels of reproducibility* based on Definition 2 and the PRIMAD model. Table 2 summarizes the different levels and the corresponding variables that can be changed while attaining that level of reproducibility.

Repeatable. An experiment is *repeatable* if the same results can be re-generated within the same computational environment, with no changes to code or data, i.e., if the results can be repeated. This is the lowest level of reproducibility (exact reproducibility), and can be used to determine if the experiment is deterministically consistent. Note that some experiments may have non-deterministic steps (e.g., a random number generator, or third-party services that are accessed remotely). In such cases, achieving the exact same results may not be possible, but they must be consistent with the original ones, i.e., the same conclusions must be reached.

Re-runnable. We say an experiment is *re-runnable* if we can vary the input data (either the data files or the input parameters) and still get results that are consistent. Note that this allows one to determine how the experiment behaves and how robust the results are for different inputs. For instance, if the input data changes significantly and the results are still consistent, the experiment may have a broader scope worth investigating. This also allows one to evaluate whether the data originally used is representative for a given domain.

Portable. An experiment is *portable* if it can be re-executed on platforms that are different from the platform where the original results were generated. In this case, the results may be reproduced on similar environments (i.e., compatible operating system but different machines), or on different environments (i.e., different operating systems and machines). The level of portability will be higher if the experiments can be run on completely different environments. Note that, by changing platforms, library dependencies may also change, and this also affects reproducibility. For instance, the version of a software library may be available on Ubuntu (the original environment) but not on Windows (the new environment).

Extendable. An experiment is *extendable* if we can reuse its original dataflow and structure, i.e., its original sequence of steps, for other experiments. Examples include integrating the original pipeline into an existing dataflow, or even *extending* the original one to include new pre- or post-processing steps, e.g., performing additional data cleaning to the input. This is possible by having access to the implementation of the experiment, either binaries or source code. Note, however, that the experiment must be portable so that the binaries can be run in different platforms.

Modifiable. We say an experiment is *modifiable* if we can change its implementation for reuse purposes, and this is achievable by having the source code of the experiment. It is worth noting that changing the implementation also allows others to verify the correctness of the original implementation. In addition, if the experiment is not yet portable, others may modify the source code to make it runnable in different platforms.

3 Provenance for Reproducibility

The variables that correspond to the computational aspects of reproducibility, i.e., P, I, and D, can be systematically captured by keeping track of the experiment's *provenance*. Provenance refers to the record trail that accounts for the origin of a piece of data [20]. These trails enable scientists to: (i) obtain insights into the chain of reasoning used in the production of a result; (ii) verify the sequence of steps that led to the experiment findings (by capturing I – implementation); (iii) identify the inputs to an experiment and where they came from (by capturing D – data); and (iv) determine where the experiment was originally executed (by capturing P – platform).

Provenance is *essential* for attaining reproducibility [10, 11]. But different *types of provenance* can be captured, and these impact reproducibility in various ways. Thus, to choose the most appropriate provenance capture method for a scientific experiment, it is important to understand how each of these types influence the different levels of reproducibility. Below, we describe different types of provenance and how they affect reproducibility. We also present a set of tools that support automatic provenance capture.

3.1 Database Provenance

Database provenance provides a description of the derivation of a piece of data that results from executing a database query against a source database [8]. It is fine-grained, in the sense that provenance is captured for individual data items: it can be used to determine which parts of the source dataset were used to generate a piece of data in the resulting dataset. Often, database provenance is captured by reasoning about the algebraic form of the query and the underlying data model of both the source and resulting datasets. Different notions of database provenance have been defined, notably *why*-, *where*-, and *how-provenance* [9]. Given a tuple t in the result of a query, why-provenance describes all the source tuples that contributed to t or helped produce t.

Where-provenance determines where the source tuples were copied from (e.g., which cell of the relation the data comes from). How-provenance, as the name suggests, describes how the source tuples derived t (e.g., how many times each source tuple contributed to t and how they were combined).

Since databases operate in a stateful mode, database provenace is important for reproducibility. Every time a transaction commits, there is a new state that reflects the changes applied within the transaction; executing a query over different database states is likely to lead to different results.

Database provenance is *descriptive*: it provides an explanation for the derivation of results. Note that this type of provenance only captures variable D: no information about the computational environment or the database management system is available.

Levels of Reproducibility. Because database provenance captures the different states of a database (D), it can help make an experiment *repeatable*. How-provenance can make an experiment *re-runnable* by re-executing the operations with different data and assessing whether the new results are consistent with the original ones.

Tools. Systems that capture database provenance include Trio [4], Orchestra [24], Perm [22], GProM [1], and ProQL [32]. Trio is a database management system based on an extended relational algebra called ULDB, which supports uncertainty and provenance. The source data is annotated with probabilities, and the captured provenance is used to compute probabilities associated with the derived results. Orchestra is a collaborative sharing system that uses how-provenance to filter data based on user-specified trust conditions (provenance is used for data quality and reliability). In Perm, different notions of database provenance, including why-, where-, and how-provenance, are supported. Perm is implemented as a modified PostgreSQL engine, where data and its provenance are represented together in a single relation; provenance queries are then rewritten to standard SQL queries, leveraging existing query optimization techniques. GProM uses the query rewriting approach from Perm in a generic provenance middleware, in addition to supporting updates, transactions, and operation-spanning transactions. In the ProQL system, SQL is extended to support how-provenance queries.

Transaction temporal databases keep track of the different states of the database as tuples are added, deleted, or updated. Therefore, they also support fine-grained provenance and reproducibility: users can revisit old states and maintain provenance of query results even when data changes [28]. Large database vendors, such as Oracle [31] and DB2 [53], support version control for their database systems by using temporal models.

3.2 Workflow Provenance

Workflow provenance consists of the record of the derivation of a result (e.g., a dataset, an image, a plot, etc.) by a computational process represented as a scientific workflow [15]. Scientific workflows are often represented as dataflows, directed acyclic graphs (DAG) whose vertices are modules (functions) that perform computations, and data flows through the edges which connect modules. They adopt a functional, deterministic model where each module receives some input data and generates a new output: the workflow structure and inputs uniquely identify the outputs.

There are several advantages of describing computations as workflows. Notably: they provide a simple programming model where a sequence of tasks is composed by connecting the outputs of one task to the inputs of another; they support useful manipulations, such as the ability to query workflows and update them in a programmatic fashion [54]; through abstraction, they provide a high-level description of the experiment, making the specification easier to understand and more amenable for publication; and by providing a unified environment to run computations, they facilitate *provenance capture*.

Different from database provenance, workflow provenance is *coarse-grained*. Modules are black boxes and provenance captures input data they consume, the function they execute and associated parameters, and data they output; operations inside the function are not visible and no information is available about how the function manipulates the input data.

There are different types of workflow provenance [20]. *Prospective provenance* corresponds to the description of the experiment and captures the specification of the workflow structure, including its modules, connections, and inputs. *Retrospective provenance*, on the other hand, captures information about the execution of the workflow, i.e., what actually happened when the workflow was run. *Workflow evolution provenance* captures the history of the workflow, i.e., the changes that were applied to a workflow over time. If we consider the PRIMAD variables, workflow provenance contains information about the data D (input files and parameters) and the implementation I (binaries or source code for each computational step). While information about the platform P is available, it is often not captured by workflow systems.

Levels of Reproducibility. Workflow provenance supports *repeatable* and *re-runnable* experiments, as parameters and data files are systematically captured and can be varied. Workflows are also *extendable*, as scientists can change the structure of the dataflow (i.e., the prospective provenance) or incorporate it in their own pipelines. A workflow is *modifiable* if the source code for the modules is available. But if a module relies on a binary executable or a Web service to process the data, it is not possible to change the implementation. Note that, if the workflow system does not capture the platform P, portability cannot be supported.

Tools. Many scientific workflow systems are available. Taverna [41] was developed to stitch together Web and third party services: a module invokes either a Web service or a local service (e.g., R scripts, or Java API classes and methods). If a workflow is only composed of Web services, it is *portable* assuming these services are available (live and accessible) and running, but not *modifiable*. Note that using third-party resources (i.e., services provided by external hosts) may impact both repeatability and longevity because they may be interrupted or changed without notice, causing the workflow to fail.

Kepler [39] and VisTrails [19] capture both prospective and retrospective provenance. A new concept introduced by VisTrails was the notion of *provenance of workflow evolution* [21]. VisTrails treats the workflows as first-class data items and also captures their provenance. The utility of workflow-evolution provenance goes beyond reproducibility. It supports reflective reasoning [45]: users can explore multiple chains of reasoning without losing any results, and because the system stores intermediate results, users can reason about and make inferences from this information. It also enables a series of operations which simplify the exploratory process that is common for scientific experiments. For example, users can easily navigate through the space of workflows created for a given task, visually compare the workflows and their results, and explore (large) parameter spaces [21]. In addition, users can query the provenance information [55] and modify workflows by analogy [54]. VisTrails manages data manipulated by the workflows and their versions by storing input, output, and intermediate data in a versioned repository. This ensures that different versions of the input data can be recovered in the future. Using this approach, the workflow does not depend on hardcoded filenames [33]. With respect to longevity, VisTrails has a mechanism that detects when upgrades are necessary by using provenance to compare a module in a given workflow with its currently available version [34].

Galaxy [23] is a platform used to perform computational analysis on genomic data. Similar to VisTrails, Galaxy can capture information about workflow evolution, keeping multiple versions of the analysis steps so that scientists can re-run previous computations. Because the system provides a Web-based interface to create and run workflows, allowing them to be accessed through the Web from any platform, *portability* is easily attainable.

3.3 Script Provenance

Script provenance is obtained by analyzing the source code of experiments represented as scripts. Similar to workflows, we can have *prospective* and *retrospective* provenance for scripts. While the former corresponds to the script specification, the latter records the actual execution of the code (i.e., which execution branches were taken, which values were used and processed, etc.). It is also possible to capture provenance for the evolution of scripts, for example, using version control systems. To capture script provenance, code is instrumented either automatically or manually through user-defined annotations. Automatic capture has the advantage of not requiring user intervention. On the other hand, the resulting provenance data may be voluminous if provenance is captured at a fine level of detail. While annotations can be intrusive and time-consuming to add, they allow users to precisely define what should be captured.

Note that script provenance often has a *finer granularity* than workflow provenance. While workflow modules are black boxes, for scripts it is possible to observe all operations in the source code.

Levels of Reproducibility. The variables D (parameters and data files) and I (source code) are often captured, allowing an experiment to be *repeatable*, *re-runnable*, *extendable*, and *modifiable*. Provenance may also be captured for the platform P, including the library dependencies explicitly used in the code. While this may not be sufficient for *portability*, it useful for analyzing and comparing execution traces.

Tools. noWorkflow [43] transparently captures retrospective provenance from Python scripts. It does not require any code instrumentation, and the captured provenance—which includes metadata, file contents, Python dependencies (i.e., environment information at the Python level), and parameters—can be analyzed by inspecting a provenance graph, comparing multiple executions, or using inference queries. Users can choose the amount of provenance information obtained by setting the depth of the capture. noWorkflow also allows scientists to analyze workflow evolution [49].

RDataTracker [37] also captures retrospective provenance from scripts, but for the R software package. Similar to noWorkflow, the tool collects and persists provenance information, provides a provenance graph that can be inspected by users, and supports querying. Users must manually annotate the code.

Tariq et al. [56] proposed an approach through which provenance instrumentation is added at compilation time using the compiler framework LLVM [35]. This works for a range of programming languages, such as C, C++, and Java. Hooks for the capture of retrospective provenance are inserted at each function entry and exit: when the framework compiles the experiment, provenance at the binary level is transparently captured, with a small overhead.

YesWorkflow [40] captures prospective, rather than retrospective provenance. Scientists can make use of language-independent annotations to make latent dataflow information from scripts explicit. There has been also work on linking the retrospective provenance collected by noWorkflow with the prospective provenance obtained by YesWorkflow [48]. ProvenanceCurious [27] is another tool for capturing prospective provenance: it builds an abstract syntax tree from Python code to generate provenance graphs.

3.4 System-Level Provenance

System-level provenance corresponds to the provenance data captured at the operating system level, which often includes the description of the platform P, providing detailed trails of how data products are derived. This provenance is often captured by monitoring system calls and tracking processes and data dependencies between these processes. Because the dependencies are recorded at the process level, the provenance data is fine-grained. Note, however, that these processes are black boxes: it is not possible to observe what happens inside them. For this reason, in terms of the implementation I, the transparency can be low.

Levels of Reproducibility. System-level provenance is often composed of D (input parameters and data files), P, and I (binaries, at least); therefore, it can help experiments to achieve all reproducibility levels: *repeatable*, *re-runnable*, *portable*, and *extendable*. It is also possible to attain *modifiability* if the source code for the processes is available.

Tools. PASS (Provenance-Aware Storage System) [42] produces audit trails for data products by monitoring the operating system kernel. These audit trails are stored in a database and can be queried. The system supports application-generated provenance to be written into the same database via library functions, generating an integrated view of application and system-level provenance. Moreover, PASS can generate scripts to reproduce an experiment in the environment in which it was originally created and executed, thus allowing it to be *repeated.*

A number of tools have been developed to *package* and preserve the computational environment. Tools like

CDE [25], CARE [30], ReproZip [11, 51], and PTU [47] trace the execution path of an experiment and obtain information about all the experiment's dependencies (e.g., files read, libraries used). This information can be used to create a package that encompasses both the experiment and its dependencies, thus supporting portability and longevity.

Most of these packing tools only support reproducibility in Linux environments that are compatible with the platform where the experiment was originally captured. CARE supports *enhanced kernel compatibility*, allowing within certain limits, for an experiment to be executed in a kernel older than the one used in the original implementation. ReproZip has an option to automatically reproduce an experiment in a virtual environment or container, thus allowing the reproduction to take place in *any* platform. ReproZip also enables scientists to inspect the captured provenance and exclude certain components from the package. Scientists can also add extra files to the package, e.g., one can add the source code if it was not originally traced. ReproZip automatically derives a VisTrails workflow for the experiment, which facilitates extending and re-using the original dataflow. PTU uses CDE internally to pack the experiment and enhances its functionalities by storing a provenance graph that can be visually inspected by scientists to determine parts of the program they want to reproduce, i.e., scientists can choose to repeat subsets of the experiment.

4 Discussion

Provenance is essential for the reproducibility of computational experiments. In this paper, we discussed different notions of reproducibility and gave an overview of different types of provenance and how they support the various levels of reproducibility. Over the past few years, there have been major developments in basic infrastructure and tools that, through systematic provenance capture, make it easier to create reproducible experiments. We have described some of these tools, the benefits they bring as well as their limitations.

While progress has been made, reproducibility is still not the norm for computational experiments. An important barrier to the adoption of reproducibility is the perception that it is time-consuming to create reproducible experiments. Recent studies have shown that insufficient time is one of the main reasons why scientists do not make their data and experiments available and reproducible [57], and that substantial effort is needed to make code work with the latest versions of required dependencies [13]. In addition, many authors argue that the process to make an experiment reproducible requires too much work for the benefit derived [6]. While one can interpret some of these as excuses, they do indicate that *usability* is an important requirement for a broader adoption of reproducibility: tools must be easy to use and, ideally, as automatic as possible. In the words of Vandewalle et al. [59], "an independent researcher should be able to reproduce all the results with a simple mouse click."

Some experiments require multiple types of provenance to be captured. For example, for a scientific workflow or script that queries and/or updates a database, it is important to collect both workflow or script provenance and database provenance. To support portability, it is also important to capture system-level provenance and detailed information about the platform and dependencies. The integration of these different types of provenance opens new opportunities to query and reason about an experiment at multiple levels. In particular, this information can enable explanations for potential issues encountered in an experiment, which can arise from the data, code, platform, or from interactions among them. But integrating different types of provenance is challenging because they are captured at different levels and have different granularities. In addition, domain-specific languages are needed that support queries over these data.

Acknowledgments. This work was supported in part by the NYU Moore Sloan Data Science Environment and National Science Foundation awards OAC-1640864, CNS-1229185, CNS-1405927, and IIS-1139832.

References

[1] B. Arab, D. Gawlick, V. Radhakrishnan, H. Guo, and B. Glavic. A Generic Provenance Middleware for Queries, Updates, and Transactions. In *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*. USENIX Association, 2014.

[2] M. Baker. Muddled Meanings Hamper Efforts to Fix Reproducibility Crisis. *Nature News & Comment*, June 2016.

[3] C. G. Begley and L. M. Ellis. Drug Development: Raise Standards for Preclinical Cancer Research. *Nature*, 483(7391), 2012.

[4] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 953–964. VLDB Endowment, 2006.

[5] J. Bohannon. Who's Afraid of Peer Review? *Science*, 342(6154):60–65, 2013.

[6] P. Bonnet, S. Manegold, M. Bjørling, W. Cao, J. Gonzalez, J. Granados, N. Hall, S. Idreos, M. Ivanova, R. Johnson, D. Koop, T. Kraska, R. Müller, D. Olteanu, P. Papotti, C. Reilly, D. Tsirogiannis, C. Yu, J. Freire, and D. Shasha. Repeatability and Workability Evaluation of SIGMOD 2011. *SIGMOD Rec.*, 40(2):45–48, 2011.

[7] T. Brody. *Evaluating Research Impact through Open Access to Scholarly Communication*. PhD thesis, University of Southampton, May 2006.

[8] P. Buneman and W.-C. Tan. Provenance in Databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1171–1173. ACM, 2007.

[9] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Found. Trends databases*, 1(4):379–474, 2009.

[10] F. Chirigati and J. Freire. Provenance and Reproducibility. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 1–5. Springer New York, 2017.

[11] F. Chirigati, D. Shasha, and J. Freire. ReproZip: Using Provenance to Support Computational Reproducibility. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, TaPP '13, pages 1:1–1:4. USENIX Association, 2013.

[12] C. Collberg and T. A. Proebsting. Repeatability in Computer Systems Research. *CACM*, 59(3):62–69, Feb. 2016.

[13] C. Collberg, T. A. Proebsting, and A. M. Warren. Repeatability and Benefaction in Computer Systems Research: A Study and a Modest Proposal. Technical Report 14-04, University of Arizona, February 2015.

[14] S. M. Crook, A. P. Davison, and H. E. Plesser. Learning from the Past: Approaches for Reproducibility in Computational Neuroscience. In J. M. Bower, editor, *20 Years of Computational Neuroscience*, volume 9 of *Springer Series in Computational Neuroscience*, pages 73–102. Springer New York, 2013.

[15] S. B. Davidson and J. Freire. Provenance and Scientific Workflows: Challenges and Opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1345–1350. 2008.

[16] D. Donoho, A. Maleki, I. Rahman, M. Shahram, and V. Stodden. Reproducible research in computational harmonic analysis. *Computing in Science & Engineering*, 11(1):8–18, Jan.-Feb. 2009.

[17] J. Freire, P. Bonnet, and D. Shasha. Computational Reproducibility: State-of-the-Art, Challenges, and Database Research Opportunities. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 593–596. ACM, 2012.

[18] J. Freire, N. Fuhr, and A. Rauber. Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041). *Dagstuhl Reports*, 6(1):108–159, 2016.

[19] J. Freire, D. Koop, E. Santos, C. Scheidegger, C. Silva, and H. T. Vo. Vistrails. In A. Brown and G. Wilson, editors, *The Architecture of Open Source Applications*. Lulu Publishing, Inc., 2011.

[20] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science and Engineering*, 10(3):11–21, 2008.

[21] J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing Rapidly-evolving Scientific Workflows. In *Proceedings of the 2006 International Conference on Provenance and Annotation of Data*, IPAW'06, pages 10–18. Springer-Verlag, 2006.

[22] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the Same Data Model Through Query Rewriting. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 174–185. IEEE Computer Society, 2009.

[23] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: A Comprehensive Approach for Supporting Accessible, Reproducible, and Transparent Computational Research in the Life Sciences. *Genome Biology*, 11:1–13, 2010.

[24] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update Exchange with Mappings and Provenance. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 675–686. VLDB Endowment, 2007.

[25] P. Guo. CDE: A Tool for Creating Portable Experimental Software Packages. *Computing in Science Engineering*, 14(4):32–35, 2012.

[26] S. Hitchcock. The Effect of Open Access and Downloads ('Hits') on Citation Impact: A Bibliography of Studies. Technical report, University of Southampton, 2009.

[27] M. R. Huq, P. M. G. Apers, and A. Wombacher. ProvenanceCurious: A Tool to Infer Data Provenance from Scripts. In *Proceedings of EDBT '13*, pages 765–768. 2013.

[28] M. R. Huq, A. Wombacher, and P. M. G. Apers. Facilitating Fine Grained Data Provenance Using Temporal Data Model. In *Proceedings of the Seventh International Workshop on Data Management for Sensor Networks*, DMSN '10, pages 8–13. ACM, 2010.

[29] J. P. A. Ioannidis. Why Most Published Research Findings Are False. *PLoS Med*, 2(8), Aug. 2005.

[30] Y. Janin, C. Vincent, and R. Duraffort. CARE, the Comprehensive Archiver for Reproducible Execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, TRUST '14, pages 1:1–1:7. ACM, 2014.

[31] K. Jernigan, L. Guo, V. Krishnaswamy, V. Radhakrishnan, V. Raja, and T. Shetler. Oracle Total Recall with Oracle Database 11g Release 2. White paper, Oracle, 2009.

[32] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying Data Provenance. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 951–962. ACM, 2010.

[33] D. Koop, E. Santos, B. Bauer, M. Troyer, J. Freire, and C. T. Silva. Bridging Workflow and Data Provenance Using Strong Links. In M. Gertz and B. Ludäscher, editors, *22nd International Conference on Scientific and Statistical Database Management, SSDBM 2010*, pages 397–415. Springer Berlin Heidelberg, 2010.

[34] D. Koop, C. E. Scheidegger, J. Freire, and C. T. Silva. The Provenance of Workflow Upgrades. In D. L. McGuinness, J. R. Michaelis, and L. Moreau, editors, *Third International Provenance and Annotation Workshop, IPAW 2010*, pages 2–16. Springer Berlin Heidelberg, 2010.

[35] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04. IEEE Computer Society, 2004.

[36] S. Lawrence. Free Online Availability Substantially Increases a Paper's Impact. *Nature*, 411(6837):521, May 2001.

[37] B. Lerner and E. Boose. RDataTracker: Collecting Provenance in an Interactive Scripting Environment. In *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*. USENIX Association, 2014.

[38] R. LeVeque. Python tools for reproducible research on hyperbolic problems. *Computing in Science & Engineering*, 11(1):19–27, Jan.-Feb. 2009.

[39] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System: Research Articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, Aug. 2006.

[40] T. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, R. K. Bocinsky, Y. Cao, J. Cheney, F. Chirigati, S. Dey, J. Freire, C. Jones, J. Hanken, K. W. Kintigh, T. A. Kohler, D. Koop, J. A. Macklin, P. Missier, M. Schildhauer, C. Schwalm, Y. Wei, M. Bieda, and B. Ludäscher. YesWorkflow: A User-Oriented, Language-Independent Tool for Recovering Workflow Information from Scripts. *International Journal of Digital Curation*, 10:298–313, 2015.

[41] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, Reloaded. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management*, SSDBM'10, pages 471–481. Springer-Verlag, 2010.

[42] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware Storage Systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06. 2006.

[43] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noWorkflow: Capturing and Analyzing Provenance of Scripts. In B. Ludäscher and B. Plale, editors, *Proceedings of the 5th International Provenance and Annotation Workshop, IPAW 2014*, pages 71–83. Springer International Publishing, 2015.

[44] G. Naik. Mistakes in Scientific Studies Surge, August 2011. `http://online.wsj.com/article/SB10001424052702303627104576411850666582080.html`.

[45] D. A. Norman. *Things That Make Us Smart: Defending Human Attributes in the Age of the Machine*. Addison Wesley, 1994.

[46] R. Nuzzo. How Scientists Fool Themselves, and How They Can Stop. *Nature*, 526(7572):182–185, 2015.

[47] Q. Pham, T. Malik, and I. Foster. Using Provenance for Repeatability. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, TaPP '13, pages 2:1–2:4. USENIX Association, 2013.

[48] J. F. Pimentel, S. Dey, T. McPhillips, K. Belhajjame, D. Koop, L. Murta, V. Braganholo, and B. Ludäscher. Yin & Yang: Demonstrating Complementary Provenance from noWorkflow & YesWorkflow. In M. Mattoso and B. Glavic, editors, *Proceedings of the 6th International Provenance and Annotation Workshop, IPAW 2016*, pages 161–165. Springer International Publishing, 2016.

[49] J. F. Pimentel, J. Freire, V. Braganholo, and L. Murta. Tracking and Analyzing the Evolution of Provenance from Scripts. In M. Mattoso and B. Glavic, editors, *Proceedings of the 6th International Provenance and Annotation Workshop, IPAW 2016*, pages 16–28. Springer International Publishing, 2016.

[50] H. A. Piwowar, R. S. Day, and D. B. Fridsma. Sharing Detailed Research Data Is Associated with Increased Citation Rate. *PLoS ONE*, 2(3):e308, 03 2007.

[51] R. Rampin, F. Chirigati, D. Shasha, J. Freire, and V. Steeves. ReproZip: The Reproducibility Packer. *The Journal of Open Source Software (JOSS)*, 2016.

[52] A. Rokem and F. Chirigati. Glossary. In J. Kitzes, D. Turek, and F. Deniz, editors, *The Practice of Reproducible Research: Case Studies and Lessons from the Data-Intensive Sciences*, pages 71–91. UC Press, 2017.

[53] C. M. Saracco, M. Nicola, and L. Gandhi. A Matter of Time: Temporal Data Management in DB2 for z/OS. White paper, IBM, 2010.

[54] C. Scheidegger, H. Vo, D. Koop, J. Freire, and C. Silva. Querying and Creating Visualizations by Analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.

[55] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and re-using workflows with vistrails. In *ACM SIGMOD*, pages 1251–1254, 2008.

[56] D. Tariq, M. Ali, and A. Gehani. Towards automated collection of application-level data provenance. In *4th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2012)*. USENIX, 2012.

[57] C. Tenopir, S. Allard, K. Douglass, A. U. Aydinoglu, L. Wu, E. Read, M. Manoff, and M. Frame. Data Sharing by Scientists: Practices and Perceptions. *PLOS ONE*, 6(6):1–21, 2011.

[58] R. Van Noorden. Science Publishing: The Trouble with Retractions. *Nature*, 478:26–28, 2011.

[59] P. Vandewalle, J. Kovacevic, and M. Vetterli. Reproducible Research in Signal Processing - What, Why, and How. *IEEE Signal Processing Magazine*, 26(3):37–47, May 2009.

Data Citation: A New Provenance Challenge

Abdussalam Alawini, Susan Davidson, Gianmaria Silvello, Val Tannen, Yinjun Wu

Abstract

In today's era of big data-driven science, an increasing amount of information is being published as curated online databases and retrieved by queries, raising the question of how query results should be cited. Because it is infeasible to associate citation information with every possible query, one approach is to specify citations for a small set of frequent queries – citation views *– and then use these views to construct a citation for general queries. In this paper, we describe this model of citation views, how they are used to construct citations for general queries, and an efficient approach to implementing this model. We also show the connection between data citation and data provenance.*

1 Introduction

Citation is an essential part of scientific and scholarly publishing. It is crucial for gauging the trust placed in published data and giving appropriate credit to authors. However, the nature of publication is shifting from traditional venues – such as books, journals and conferences – for which citation is well understood, to databases containing curated information which is retrieved by queries. This is especially true in Big Data-driven science, where many scientific reference works and collections of experimental results are now being published as curated on-line databases with web-page views.

Typically, database owners specify the citation to a web-page view as a journal article whose title includes the name of the database and whose author list includes the chief personnel (e.g. the PI, DBA, lead annotator, etc), along with the query and date of access. However, in many cases the content of the query result is contributed by members of the community and curated by experts, who are not on the author list of the journal article, and the lack of appropriate citation is becoming a stumbling block as evidenced by the recent "data parasite" controversy [14]. Appropriate data citation is therefore essential to motivate members of the scientific community to continue to share data and experimental results so that they can be used and built on by others in the advancement of science.

Many of the "citable" databases that we have examined - e.g. the *Reactome Pathway* database [13] and the *eagle-i* [17] resource discovery tool - describe in English what snippets of information are to be included in a citation for data displayed on a web page; however, users must then construct the citation by hand. However, the English specifications are fairly complex, and the effort required to pull the data off the web page discourages users from generating the citations. Users are therefore unlikely to cite this data correctly unless the citations are *automatically* generated and returned to users along with the data retrieved by a query.

The most advanced database from the perspective of citation that we have examined is the *IUPHAR/BPS Guide to Pharmacology (GtoPdb)* [16], a relational database that contains expertly curated information about

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

drugs, the cellular targets of the drugs, and their mechanisms of action in the body. In this database, users view information through a hierarchy of web pages. The top level divides information by "families" of drug targets that reflect typical pharmacological thinking; lower levels divide the families hierarchically into sub-families and so on down to individual drug targets and drugs. For data displayed in Family and Family Introduction web pages, GtoPdb places a human readable citation calculated from information in the underlying database in the page; the citation can then be copy-pasted and reformatted into whatever style the user wishes. The citation varies depending on the part of the database being queried (e.g. the particular Family), and contains an identifier for the data along with information about the contents (analogous to a title), the contributors and/or curators of the data (analogous to authors), the date on which the contents were last modified, and the date the database was queried. In the future, owners of GtoPdb would like to enable general queries against the underlying database rather than restricting access to the database to queries expressed as web-pages, and automatically return citations along with the data. The question is: how should citations to general queries be generated?

Data citation is a challenge because, unlike traditional publications which have a fixed granularity to which citations can be attached, the granularity of reference varies; there are a large number of possible queries over a database, each returning a different subset of data. The "snippets" of information to include in the citation may also vary from query to query, e.g. descriptive information about the data subset being returned, analogous to the distinct titles that different chapters have in an edited collection. Note that these snippets of information play an important *human* role: While the query and date of access (or some form of digital object identifier) is sufficient to locate the query result, they are not informative enough if used as a citation as they do not give intuition about the content. This is analogous to the fact that, in traditional journals, a citation like "Nature, 171,737-738" specifies how to locate the article but doesn't tell you why you might want to do so, whereas adding the information "Watson and Crick: Molecular Structure of Nucleic Acids" does. It is therefore necessary to be able to *specify* citations to query results [5].

Since it is impossible to specify the content of a citation to every possible query over a database, one strategy is to specify citations to a small number of frequent queries – *citation views* [8, 9] – and use these to construct citations to other "general" queries. The citation views may be combined (*jointly* used) to construct the citation, and there may be *alternate* ways in which combinations of citation views can be used. The interpretations of joint and alternate use (e.g. union or join) are *policies* to be specified by the database owner.

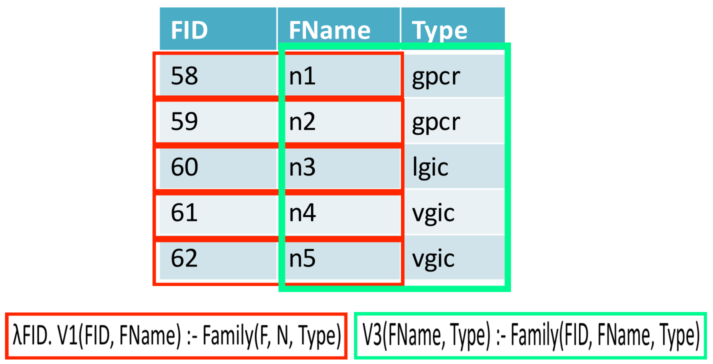
There is an interesting connection between data citation and data provenance: Naively, citation captures the "origins" of data by giving credit to the people responsible for it. However, the connection goes deeper by viewing both citation and provenance as *annotations on data that are carried through queries*. That is, each tuple t in a base relation is annotated with a view iff t is used to construct the materialized view. When a user query is issued, the view annotations are regarded as provenance tokens to be propagated along with values to the final query result. The view annotations of each tuple are then reasoned over to determine whether or not they are valid in a citation for the tuple.

In the remainder of this paper we explore this connection. We start in Section 2 by describing the model of citation views and how they are used to construct citations to general queries. In Section 3 we discuss the connection to *where*- and *why*-provenance [6, 11]. We then describe in Section 4 an efficient approach to implementing the model which draws on the ideas of the previous two sections. We conclude in Section 5.

2 Model of Citation Views

The citation framework is based on *conjunctive queries* [1]. Conjunctive queries are "universal" across different types of databases (e.g. relational, semistructured, RDF, etc.), and simplify the reasoning used to generate citations. Throughout the paper, we will use Datalog as the syntax for queries, and assume that fresh variables are introduced everywhere in relational subgoals rather than being reused.

We start by defining the notion of citation views, which defines how to associate citations to a fixed set of



FID	FName	Type
58	n1	gpcr
59	n2	gpcr
60	n3	lgic
61	n4	vgic
62	n5	vgic

λFID. V1(FID, FName) :- Family(F, N, Type)

V3(FName, Type) :- Family(FID, FName, Type)

Figure 1: Effect of Parameters on Views

frequent queries, and then give a semantics for how to associate citations to *general* queries based on citation views.

2.1 Citation views.

A citation view specifies: 1) the data being cited (*view definition*); 2) the information to be used to construct the citation (*citation queries*); and 3) how the information is combined to construct the citation (*citation function*). The citation function takes the snippets of information retrieved from citation queries as input, and generates an appropriately formatted citation as output (e.g. human readable, BibTex, RIS or XML). Note that the snippets of information required for the citation must be in the database, and that the citation can be thought of as an *annotation* on every tuple in the view result.

The view definition and citation queries are optionally *parameterized*, where the parameters (λ-*variables*) appear as variables somewhere in the body of the query. [1] A parameterized view creates a set of *instantiated views*, one for each possible choice of parameters. The number of such views is therefore instance-dependent, as illustrated in Figure 1. We will use the input parameter(s) to distinguish such views, e.g. V1(60) refers to the instantiated view V1 for FID=60.

Example 1: Recall that in the GtoPdb database, users view information through a hierarchy of web pages. The top level divides information by families of drug targets; lower levels divide the families into sub-families and so on down to individual drug targets and drugs. The content of a particular family "landing" page (referred to as the Family relation) is curated by a committee of experts; a family may also have a "detailed introduction page" (referred to as the FamilyIntro relation) which is written by a set of contributors, who are not necessarily the same as the committee of experts for the family.

Citation views for the Family and FamilyIntro relations can be specified as follows. Views V1, V2 and V4 are parameterized by the key FID, whereas V3 is unparameterized.

$$\begin{array}{ll}
\lambda FID.V1(FID, FName) & :-Family(FID, FName, Type) \\
\lambda FID.V2(FID, Text) & :-FamilyIntro(FID, Text) \\
\qquad V3(FName, Type) & :-Family(FID, FName, Type) \\
\lambda FID.V4(FID, FName, Text) & :-Family(FID, FName, Type), FamilyIntro(FID1, Text), FID = FID1
\end{array}$$

For each view V, we define one or more citation queries C_V. We show below examples of citation queries for views V1 and V3, where relation FC captures the committee members who curate the content for V1 and relation MetaData captures general snippets of information such as the owner and url of the database.

$$\begin{array}{ll}
\lambda FID.\ C_{V1}(FID, FName, PName) & :-\ Family(FID, FName, Type), FC(FID, PID), Person(PID, PName) \\
\qquad C_{V3}(X1, X2) & :-\ MetaData(T1, X1), T1 = \text{'}Owner\text{'}, MetaData(T2, X2), T2 = \text{'}URL\text{'}
\end{array}$$

[1] Also called *binding patterns* in [15].

This information can then be used to construct citations for tuples in the query result. For example, the JSON-formatted citation for V1(60) could be {FID: '60', FName: 'RXFP', PName: ['Roger', 'Ross']}, and that for V3 could be {Owner: ['Alexander', 'Davenport'], URL: 'http://www.guidetopharmacology.org'}. $\square$

Attaching citations to general queries. To give a semantics to citations for *general queries*, we use the following intuition: *If a view tuple can be used to create a tuple and is visible in the query result, then the result tuple carries the view tuple's citation annotation.* To do this, mappings between the view definitions and input query are created which maximally and non-redundantly cover the query subgoals; we call this a *covering set* of mappings [3]. This is similar to the notion of query rewriting using views as used, for example, in query optimization and data integration [12]. More formally:

Definition 1: Given a view definition V and query Q

$$\mathtt{V}(\bar{\mathtt{Y}}) : -\mathtt{A}_1(\bar{\mathtt{Y}}_1), \mathtt{A}_2(\bar{\mathtt{Y}}_2), \ldots, \mathtt{A}_k(\bar{\mathtt{Y}}_k), \mathtt{condition}(\mathtt{V})$$
$$\mathtt{Q}(\bar{\mathtt{X}}) : -\mathtt{B}_1(\bar{\mathtt{X}}_1), \mathtt{B}_2(\bar{\mathtt{X}}_2), \ldots, \mathtt{B}_m(\bar{\mathtt{X}}_m), \mathtt{condition}(\mathtt{Q})$$

in which A_i, B_j are relational subgoals, and condition() are *non-relational* subgoals which include comparisons of variables to constants (called *local* predicates) and comparisons of variables with variables (called *global* predicates if the variables come from different relational subgoals and *local* otherwise). Then a **view mapping** M from V to Q is a tuple (h, ϕ) in which:

- h is a partial one-to-one function from $\{A_1, ..., A_k\}$ to $\{B_1, ..., B_m\}$ which 1) maps A_i to B_j only if they have the same relation name; and 2) cannot be extended to include more subgoals of Q (i.e. there is no unmapped A_i, B_j which have the same relation name).
- ϕ are the variable mappings from $\bar{Y}' = \cup_{i=1}^{k} \bar{Y}_i$ to $\bar{X}' = \cup_{i=1}^{m} \bar{X}_i$ induced by h

A relational subgoal B_j of Q is *covered* iff $h(A_i) = B_j$ for some i. A variable $x_j \in \bar{X}'$ is *covered* iff $\phi(y_i) = x_j$ for some variable $y_i \in \bar{Y}'$. Note that a view may be in zero or more view mappings for a given query.

We also use the notion of the *extension* of Q, called Q_{ext}, which expands the head of Q to include all variables in the body ($\bar{X}'$).

Definition 2: Valid View Mapping Given a database instance D, a view mapping $M = (h, \phi)$ of V is *valid* for a tuple $t \in Q_{ext}(D)$ iff:

- The projection of t on the variables that are mapped in Q_{ext} under the mapping ϕ is a tuple in $V_{ext}(D)$: $\Pi_{\phi(\bar{Y}')} t \in V_{ext}(D)$
- There exists at least one variable $y \in \bar{Y}$ such that $\phi(y)$ is a distinguished variable.
- All lambda variables in V are mapped to variables in $\bar{X}'$.

Given a set of views $\mathcal{V}$, a query Q and a database instance D, we can build a set of valid view mappings $\mathcal{M}(t)$ for each tuple $t \in Q(D)$ according to Definitions 1 and 2. We then combine different view mappings from $\mathcal{M}(t)$ to create a *covering set* of views for t.

Definition 3: Covering set Let $C \subseteq \mathcal{M}(t)$ be a set of valid view mappings. Then C is a covering set of view mappings for t iff it is *maximal* and *nonredundant*:

- No $V \in \mathcal{M}(t) \setminus C$ can be added to C to cover more subgoals of Q or variables in $\bar{X}$; and

- No $V \in C$ can be removed from C and cover the same subgoals of Q and variables in $\bar{X}$.

Within a covering set, the citation views are *jointly* used (indicated by "*") to construct a citation, and if there are more than one covering set the citations of the covering sets are *alternatively* used (indicated by "$+^R$") to construct a citation to each tuple. The citations for each tuple in the query result can then be *aggregated* (indicated by *Agg*) to form a citation for the entire query result.

Example 2: In this example, there will be at most one valid view mapping from a view V to a query Q; we will therefore use the name of the view as the name of the mapping.

Suppose we had the input query

$$Q1(Name) : -Family(FID, Name, Type), FID >= 60$$

For this, one covering set is {V1}: there is a mapping from the body of V1 to the relational subgoal (Family) as well as an input to the parameter FID. {V3} and {V4} are also covering sets; note that mapping V4 is partial since FamilyIntro is not mapped to any relational subgoal in $Q1$. The citation for the first tuple (FID=60) would therefore alternatively use the citations for V1(60), V3, and V4(60), i.e. it would be Cite($V1(60) +^R V3 +^R V4(60)$). If "$+^R$" were interpreted as "most specific", the resulting citation would be Cite($V1(60)$). Similarly, for tuple FID=62 the citation would be Cite($V1(62) +^R V3 +^R V4(62)$). The citations for each tuple are then aggregated to derive a citation for the entire query result. If aggregation were interpreted as some form of intersection, and "$+^R$" balanced size with specificity, the citation for the query result would be Cite(V3).

On the other hand, consider the following the input query

$$Q2(Type, Text) : -Family(FID1, Name, Type), FamilyIntro(FID2, Text), FID1 = FID2 = 60$$

One covering set is {V2, V3}: there are mappings from the body of V3 to the relational subgoal Family as well as from the body of V2 to the relational subgoal FamilyIntro. We would therefore jointly use the citations for V3 and V2, written Cite($V3 * V2(60)$). Another covering set is {V1, V2}, resulting in citation Cite($V1(60) * V2(60)$). Finally, there is a mapping from the body of V4 which covers all relational subgoals of $Q2$, however {V4} is not a covering set since it only covers the distinguished variable $Text$, and can be augmented with V3 to cover both $Type$ and $Text$. The third covering set is therefore {V3, V4}. The final citation for the single result tuple is therefore Cite($V1(60) * V2(60) +^R V3 * V2(60) +^R V3 * V4(60)$).

Finally, suppose we had an input query which is a subset of the cross product of Family and FamilyIntro:

$$Q3(Type, Text) : -Family(FID1, Name, Type), FamilyIntro(FID2, Text), FID1 <= 60, FID2 <= 60$$

Note that some of the tuples in the result may be in the join of the two tables, and therefore be visible in V4, while others are not. This motivates the need to evaluate the *extended* query $Q3_{ext}$. Since this returns the value of all variables in the body, the validity of the join predicate FID1=FID2 can be evaluated for each result tuple, thereby determining whether the V4 is valid for that particular tuple. Thus, for tuples in the result that are not in the join, the covering sets would be {V1, V2} and {V3, V2}, while for tuples in the join it would also include {V3, V4} (as in $Q2$). □

3 Connection to Provenance

To understand the connection to provenance, we start with a simple but common subclass of view definitions called *partitioning views* which corresponds to *where-* provenance, and then move to the (more complex) general case which corresponds to *why*-provenance [6, 11].

3.1 Partitioning Views

A set of select-project views over a single relation R is partitioning if each attribute of R appears in at most one view; a set of views is partitioning over a database schema $\mathcal{S}$ if it is partitioning for each $R \in \mathcal{S}$.[2] As an example, the following set of citation views is partitioning for our running example:

$$\begin{array}{ll} \lambda FID.V10(FID, FName) & \text{:- } Family(FID, FName, Type), Type = \text{‘}gpcr\text{'} \\ \lambda FID.V11(Type) & \text{:- } Family(FID, FName, Type) \\ \lambda FID.V12(FID, Text) & \text{:- } FamilyIntro(FID, Text) \end{array}$$

In this case, since fresh variables are introduced for every relational subgoal in queries, each attribute of each tuple in a query result is visible in at most one view, and must be the same view across all tuples.[3]

In *where*-provenance [4, 10], each attribute of each tuple in an instance of a relation is annotated with a (unique) provenance token, which is then carried through queries to annotate tuples in the query result. Assume for now that duplicates in queries are not removed, and that views are *materialized*. Then the citation for each tuple in the query result would be the set (*joint use*) of citation views in which the where-provenance tokens for attributes in the tuple appear. When duplicates are removed, the union of the sets of citation views for each duplicate tuple would be used.

Table 3: Base relations with provenance tokens

(a) Family

	FID		FName		Type		
t_1	58	a_1	n1	a_6	gpcr	a_{11}	s_1
t_2	59	a_2	n2	a_7	gpcr	a_{12}	s_2
t_3	60	a_3	n3	a_8	lgic	a_{13}	s_3
t_4	61	a_4	n4	a_9	vgic	a_{14}	s_4
t_5	62	a_5	n5	a_{10}	vgic	a_{15}	s_5

(b) FamilyIntro

	FID		Text		
t'_1	58	b_1	tx1	b_5	r_1
t'_2	60	b_2	tx2	b_6	r_2
t'_3	61	b_3	tx3	b_7	r_3
t'_4	62	b_4	tx3	b_8	r_4

Table 4: Materialized views V10-V12 and query result Q4(D) with provenance tokens

(a) V10

	FID		FName	
t_{101}	58	a_1	n1	a_6
t_{102}	59	a_2	n2	a_7

(b) V11

	Type	
t_{111}	gpcr	a_{11}, a_{12}
t_{112}	lgic	a_{13}
t_{113}	vgic	a_{14}, a_{15}

(c) V12

	FID		Text	
t_{121}	58	b_1	tx1	b_5
t_{122}	60	b_2	tx2	b_6
t_{123}	61	b_3	tx3	b_7
t_{124}	62	b_4	tx3	b_8

(d) $Q4(D)$

	FID		FName		Type		covering sets
t_{q41}	58	$b_1 \rightarrow \{V12\}$	n1	$a_6 \rightarrow \{V10\}$	gpcr	$a_{11} \rightarrow \{V11\}$	$V12 * V10 * V11$
t_{q42}	60	$b_2 \rightarrow \{V12\}$	n3	$a_8 \rightarrow \{\}$	lgic	$a_{13} \rightarrow \{V11\}$	$V12 * V11$
t_{q43}	61	$b_3 \rightarrow \{V12\}$	n4	$a_9 \rightarrow \{\}$	vgic	$a_{14} \rightarrow \{V11\}$	$V12 * V11$
t_{q44}	62	$b_4 \rightarrow \{V12\}$	n5	$a_{10} \rightarrow \{\}$	vgic	$a_{15} \rightarrow \{V11\}$	$V12 * V11$

Example 3: Consider the provenance-annotated relations in Table 3, and ignore for now the annotations in the last column. Observe that for tuple t_1 in Family the provenance annotation for FID is a_1 and that for FName is

[2]Note that more complicated cases of partitioning could also be considered, e.g. *horizontal* in which conditions on variables in the views are mutually exclusive.

[3]Recall that an attribute appears in at most one view, but that a local predicate may cause some tuples in the underlying relation not to be considered. For example, V10 only applies to tuples in Family whose type is ‘gpcr’.

a_6. In the materialized instance of V10 shown in Table 4a, tuple t_{101} is the projection of t_1 and therefore carries these annotations.

Assume we have the following query which joins the Family and FamilyIntro relations:

Q4(FID, FName, Type):- Family(FID1, FName, Type), FamilyIntro(FID, Text), FID1=FID

The annotated result of Q4(D) is shown in Figure 4d, where we show the mapping from each where-provenance token to the materialized view in which it occurs. For example, the provenance token b_1 in the result tuple t_{q41} appears in tuple t_{121} of $V12(D)$. For each tuple in the query result, the combination of views from each where-provenance token are *jointly* combined to cover as many distinguished variables of query as possible. For instance, for tuple t_{q41} the combination of views is $V10 * V11 * V12$, which is the covering set for tuple t_{q41}. □

3.2 General views

However, when views are *not* partitioning (as in the case of views V1-V4), where-provenance is no longer sufficient; we must also understand how the tuple in the result was constructed from the input tuples (*why*-provenance). The final column in Table 3 represents the why-provenance for each tuple.

Example 4: The materialized instances of views V1-V4 from Example 1, together with their provenance annotations, are shown in Tables 5a-5d and the result of query Q3 from Example 2 with provenance annotations is shown in Table 5e. Note that the views V1-V4 and the query Q3 carry the provenance tokens from the underlying relations.

Table 5: Materialized views V1-V4 and Query result Q3(D) with provenance tokens

(a) V1

	FID		FName		
t_{11}	58	a_1	n1	a_6	$\{s_1\}$
t_{12}	59	a_2	n2	a_7	$\{s_2\}$
t_{13}	60	a_3	n3	a_8	$\{s_3\}$
t_{14}	61	a_4	n4	a_9	$\{s_4\}$
t_{15}	62	a_5	n5	a_{10}	$\{s_5\}$

(b) V2

	FID		Text		
t_{21}	58	b_1	tx1	b_5	$\{r_1\}$
t_{22}	60	b_2	tx2	b_6	$\{r_2\}$
t_{23}	61	b_3	tx3	b_7	$\{r_3\}$
t_{24}	62	b_4	tx3	b_8	$\{r_4\}$

(c) V3

	FName		Type		
t_{31}	n1	a_6	gpcr	a_{11}	$\{s_1\}$
t_{32}	n2	a_7	gpcr	a_{12}	$\{s_2\}$
t_{33}	n3	a_8	lgic	a_{13}	$\{s_3\}$
t_{34}	n4	a_9	vgic	a_{14}	$\{s_4\}$
t_{35}	n5	a_{10}	vgic	a_{15}	$\{s_5\}$

(d) V4

	FID		FName		Text		
t_{41}	58	a_1	n1	a_6	tx1	b_5	$\{s_1, r_1\}$
t_{42}	60	a_3	n3	a_8	tx2	b_6	$\{s_3, r_2\}$
t_{43}	61	a_4	n4	a_9	tx3	b_7	$\{s_4, r_3\}$
t_{44}	62	a_5	n5	a_{10}	tx3	b_8	$\{s_5, r_4\}$

(e) $Q3(D)$

	Type		Text		
t_{q31}	gpcr	a_{11}	tx1	b_5	$\{s_1, r_1\}$
t_{q32}	gpcr	a_{12}	tx2	b_6	$\{s_2, r_1\}$
t_{q33}	lgic	a_{13}	tx1	b_5	$\{s_3, r_1\}$
t_{q34}	gpcr	a_{11}	tx2	b_6	$\{s_1, r_2\}$
t_{q35}	gpcr	a_{12}	tx1	b_5	$\{s_2, r_2\}$
t_{q36}	lgic	a_{13}	tx2	b_6	$\{s_3, r_2\}$

The validity of the view mappings must be considered for each tuple in the query result; however, this cannot be determined simply by reasoning over the where-provenance tokens. For example, for tuple $t_{q32} \in Q3(D)$,

Table 6: Query result $Q3(D)$ with provenance annotations and valid views

	Type		Text		why-provenance	covering sets
t_{q31}	gpcr	$a_{11} \to \{V3\}$	tx1	$b_5 \to \{V2, V4\}$	$\{s_1, r_1\}$	$V3 * V2 +^R V4 * V3$
t_{q32}	gpcr	$a_{12} \to \{V3\}$	tx2	$b_6 \to \{V2\}$	$\{s_2, r_1\}$	$V3 * V2$
t_{q33}	lgic	$a_{13} \to \{V3\}$	tx1	$b_5 \to \{V2\}$	$\{s_3, r_1\}$	$V3 * V2$
t_{q34}	gpcr	$a_{11} \to \{V3\}$	tx2	$b_6 \to \{V2\}$	$\{s_1, r_2\}$	$V3 * V2$
t_{q35}	gpcr	$a_{12} \to \{V3\}$	tx1	$b_5 \to \{V2\}$	$\{s_2, r_2\}$	$V3 * V2$
t_{q36}	lgic	$a_{13} \to \{V3\}$	tx2	$b_6 \to \{V2, V4\}$	$\{s_3, r_2\}$	$V3 * V2 +^R V4 * V3$

there are two where-provenance tokens, i.e. a_{12} for attribute "Type" and b_6 for attribute "Text", which come from tuple t_2 in the Family relation and t'_2 from the FamilyIntro relation. There exists a view mapping from view $V4$ to Q2, however its join condition (global predicate) under the mapping, $FID1 = FID2$, is not satisfied since the attribute FID in tuples t_2 and t'_2 do not match. However, if we just compare the where-provenance tokens, the token b_6 exists in both tuple $t_{q32} \in Q3(D)$ and the tuple $t_{42} \in V4(D)$, which leads to the incorrect conclusion that there is a valid view mapping for view $V4$ in tuple t_{q32}.

The validity of view mappings can be checked by reasoning over the result of the extended query, $Q3_{ext}$, as illustrated in Example 2. However, there is an alternative approach which reasons over why-provenance information. For example, the last column in Tables 5a-5e records the why-provenance annotations for each tuple in the materialized views and query result. If we consider tuple t_{q31} in $Q3(D)$ again, the corresponding why-provenance annotation also appears in tuple t_{41} of $V4(D)$, which means that the tuples from Family and FamilyIntro used to construct tuple t_{q31} can also work for the construction of tuple t_{41}. This implies that the join condition under the mapping for $V4$ can be satisfied by tuple t_{q31}, hence that the view mapping is valid for t_{q31}. However, for tuple t_{q32}, the why-provenance annotation is $\{s_2, r_1\}$ which does not exist in any tuple in $V4(D)$ and thus the view mapping of $V4$ is not valid.

After checking the validity of views for each tuple in the query result, a mapping is built between each where-provenance token a of a tuple t and the valid materialized views for t in which the token appears. For example, as Table 6 shows, for tuple t_{q31}, the token a_{11} is mapped to view $V3$ while the token b_5 is mapped to views $V2$ and $V4$. In order to cover the attributes "Type" and "Text", the views from each where-provenance token are combined. Notice that we can combine $V3$ with either $V2$ or $V4$ for tuples t_{q31} and t_{q36}, which leads to two *alternative* combinations. □

4 Implementing the Model

In this section, we describe an implementation of the model presented in Section 2 which generates citations for individual tuples in the query result. The citations can then be aggregated to compute a citation to any subset of the query result. The approach was implemented in a prototype, demonstrated in [3], and is called the Tuple Level Approach (TLA). TLA is similar to the *eager approach* to computing provenance [7], which also uses an extended query to carry an extra annotation column from the database. Note that in the TLA implementation, citation views are *not* materialized and all reasoning is done using the instance returned by the extended query discussed in Section 2.

In TLA, as much initial work is done as possible for reasoning about covering sets: each tuple is annotated with all views in which it potentially participates. This is done by expanding the schema of each base relation R with a single column (called the *view vector* column), and adding view V to the view vector for tuple t in R whenever R appears as a relational subgoal of V and t satisfies the local predicates of V. Checking global predicates in V is delayed until the user query is executed. Sample instances with view vectors for the $Family$ and $FamilyIntro$ relations are shown in Tables 7 and 8.

Table 7: Sample table for base relation Family

Family_id	Name	Type	View vector
58	n1	gpcr	V1,V3,V4
59	n2	gpcr	V1,V3,V4
60	n3	lgic	V1,V3,V4
61	n4	vgic	V1,V3,V4
62	n5	vgic	V1,V3,V4

Table 8: Sample table for base relation FamilyIntro

Family_id	Text	View vector
58	tx1	V2,V4
60	tx2	V2,V4
61	tx3	V2,V4
62	tx3	V2,V4

The approach is implemented in four steps:

Preprocessing step. When a query $Q : -B_Q$ is issued, we first calculate all *possible* view mappings according to Definition 1. View mappings will be filtered out in this step if they cannot cover any distinguished variables of Q (the second condition of Definition 2). This result is a set of view mappings $M(Q)$.

In order to derive valid view mappings for each tuple in the query result, Q is extended to include view vectors of every base relation occurring in B_Q and the boolean expressions of any *global predicates* under the view mappings $M(Q)$. The lambda variables under all view mappings in $M(Q)$ will also be included in the head of the extended query if they are not distinguished variables of Q.

Query execution step. The extended query, Q_{ext1}, is then executed over the database instance D yielding an instance $Q_{ext1}(D)$ over which the citation reasoning occurs.

Reasoning step. In first phase of citation reasoning, valid view mappings within each view vector are calculated for each tuple $t \in Q_{ext1}(D)$. A view mapping M from $M(Q)$ is valid for a view vector from relational subgoal R iff there exists a view annotation V in this view vector so that M can be derived from V, the global predicates under M are satisfied, and M covers at least one head variable in the query. In the second phase, combinations of valid view mappings from each view vector are considered to find the covering sets.

Population step To avoid the expense of calculating covering sets tuple by tuple, subsets of tuples that will *share* the same covering sets are found using the view vectors and boolean values of global predicates. Such tuples are then grouped; covering sets are calculated once per group and propagated to all tuples in the group. For example, in Table 9, the result tuples form a single group and therefore the covering set is only calculated once. This optimization leads to significant performance gains.

Example 5: Consider the following query:

$Q5(Type, Text) : -Family(FID1, Name, Type), FamilyIntro(FID2, Text), FID1 = FID2$

After deriving valid view mappings within each view vector, the resulting instance of the extended query $Q5_{ext1}(D)$ is shown in Table 9. It is worth noting that the boolean expression of the global predicate $FID = FID1$ from $V4$ is not evaluated since it matches the global constraint of $Q5$. Since each view in this example only has one view mapping, we use $V1, V2, \ldots, V4$ to denote the corresponding view mappings.[4] Note that $FID1, FID2$ are included in the head of $Q5_{ext1}$ since they are lambda variables. The final query result with covering sets is shown in Table 10. Parameterized views are instantiated by passing the parameter values (e.g. V1(59) indicates V1 for FID=59). Multiple covering sets for each tuple are combined with $+^R$ (alternative use). After projecting over the distinguished variables of $Q5$, the third tuple and the fourth tuple in $Q5_{ext1}(D)$ share

[4]In general, since a query may use the same relation more than once in a subgoal, a view may have multiple view mappings.

Table 9: Result of executing the extended query, $Q5_{ext1}(D)$

Type	Text	FID1	FID2	Valid view mappings from view vector 1	Valid view mappings from view vector 2
gpcr	$tx1$	58	58	V3	V2, V4
gpcr	$tx2$	60	60	V3	V2, V4
vgic	$tx3$	61	61	V3	V2, V4
vgic	$tx3$	62	62	V3	V2, V4

Table 10: The final result, $Q5(D)$, annotated with the covering sets

Type	Text	Covering sets
gpcr	$tx1$	$V3 * V2(58) +^R V4(58) * V3$
lgic	$tx2$	$V3 * V2(60) +^R V4(60) * V3$
vgic	$tx3$	$(V3 * V2(61) +^R V4(61) * V3) + (V3 * V2(62) +^R V4(62) * V3)$

the same $Type$ and $Text$ and thus the covering sets of the two tuples are combined with a new alternate use operator, denoted $+$.

Citations are then generated for each tuple using the covering sets, the citation query and function for each view in the covering sets, and the policies for *, $+^R$, and $+$.

For example, suppose $+^R$ is interpreted as min according to a cost function which calculates the total number of unmatched terms (distinguished variables or subgoals) between the views in a covering sets and the query. Then for each tuple in Table 10 the resulting covering set for each tuple will be $V3 * V2(FID)$. Furthermore, suppose $*, +$ and Agg are interpreted as join, union and intersection, respectively, and that the JSON-formatted citations for view $V3$ and each instantiated view $V2$ are as shown in Table 11. Then the citation for the covering set $V3 * V2(58)$ in the first tuple of the query result would be {ID: '58', author: ['Mark', 'Steve', 'Roger'], Committee: ['Poyner', 'Hay', 'Justo']}, which is the *join* of the citations from $V3$ and $V2(58)$. To construct a citation for the entire query result, the citations from each tuple ($V3 * V2(FID)$) in the query result are aggregated, yielding {ID: ['58', '60', '61', '62'], author: ['Mark', 'Steve', 'Roger', 'Jens', 'Rodrigo', 'John'], Committee: ['Hay', 'Poyner', 'Justo', 'Andrew', 'Leo', 'Joel']}. □

Overview of citation framework. The architecture of the citation framework is shown in Figure 2. The DBA specifies the citation views and policies for how the views are to be used in constructing a citation to a general query. When a user submits a query to the database, view definitions are mapped to it and their associated citations combined according to the policies; a citation is then returned to the author along with the query result. When a Reader later uses a citation to access the cited data, the process of citation generation is reverse engineered (see dashed arrows in Figure 2). The citation is dereferenced, obtaining the original query and the citation views that were used; note that versioning is an important component of the solution but is not discussed in this paper. A specialized version of this architecture was developed for *eagle-i* as a proof-of-concept [2].

Table 11: Citations for sample views

View	Result of citation function
V2(58)	{ID: '58', author: ['Mark'], Committee: ['Hay', 'Poyner']}
V2(60)	{ID: '60', author: ['Jens'], Committee: ['Andrew']}
V2(61)	{ID: '61', author: ['Rodrigo'], Committee: ['Leo']}
V2(61)	{ID: '62', author: ['John'], Committee: ['Joel']}
V3	{author: ['Steve', 'Roger'], Committee: ['Hay', 'Justo']}

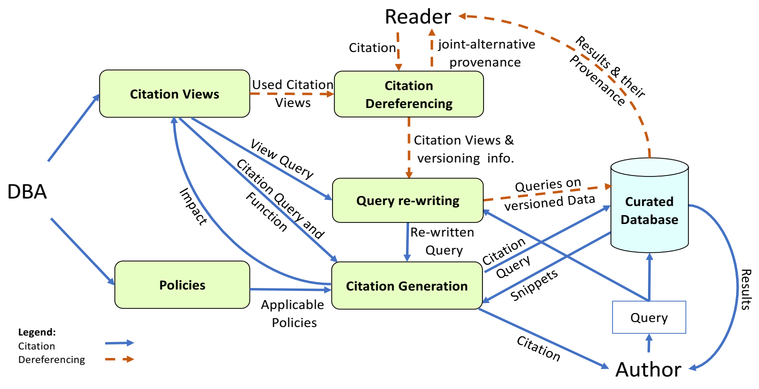


Figure 2: Citation Framework

5 Conclusions

In this paper, we address the problem of generating citations to the results of queries over data published in databases – *data citation* – and explore the connection to *data provenance*, in particular where-provenance and why-provenance. Our approach to data citation is based on *citation views*, as proposed in [9] and implemented in [3]. In this approach, citations are specified to some small number of frequent queries (e.g. web-page views of the database), and are used to construct citations for general queries.

Intuitively, a citation captures the origins of data by giving credit to its authors, i.e. the contributors and/or curators responsible for the data. When the citation views are select-project views over single relations and are partitioning, the view tuples which are used to create a tuple in the result can be simply calculated using where-provenance. However, in the general case where citation views may involve multiple relations (e.g. joins) and may overlap, where-provenance is no longer sufficient; one must also understand how a tuple in the result was constructed from the input tuples (why-provenance).

Rather than constructing and maintaining the materialized citation views as suggested by the connection above, our implementation of citation views reasons solely over the input query Q and the view definitions. It starts by annotating tuples in the base relations with the views in which they potentially participate, and determines which of the potential views are valid for result tuples by evaluating global predicates in $Q_{ext}(D)$. A number of clever optimizations are used to improve the efficiency of the approach, e.g. constructing citations for groups of tuples which share the same covering sets. Initial performance results (not discussed in this paper) show that citations can be generated for typical views and queries in a reasonable amount of time.

In future work, we would like to explore how to *integrate data citation within provenance-enabled* database systems. We would also like to study how existing *versioning* techniques can be adapted for data citation. Note that in this context versioning must be triggered when a user cites a data entry and only needs to record change on the cited data, thus interesting optimizations may be possible. Finally, we would like to explore how citations can be integrated into *data science environments*, in which queries are interleaved with analysis steps. This is difficult since provenance is not well understood in the context of machine learning algorithms.

Acknowledgments. This research was partially funded by NSF IIS 1302212, NSF ACI 1547360, and NIH 3-U01-EB-020954-02S1. The authors would like to thank Peter Buneman for conceiving the idea that data citation is a computational challenge, and Daniel Deutch and Tova Milo for helping formalize the connection to provenance.

References

[1] S. Abiteboul, R. Hull and V. Vianu. Foundations of Databases. *Addison-Wesley*, 1995.

[2] A. Alawini, L. Chen, S. B. Davidson, N. P. D. Silva and G. Silvello. Automating Data Citation: The eagle-i Experience. In *2017 ACM/IEEE Joint Conference on Digital Libraries, JCDL 2017, Toronto, ON, Canada, June 19-23, 2017*, pages 169-178, 2017.

[3] A. Alawini, S. B. Davidson, W. Hu and Y. Wu. Automating Data Citation in CiteDB. *PVLDB*, 10(12):1881-1884, 2017.

[4] P. Buneman, J. Cheney and S. Vansummeren. On the Expressiveness of Implicit Provenance in Query and Update Languages. In *Database Theory - ICDT 2007, 11th International Conference, Barcelona, Spain, January 10-12, 2007, Proceedings*, pages 209-223, 2007.

[5] P. Buneman, S. B. Davidson and J. Frew. Why data citation is a computational problem. *Commun. ACM*, 59(9):50-57, 2016.

[6] P. Buneman, S. Khanna and W. C. Tan. Why and Where: A Characterization of Data Provenance. In *Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings.*, pages 316-330, 2001.

[7] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in Databases: Why, How, and Where. *Found. Trends databases*, 1(4):379-474, Apr.2009.

[8] S. B. Davidson, P. Buneman, D. Deutch, T. Milo and G. Silvello. Data Citation: A Computational Challenge. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1-4, 2017.

[9] S. B. Davidson, D. Deutch, T. Milo and G. Silvello. A Model for Fine-Grained Data Citation. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, http://cidrdb.org, 2017.

[10] J. N. Foster, T. J. Green and V. Tannen. Annotated XML: queries and provenance. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 271-280, 2008.

[11] T. J. Green, G. Karvounarakis and V. Tannen. Provenance semirings. In *PODS*, pages 31-40, ACM, 2007.

[12] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270-294, 2001.

[13] G. Joshi-Tope, M. Gillespie, I. Vastrik, P. Eustachio, E. Schmidt, and et al. Reactome: A knowledge base of biological pathways. *Nucleic Acids Res*, 33 (DATABASE ISS.), 2005.

[14] D. Longo and J. Drazen. Data Sharing. *N Engl J Med [Internet]*, 374(3):276-277, 2016.

[15] A. Rajaraman, Y. Sagiv and J. D. Ullman. Answering Queries Using Templates with Binding Patterns. In *Proc. of the 14th Symposium on Principles of Database Systems*, pages 105-112, 1995.

[16] C. Southan, J. L. Sharman, H. E. Benson, E. Faccenda, A. J. Pawson, S. P. H. Alexander, O. P. Buneman, A. P. Davenport, J. C. McGrath, J. A. Peters, M. Spedding, W. A. Catterall, D. Fabbro, J. A. Davies and Nc-Iuphar. The IUPHAR/BPS Guide to PHARMACOLOGY in 2016: towards curated quantitative interactions between 1300 protein targets and 6000 ligands. *Nucleic Acids Research*, 44:1054-1068, 2016.

[17] C. Torniai, D. Bourges-Waldegg and S. Hoffmann. eagle-i: Biomedical research resource datasets. *Semant Web*, 6:139-146, 2015.

Provenance Analysis for Missing Answers and Integrity Repairs

Jane Xu, Waley Zhang, Abdussalam Alawini, and Val Tannen
Dept. Computer and Information Science
University of Pennsylvania
$\{$xuyuan, wzha, alawini, val$\}$@seas.upenn.edu

Abstract

Data provenance approaches track how the answer to a database query derive from input items; however, prior approaches used "positive" provenance and were not directly usable for explaining "expected" but missing answers. A similar problem arises with the failure of integrity constraints. Our perspective is to offer explanations via possible (minimal) repairs using provenance. This is useful for debugging, repairing, and cleaning databases. In this paper, we introduce a novel approach to this problem for both missing/erroneous answers and integrity failures. The approach uses recent advances in provenance for first-order model checking.

1 Introduction

When a user submits a query to a relational database, she often expects certain answers to appear in her query result. Sometimes, some answers may be missing or erroneous. For example, a user queries her university's database to list all courses offered in the Fall semester. However, she is not able to find her advisors course in the result, even though she knows that he will be teaching a course in the fall. Our perspective, related to *causality* [21], assumes that it is useful to provide the user with explanations via repairs—a set of *delete* and *insert* commands that would modify the database instance so that the query returns the expected answers.

Another important problem is that of integrity constraints (e.g., conditional functional dependencies, conditional inclusion constraints, etc., see [10]) failing in databases obtained via data integration/warehousing or via mining/crawling large amounts of information. Here, *repairs* can be similarly useful for explaining why integrity constraints fail. Applying such repairs belongs to the field of *data cleaning* (see below).

In this paper we present, principally via examples, a methodology for addressing explanations and repairs for both missing/erroneous answers and failure of integrity constraints. The commonality is based on both being expressible in (unrestricted) first-order logic (FOL). This approach has become possible after recent work by the last author and Erich Gräedel [13] (see Section 2).

The problem of missing answers has been studied both in the context of relational databases e.g. [3, 15, 21] (also known as why-not provenance and provenance of non-answers), and Datalog-defined computer networks [51, 52] (known there as negative provenance). Chapman and Jagadish [7] provided a model and definitions for

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

describing a data item that is not in the result set, and for asking why it does not appear in the result. Huang et al. in [18] developed a framework and algorithm for calculating the provenance of a single non-answer to a query. Herschel and Hernandez in [17] further expanded their work by having an algorithm that calculated why-not provenance for a set of missing answers including those for aggregation queries. In [16], Herschel and Hernandez continued expanding their model by providing support for queries that include selection, projection, join, union, aggregation and grouping (SPJUA). The larger context includes work on reverse query processing [20], the work of Meliou and Suciu on reverse data management [22, 23] and perhaps view update [9]. While steady progress has been made for positive queries, negation has always presented a challenge. Although we do not address aggregation, our work deals with general first-order queries (equivalent to the full relational algebra) while exploiting a particular way of expressing them: in stratified non-recursive Datalog with negation.

Database research related to integrity constraint repairs include the work of Arenas, Chomicki and Bertossi on consistent query answering [1, 2, 8]. Attribute-based repairs—repairs that only update some of the values of tuples' attributes in the database—have been studied by Bohannon et al. in [4]. A comprehensive treatment of the problem of data cleaning has been initiated by Fan et al. in [11, 12] with a focus on conditional dependencies. Kolahi and Lakshmanan in [19] showed that the problem of computing a minimal repair on databases that violate functional dependencies is NP-complete. Our discussion of repair costs (see Section 5) contributes to the line of work on *weighted repairs* in [5, 6].

Previous work on provenance (see [14] for a survey) used a framework based on commutative semirings for both expressing provenance (as polynomials) and applications such as cost, confidence, or access control. Following [13] (see Section 2) we use a semiring of *dual* polynomials with two kinds of indeterminates (provenance tokens), for positive and for negative facts to compute provenance of FOL sentences being checked in an FOL model. Next we compute explanations/repairs by solving the equation that makes such dual polynomials equal to the (semiring) 0. A solution/repair consists of a set of provenance tokens being made 0. We can then interpret these provenance tokens in semirings such as the Viterbi or tropical ones for cost or confidence associated with the repairs.

We regard this work as preliminary since it still relies on the a "closed world assumption" (CWA) to compute provenance in terms of negative tokens. This means that we only consider values that occur in the instance. We hope to expand our understanding of database repairs to a"open world assumption" (OWA) in the future via a more flexible approach that refer to data that we might need to insert into the database using labeled nulls.

This paper is organized as following. We start by reviewing the basic facts about computing provenance for FOL sentences and about dual polynomials (Section 2). Next, we show a couple of examples of repairs to missing answers and we describe a general algorithm for computing such repairs (Section 3). In Section 4, we give examples of repairs to a denial constraint and a tuple-generating constraint that goes beyond what has been studied in the literature so far. We discuss selecting optimal cost repairs in Section 5.

2 Dual Polynomials

Consider a finite relational vocabulary, $\mathscr{V}$.[1] From this vocabulary and a finite *non-empty* set A of *ground values* we construct the set Facts_A of all ground relational atoms (facts) $R(\mathbf{a})$, the set $\mathsf{NegFacts}_A$ of all negated facts $\neg R(\mathbf{a})$ and thus the set $\mathsf{Lit}_A = \mathsf{Facts}_A \cup \mathsf{NegFacts}_A$ of all *literals*, positive and negative facts, over $\mathscr{V}$ and A. By convention we will identify $\neg\neg R(\mathbf{a}) \equiv R(\mathbf{a})$) so the negation of a literal is again a literal.

Let $(K, +, \cdot, 0, 1)$ be a commutative semiring. Very roughly speaking, $0 \in K$ is intended to interpret false assertions, while an element $a \neq 0$ in K provides a "nuanced" interpretation for true assertions (call them

[1]For simplicity we omit constants from definitions, but we may make use of them in (counter)examples.

"a-true").

Next, K-interpretations will map literals to elements of K and are then extended to all formulae. Disjunction and existential quantification are interpreted by the addition operation of K. Conjunction and universal quantification are interpreted by the multiplication operation of K. For quantifiers, the finiteness of the universe A of ground values will be essential. For negation we use the well-known syntactic transformation to *negation normal form (NNF)*, denoted $\psi \mapsto \mathsf{nnf}(\psi)$. Note that $\mathsf{nnf}(\psi)$ is a formula constructed from literals (positive and negative facts) and equality/inequality atoms using just $\wedge, \vee, \exists, \forall$.

Definition 1: A K**-interpretation** is a mapping $\pi : \mathsf{Lit}_A \to K$. This is extended to FO formula given valuations $\nu : \mathsf{Vars} \to A$:

$$\begin{array}{rclrcl} \pi[\![R(\mathbf{x})]\!]_\nu & = & \pi(R(\nu(\mathbf{x}))) & \pi[\![\neg R(\mathbf{x})]\!]_\nu & = & \pi(\neg R(\nu(\mathbf{x}))) \\ \pi[\![x \;\mathsf{op}\; y]\!]_\nu & = & \text{if } \nu(x) \;\mathsf{op}\; \nu(y) \text{ then } 1 \text{ else } 0 & \pi[\![\varphi \wedge \psi]\!]_\nu & = & \pi[\![\varphi]\!]_\nu \cdot \pi[\![\psi]\!]_\nu \\ \pi[\![\varphi \vee \psi]\!]_\nu & = & \pi[\![\varphi]\!]_\nu + \pi[\![\psi]\!]_\nu & \pi[\![\exists x\, \varphi]\!]_\nu & = & \sum_{a \in A} \pi[\![\varphi]\!]_{\nu[x \mapsto a]} \\ \pi[\![\forall x\, \varphi]\!]_\nu & = & \prod_{a \in A} \pi[\![\varphi]\!]_{\nu[x \mapsto a]} & \pi[\![\neg\varphi]\!]_\nu & = & \pi[\![\mathsf{nnf}(\neg\varphi)]\!]_\nu \end{array}$$

The symbol op stands for either $=$ or $\neq$. As you can see from the definition, the equality and inequality atoms are interpreted in K as 0 or 1, i.e., their provenance is not tracked.

Let $X, \bar{X}$ be two disjoint sets together with a one-to-one correspondence $X \longleftrightarrow \bar{X}$. We denote by $p \in X$ and $\bar{p} \in \bar{X}$ two elements that are in this correspondence. We refer to the elements of $X \cup \bar{X}$ as **provenance tokens** as they will be used to label/annotate some of the "data", i.e., literals over some ground values, via the concept of K-interpretation that we defined previously. Indeed, if, as before, we fix a finite non-empty set A and consider $\mathsf{Lit}_A = \mathsf{Facts}_A \cup \mathsf{NegFacts}_A$ then we shall use X for Facts_A and $\bar{X}$ for $\mathsf{NegFacts}_A$. By convention, if we annotate $R(\mathbf{a})$ with the "positive" token p then the "negative" token $\bar{p}$ can only be used to annotate $\neg R(\mathbf{a})$, and vice versa. We refer to p and $\bar{p}$ as *complementary* tokens.

Further, we denote by $\mathbb{N}[X, \bar{X}]$ the quotient of the semiring of polynomials $\mathbb{N}[X \cup \bar{X}]$ by the congruence generated by the equalities $p \cdot \bar{p} = 0$ for all $p \in X$.[2] Observe that two polynomials $\mathfrak{p}, \mathfrak{q} \in \mathbb{N}[X \cup \bar{X}]$ are congruent iff they become identical after deleting from each of them the monomials that contain complementary tokens. Hence, the congruence classes in $\mathbb{N}[X, \bar{X}]$ are in one-to-one correspondence with the polynomials in $\mathbb{N}[X \cup \bar{X}]$ such that none of their monomials contain complementary tokens. We shall call these **dual-indeterminate polynomials** although we might often omit "-indeterminate" just use "dual polynomials". The following is the universality property of the semiring of dual polynomials:

Proposition 2: For any commutative semiring K and for any $f : X \cup \bar{X} \to K$ such that $\forall p \in X,\ f(p) \cdot f(\bar{p}) = 0$ there exists a unique semiring homomorphism $h : \mathbb{N}[X, \bar{X}] \to K$ such that $\forall x \in X \cup \bar{X}\ h(x) = f(x)$.

We note that $\mathbb{N}[X, \bar{X}]$ is "+-positive", that is, $\mathfrak{p} + \mathfrak{q} = 0$ implies $\mathfrak{p} = \mathfrak{q} = 0$, however, it has divisors of 0. Examples: $p \cdot \bar{p} = 0$; $(p + \bar{q})\bar{p}q = 0$; $(p\bar{q} + \bar{p}q)(pq + \bar{p}\bar{q}) = 0$. We also note that a dual polynomial is identically 0 iff it evaluates to 0 for every assignment $X \cup \bar{X} \to \{0, 1\}$ that sets complementary tokens to $0/1$ or $1/0$.

Definition 3: A **provenance-tracking** interpretation is a $\mathbb{N}[X, \bar{X}]$-interpretation $\pi : \mathsf{Lit}_A \to \mathbb{N}[X, \bar{X}]$ such that $\pi(\mathsf{Facts}_A) \subseteq X \cup \{0, 1\}$ and $\pi(\mathsf{NegFacts}_A) \subseteq \bar{X} \cup \{0, 1\}$.

Only provenance-tracking interpretations (annotations) are considered in the sections ahead.

[2] This is the same as quotienting by the ideal generated by the polynomials $p\bar{p}$ for all $p \in X$.

Email	*Sender*	*Day*	*Receiver*	
	Bob	Mon	Danny	
	Carla	Tue	Bob	
	Danny	Tue	Bob	
	Danny	Tue	Carla	y
	Bob	Mon	Carla	$\bar{p}$
	Bob	Tue	Carla	$\bar{q}$
	Danny	Mon	Carla	$\bar{z}$

Class	*Student*	*Course*	
	Bob	Bio	r
	Bob	Calc II	s
	Carla	Chem	t
	Carla	Calc II	u
	Danny	Bio	x
	Danny	Chem	p_2
	Bob	Chem	$\bar{p}_1$
	Carla	Bio	$\bar{w}$
	Danny	Calc II	$\bar{v}$

Figure 1: Database for missing answers examples

3 Missing Answers

We present a small example database containing a table of student email exchanges and another one with the classes they are taking. *Email* and *Class* are shown below. The tuples that are present in the instance under consideration are above the dashed lines and some of them are annotated (labeled) with positive provenance tokens. We apply the framework described in Section 2 with A as the active domain of this database. The tuples below the dashed lines are some of the closed-world-absent tuples from the instance and they are annotated with negative provenance tokens. This constitutes a provenance tracking interpretation. We hope that the reader will keep the use of the variables x, y as provenance tokens distinct from the use of the same as Datalog variables. To keep the picture less cluttered we only annotated the tuples that we actually use in the examples.

We consider the following **query:** return the pairs (sender, receiver) for the non-Monday emails where the receiver is either not taking Chem or is taking every class that the sender is taking. The query, denoted Q, is expressed in non-recursive Datalog with negation as follows (c is a variable!):

$$\begin{aligned} S(x,y) &\leftarrow \textit{Class}(x,c), \neg\textit{Class}(y,c), \textit{Class}(y,\text{“Chem”}) \\ Q(x,y) &\leftarrow \textit{Email}(x,d,y), \neg S(x,y), d \neq \text{“Mon”} \end{aligned}$$

In first-order logic notation we have:

$$\begin{aligned} S &= \{(x,y) \mid \exists c\, \textit{Class}(x,c) \wedge \neg\textit{Class}(y,c) \wedge \textit{Class}(y,\text{“Chem”})\} \\ Q &= \{(x,y) \mid \exists d\, \textit{Email}(x,d,y) \wedge \neg S(x,y) \wedge d \neq \text{“Mon”}\} \end{aligned}$$

Notice that the output of Q consists of (“Carla”, “Bob”) and (“Danny”, “Bob”). We are interested in why the tuple (“Bob”, “Carla”) is not in the set of outputs. Our approach is to find explanations for such questions by finding all the minimal repairs that would make Q(“Bob”, “Carla”) true. And we find these repairs by computing the dual polynomial $\mathfrak{p}$ corresponding to $\neg Q$(“Bob”, “Carla”) and then “solving” the equation $\mathfrak{p} = 0$, that is, finding which tokens can be set to 0 to insure that $\mathfrak{p}$ becomes 0. We have:

$$\neg Q(\text{“Bob”}, \text{“Carla”}) \;=\; \forall d\, \neg\textit{Email}(\text{“Bob”}, d, \text{“Carla”}) \;\vee\; S(\text{“Bob”}, \text{“Carla”}) \;\vee\; [d = \text{“Mon”}]$$

The computations of dual polynomials is done according to Definition 1. When it comes to quantifiers we need, in principle, to range over all the elements of the active domain for every variable. However, it is clear that it suffices to range over the elements that actually occur in the columns that correspond to the variables. For example, it suffices to take $d \in \{\text{“Mon”}, \text{“Tue”}\}$ and $c \in \{\text{“Bio”}, \text{“Calc II”}, \text{“Chem”}\}$. We call this the *typed active domain* approach and note that it is widely applicable in practice.

Now we calculate the polynomial corresponding to $\neg Q$(“Bob”, “Carla”):

$$\mathfrak{p} \;=\; (\bar{p} + \mathfrak{q} + [\text{“Mon”} = \text{“Mon”}])(\bar{q} + \mathfrak{q} + [\text{“Tue”} = \text{“Mon”}]) \;=\; (\bar{p} + \mathfrak{q} + 1)(\bar{q} + \mathfrak{q})$$

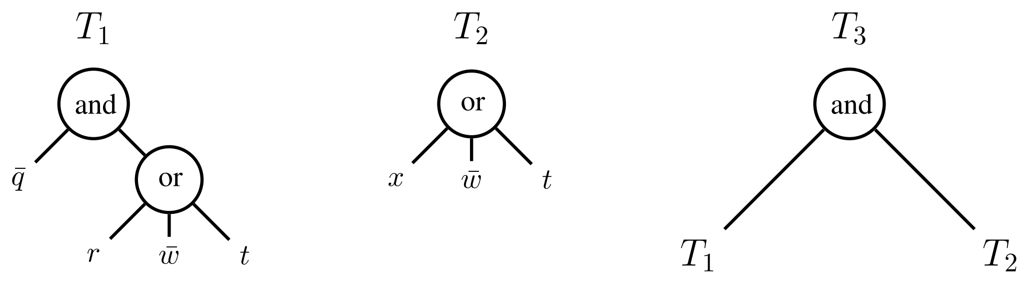


Figure 2: Repair and-or trees for missing answers Q(“Bob”, “Carla”) and Q(“Danny”, “Carla”)

where $\mathfrak{q}$ is the polynomial corresponding to S(“Bob”, “Carla”). We have:

$$\begin{aligned} S(\text{“Bob”}, \text{“Carla”}) &= \exists c\, \mathit{Class}(x, c) \wedge \neg \mathit{Class}(y, c) \wedge \mathit{Class}(y, \text{“Chem”}) \\ \mathfrak{q} &= r\bar{w}t + s\bar{u}t + p_1\bar{t}t \;=\; r\bar{w}t + s\bar{u}t \end{aligned}$$

Notice that the polynomial $\mathfrak{q}$ can be further simplified because our database already includes the tuple *Class*(“Carla”, “Calc II”) annotated with u, hence we can already assume that $\bar{u} = 0$. This avoids computing solutions that unnecessarily include $\bar{u} = 0$ (they would ask to insert a tuple that is already in the database). It follows that we can take $\mathfrak{q} \;=\; r\bar{w}t$.

For $\mathfrak{p} = 0$ one possible solution, by inspection, is $\{\bar{q} = t = 0\}$. Setting a negative token to 0 corresponds to invalidating a negative fact, i.e., inserting a tuple. Similarly, setting a positive token to 0 corresponds to deleting a tuple. Therefore this solution (that we found in an ad-hoc manner) corresponds to the repair that inserts $[+$*Email*(“Bob”, “Tue”, “Carla”)$]$ and that deletes $[-$*Class*(“Carla”, “Chem”)$]$ and thus brings (“Bob”, “Carla”) into the output of the query Q.

However, we need to systematically find *all* minimal repairs and to do this, anticipating dealing with more general queries, we proceed *without* substituting $\mathfrak{q}$ in $\mathfrak{p}$ and thus taking advantage of the structure of the query as a stratified, non-recursive Datalog$^{\neg}$ program. The first observation is that $\mathfrak{p}$ is a product of factors and in general it is possible for $\mathfrak{p}$ to be identically 0 (hence all minimal solutions would be empty) since $\mathbb{N}[X, \bar{X}]$ has divisors of 0. But this is not the case here since, for example, $\bar{p}\bar{q}$ occurs in $\mathfrak{p}$. Therefore, any solution makes at least one of the two factors 0. Note that each term in each factor of $\mathfrak{p}$ is either a single token or $\mathfrak{q}$ or a 1. The factor that contain a 1 cannot be made 0. Thus we must have $\bar{q} + \mathfrak{q} = 0$, hence, $\bar{q} = 0$ and $\mathfrak{q} = 0$. Notice that the stratified structure can lead to explanations for missing answers that may entail wrong answers for some of the subqueries (non-answer intensional predicates; in this example S(“Bob”, “Carla”)). Thus, the missing answers and the wrong answers problems are intricately related for this query language.

Since S is an intensional predicate we must continue until we find the repairs to the actual database. $\mathfrak{q} = 0$ has three (minimal) solutions; $\{t = 0\}$ is one, the others are $\{r = 0\}$ and $\{\bar{w} = 0\}$. This yields three minimal solutions for $\mathfrak{p} = 0$: $\{\bar{q} = t = 0\}$, $\{\bar{q} = r = 0\}$ and $\{\bar{q} = \bar{w} = 0\}$. The reader can check that the repairs corresponding to each of these solutions, for example, insert $[+$*Email*(“Bob”, “Tue”, “Carla”)$]$ and insert $[+$*Class*(“Carla”, “Bio”)$]$ for the third solution, also bring (“Bob”, “Carla”) into the output of Q. All the minimal solutions are nicely captured by the and-or tree T_1 in Figure 2 (on the leaves of these trees we have partial solutions $token = 0$ which are represented, for simplicity, just by $token$.

Let us now find the repairs for another answer missing from the output of Q, namely (“Danny”, “Carla”). The dual polynomial corresponding to $\neg Q$(“Danny”, “Carla”) is $\mathfrak{r} = (\bar{z} + \mathfrak{s} + 1)(\bar{y} + \mathfrak{s})$ where $\mathfrak{s} = x\bar{w}t + v\bar{u}t + p_2\bar{t}t = x\bar{w}t + v\bar{u}t$. As before, we can simplify the polynomials by setting $\bar{y} = \bar{u} = 0$ since the tuples annotated with y and u are already in the database, obtaining $\mathfrak{r} = (\bar{z} + \mathfrak{s} + 1)x\bar{w}t$. Although it leads to the same result, we could also have set $v = 0$ since the tuple annotated with $\bar{v}$ is *not* in the database (and a repair that contained $v = 0$ would have asked that we delete a tuple already absent. All the solutions to $\mathfrak{r} = 0$ are captured by the and-or tree T_2 in Figure 2.

What if we wish to repair *both* Q(“Bob”, “Carla”) and Q(“Danny”, “Carla”)? This would need the dual polynomial corresponding to $\neg[Q(\text{“Bob”}, \text{“Carla”}) \wedge Q(\text{“Danny”}, \text{“Carla”})]$, i.e., $\mathfrak{p} + \mathfrak{r}$. The solutions to $\mathfrak{p} + \mathfrak{r} = 0$ are captured by the and-or tree T_3 in the same figure. Observe that T_1 describes 3 minimal repairs, and so does T_2. Now T_3 describes $3 \times 3 = 9$ repairs, but *only* 3 of them are minimal: $\{\bar{q} = r = x = 0\}$, $\{\bar{q} = \bar{w} = 0\}$ and $\{\bar{q} = t = 0\}$.

In general, our approach is not guaranteed to compute only minimal repairs, as can be seen by considering $Q'(y) \leftarrow Q(x, y)$ and the missing answer Q'(“Carla”). However, we can prove that all the minimal repairs are included among those computed by the algorithm shown below. It is also clear that redundant, non-minimal repairs, can be pruned away.

Algorithm Next we present a procedure for finding repairs (including all the minimal repairs) for missing and/or wrong answers for queries expressed in non-recursive Datalog with negation (and recall that any first-order query is logically equivalent with one such). The algorithm takes advantage of the stratified structure of the language to produce a compact and-or repair tree.

The algorithm's input is a triple $(R, \mathbf{a}, \mu)$ where R is a predicate, $\mathbf{a}$ is tuple of constants from the active domain of the instance (R and $\mathbf{a}$ have the same arity), and μ is a ”mode”parameter that can have two values: $\mathsf{missing}$ and wrong. Let Q be the answer predicate of the query. There may be multiple Datalog rules whose head can be instantiated to $Q(\mathbf{a})$: let's denote these instatiations by $Q(\mathbf{a}) \leftarrow B_i(\mathbf{a}, \mathbf{y}_i)$ for $i = 1, \ldots, k$. Then, in FOL,

$$Q(\mathbf{a}) = [\exists \mathbf{y}_1\, B_1(\mathbf{a}, \mathbf{y}_1)] \vee \cdots \vee [\exists \mathbf{y}_k\, B_k(\mathbf{a}, \mathbf{y}_k)]$$

where each B_i is a *conjunction* of positive or negative atoms(extensional or intensional), equalities, or inequalities. Moreover,in NNF,

$$\neg Q(\mathbf{a}) = [\forall \mathbf{y}_1\, \overline{B}_1(\mathbf{a}, \mathbf{y}_1)] \wedge \cdots \wedge [\forall \mathbf{y}_k\, \overline{B}_k(\mathbf{a}, \mathbf{y}_k)]$$

where each $\overline{B}_i$ (in NNF) is a *disjunction* of negative or positive atoms(extensional or intensional), inequalities, or equalities.

On input $(Q, \mathbf{a}, \mathsf{missing})$, the algorithm returns the repair and-or tree corresponding to the missing answer $Q(\mathbf{a})$. It does so by first computing the dual polynomial $\mathfrak{p}$ corresponding to the first-order sentence $\mathsf{nnf}(\neg Q(\mathbf{a}))$. This computation includes simplifications like those mentioned in our examples:

- If $\bar{u}$ occurs in the polynomial and u annotates a tuple already in the instance, then replace $\bar{u}$ with 0.
- If v occurs in the polynomial and v annotates a tuple that is not in the database then replace u with 0.

Next the algorithm finds solutions to $\mathfrak{p} = 0$. A solution is a set of provenance tokens, negative or positive, all set to 0. Since intensional predicates may occur in $\mathsf{nnf}(\neg Q(\mathbf{a}))$ we generate fresh provenance tokens to annotate temporarily these intentional literals. Note that $\mathfrak{p}$ is a product of factors, each factor being a sum of (positive or negative, extensional or intensional) provenance tokens or 1's. Moreover, we can prove that $\mathfrak{p}$ cannot be identically 0. Indeed, by the range-restriction condition, each rule must contain at least one positive literal. This means that each factor in $\mathsf{nnf}(\neg Q(\mathbf{a}))$ must contain at least one negative token and the product of all these negative tokens occurs as a monomial in $\mathfrak{p}$. When constructing the and-or tree of solutions to $\mathfrak{p} = 0$, we collect trees that are either leaves or, by setting an intensional token to 0, are obtained from recursive call to the algorithm.

On input $(Q, \mathbf{a}, \mathsf{wrong})$, the algorithm returns the repair and-or tree corresponding to the wrong answer $Q(\mathbf{a})$. It does so by computing the dual polynomial $\mathfrak{q}$ corresponding to the first-order sentence $\mathsf{nnf}(Q(\mathbf{a}))$, simplified as above and with fresh intensional tokens as above. Solutions are found, again, by setting $\mathfrak{q} = 0$. Note that $\mathfrak{p}$ is a sum of terms, each term being a product of tokens (i.e., a monomial).

In stating the algorithm we use the notations $\bar{p} \sim L_1$, respectively $p \sim L_2$, to mean that the negative token $\bar{p}$ annotates the negative literal L_1, respectively the positive token p annotates the positive literal L_2.

All this results in Algorithm **??**.

Size of repair trees We can show that the number of minimal repairs is in general exponential in the size of the active domain. However, the and-or trees that we compute offer a compact representation of all these repairs. Indeed, let n be an upper bound on the size of the active domain and let q be an upper bound on the size of the query (expressed in non-recursive Datalog with negation). The algorithm calculates two kinds of polynomials:

- For missing tuples it calculates polynomials as products of factors. Each factor's size is bounded by the size of rules is $O(q)$. The number of factors is bounded by the number of rules times the number of iterations of universally quantified variables in each rule, therefore is $O(q\, n^q)$. Thus, the resulting polynomial has size $O(q^2\, n^q)$.
- For wrong tuples it calculates polynomials as sums of monomials. Since there are at most q existentially quantified variables in each rule, and there are at most q rules there are at most $q\, n^q$ monomials in this polynomial. Each monomial's size is bounded by q so the resulting polynomial also has size $O(q^2\, n^q)$.

It follows that in the repair tree each node has $O(q^2\, n^q)$ children. The height of the tree is bounded by the number of rules, hence by q. Therefore the repair tree has size $O(q^{2q}\, n^{q^2})$, thus of polynomial data complexity.

Essentially the same argument shows that Algorithm 1 runs in polynomial time (data complexity).

4 Repairing Integrity Constraints

Since most integrity constraints used in databases are expressible in first order logic, our approach will also work to "repair" them. Here is an example. Consider the table *Class1* in Figure **??**. Notice that Danny has decided to take another course–Calc II. Uh oh… adding the course wasn't a good idea. Danny gets overloaded with work and finds himself failing Chemistry. The university notices this and puts Danny on probation (of a ruthless kind!), so Danny can now take only at most 2 courses at a time. Consequently, our database now breaks an integrity constraint. How do we fix it?

The integrity constraint "NO STUDENT CAN TAKE THREE OR MORE DISTINCT COURSES" can be written

$$I \;=\; \forall s \;\; \neg(\exists x, y, z \;\; \mathit{Class1}(s,x) \,\wedge\, \mathit{Class1}(s,y) \,\wedge\, \mathit{Class1}(s,z) \,\wedge\, x \neq y \,\wedge\, x \neq z \,\wedge\, y \neq z)$$

This is an example of a *denial constraint* [2]. Of course, it fails in the database *Class1*. To find out the reasons it fails as well as to obtain repairs to the database we compute the dual provenance polynomial of $\neg I$. We have

$$\mathsf{nnf}(\neg I) \;=\; \exists s, x, y, z \;\; \mathit{Class1}(s,x) \,\wedge\, \mathit{Class1}(s,y) \,\wedge\, \mathit{Class1}(s,z) \,\wedge\, x \neq y \,\wedge\, x \neq z \,\wedge\, y \neq z$$

```
Input  : $\mu$ tells us whether $R(\mathbf{a})$ is missing or is wrong.
Output: An and-or tree of updates
if $R$ is an extensional predicate then
    if $\mu = \mathsf{missing}$ then
        Return a single node tree labeled with token $\bar{p} \sim \neg R(\mathbf{a})$.
    end
    if $\mu = \mathsf{wrong}$ then
        Return a single node tree labeled with token $p \sim R(\mathbf{a})$.
    end
end
if $R$ is an intensional predicate then
    $\mathscr{C} := \emptyset$.
    if $\mu = \mathsf{missing}$ then
        Compute, as a product of factors, the polynomial $\mathfrak{p}$ corresponding to $\mathsf{nnf}(\neg R(\mathbf{a}))$.
        for each factor $f$ in $\mathfrak{p}$ do
            $\mathscr{T} := \emptyset$.
            for each term $t$ in the factor $f$ do
                if $t \sim$ positive literal $S(\mathbf{b})$ then
                    $\mathscr{T} := \mathscr{T} \cup \{RepTree(S, \mathbf{b}, \mathsf{wrong})\}$.
                end
                if $t \sim$ negative literal $\neg S(\mathbf{b})$ then
                    $\mathscr{T} := \mathscr{T} \cup \{RepTree(S, \mathbf{b}, \mathsf{missing})\}$.
                end
                if $t \sim 1$ then
                    $\mathscr{T} := \mathscr{T} \cup \{"never - zero"\}$.
                end
            end
            if $"never - zero" \notin \mathscr{T}$ then
                Build an and-tree $T$ with the trees collected in $\mathscr{T}$ as children of the root.
                $\mathscr{C} := \mathscr{C} \cup \{T\}$.
            end
        end
        Build an or-tree $T$ with the trees collected in $\mathscr{C}$ as children of the root. Return with $T$.
    end
    if $\mu = \mathsf{wrong}$ then
        Compute, as a sum of terms, the polynomial $\mathfrak{q}$ corresponding to $\mathsf{nnf}(R(\mathbf{a}))$.
        for each term $t$ in $\mathfrak{q}$ do
            $\mathscr{T} := \emptyset$.
            for each factor $f$ in the term $t$ do
                if $f \sim$ positive literal $S(\mathbf{b})$ then
                    $\mathscr{T} := \mathscr{T} \cup \{RepTree(S, \mathbf{b}, \mathsf{wrong})\}$.
                end
                if $f \sim$ negative literal $\neg S(\mathbf{b})$ then
                    $\mathscr{T} := \mathscr{T} \cup \{RepTree(S, \mathbf{b}, \mathsf{missing})\}$.
                end
            end
            Build an or-tree $T$ with the trees collected in $\mathscr{T}$ as children of the root.
            $\mathscr{C} := \mathscr{C} \cup \{T\}$.
        end
        Build an and-tree $T$ with the trees collected in $\mathscr{C}$ as children of the root. Return with $T$.
    end
end
```

Algorithm 1: $RepTree(R, \mathbf{a}, \mu)$ builds a repair tree

Class1	*Student*	*Course*	
	Bob	Bio	
	Bob	Calc II	
	Carla	Chem	
	Carla	Calc II	
	Danny	Bio	r
	Danny	Chem	s
	Danny	Calc II	t
	Bob	Chem	$\bar{u}$
	Carla	Bio	$\bar{v}$

Admin	*Course*	*Dept*	
	Calc II	Math	p
	Chem	Sci	q
	Bio	Sci	r
	Calc II	Sci	$\bar{t}$
	Chem	Math	$\bar{u}$
	Bio	Math	$\bar{v}$

Class2	*Student*	*Course*	
	Bob	Bio	w
	Bob	Calc II	x
	Danny	Bio	w_1
	Danny	Chem	y_1
	Bob	Chem	$\bar{y}$
	Danny	Calc II	$\bar{z}$

Figure 3: Tables for the integrity constraint examples

The corresponding dual polynomial $\mathfrak{p}$ is a sum. Using the *typed* active domain approach, this sum will have, in principle, $3 \times 3 \times 3 \times 3 = 81$ terms. However, most of the inequalities yield 0, namely when x, y, z are not assigned distinct values. We are left with $3 \times (3!)$ terms. Moreover, $2 \times (3!)$ of these are monomials containing $\bar{u}$ or $\bar{v}$ which we can set to 0 (because Bob and Carla are not breaking the constraint :). We are left with $\mathfrak{p} = 3rst$ which yields a repair **or**-tree with three leaves. As expected, we find that the integrity constraint is broken because all three tuples annotated r, s, or t are in the database. Also as expected, we have three possible minimal repairs, each corresponding to deleting one of the three tuples.

The second example we will show in this section uses the tables *Admin* and *Class2* in Figure **??**. It also uses the constraint "IF A STUDENT IS TAKING MORE THAN ONE CLASS, ALL OF THEIR CLASSES MUST BE ADMINISTERED BY THE SAME DEPARTMENT". We can express this constraint as:

$$J = \forall x, y\ [\exists s\ \textit{Class2}(s, x) \wedge \textit{Class2}(s, y) \wedge x \neq y] \rightarrow [\exists t\ \textit{Admin}(x, t) \wedge \textit{Admin}(y, t)]$$

This is an example of a tuple-generating constraint, even more general than the conditional inclusion constraints mentioned in [2, 10]. It does fail in the database above since Bob saw fit to take both Bio and Calc II. As with the denial constraint we considered above, we find the reasons for failure and possible repairs by computing the dual polynomial corresponding to $\mathsf{nnf}(\neg J)$. We have

$$\mathsf{nnf}(\neg J) = \exists x, y\ [\exists s\ \textit{Class}(s, x) \wedge \textit{Class}(s, y) \wedge x \neq y] \wedge [\forall t\ \neg\textit{Admin}(x, t) \vee \neg\textit{Admin}(y, t)]$$

We compute the dual provenance polynomial of $\neg J$. Again, we use the *typed* active domain approach and we notice that the first factor will always be zero if we choose x and y to both be the same course. The result is a sum of polynomials, which we denote $\mathfrak{q}$:

$$\mathfrak{q} = (zw_1 + xw)(\bar{t} + \bar{r})(\bar{p} + \bar{v}) + (y_1 z + yx)(\bar{q} + \bar{t})(\bar{u} + \bar{p}) + (y_1 w_1 + yw)(\bar{q} + \bar{r})(\bar{u} + \bar{v})$$

However, given which tuples are already/are not in the database we can simplify this polynomial by setting $y = z = 0$ and $\bar{p} = \bar{q} = \bar{r} = 0$ and we are left with $\mathfrak{q} = xw\bar{t}\bar{v}$. To fix the constraint, we try to make $\mathfrak{q} = 0$. which yields a repair **or**-tree with four leaves. These correspond to four different minimal repairs (two of them are deletions and the others insertions): $[-\textit{Class}$("Bob", "Calc II")], or $[-\textit{Class}$("Bob", "Bio")], or $[+\textit{Admin}$("Calc II", "Sci")], or $[+\textit{Admin}$("Bio", "Math")].

5 Costing Repairs

For costs we use the *tropical semiring* $\mathbb{T} = (\mathbb{R}_+^\infty, \min, +, \infty, 0)$. Its elements and operations appear in *min-cost* interpretations (e.g., shortest paths). Here we are going to show how to use it to select repairs that are in some sense optimal.

Recall the integrity constraint J from Section 4. We obtained all possible minimal repairs by computing the dual polynomial $\mathfrak{q}$ corresponding to $\mathsf{nnf}(\neg J)$ and obtained four repairs: $\{x = 0\}, \{w = 0\}, \{\bar{t} = 0\}, \{\bar{v} = 0\}$. To choose among them, we now compute the dual polynomial $\mathfrak{r}$ corresponding to $\mathsf{nnf}(J)$:

$$\mathfrak{r} = [(\bar{z} + \bar{w}_1)(\bar{x} + \bar{w}) + tr + pv] \cdot [(\bar{y}_1 + \bar{z})(\bar{y} + \bar{x}) + qt + up] \cdot [(\bar{y}_1 + \bar{w}_1)(\bar{y} + \bar{w}) + qr + uv]$$

Next we apply all the simplifications allowed by the database *except* those involving the tokens that appear in the repairs, namely $\bar{w}_1 = \bar{y}_1 = u = 0$. We obtain: $\mathfrak{r} = [\bar{z}(\bar{x} + \bar{w}) + tr + pv] \cdot [\bar{z}(\bar{y} + \bar{x}) + qt] \cdot [qr]$.

Our definition of optimality[3] is to find the repair that minimizes the cost of $\mathfrak{r}$ when interpreted in $\mathbb{T}$, given costs for the remaining tokens in r, say $cost(\bar{z}) = \alpha$, $cost(r) = \beta$, $cost(p) = \gamma$, $cost(\bar{y}) = \delta$, $cost(q) = \iota$, and assuming, in turn, each of the four repairs have a cost also, let's say that a deletion costs λ and an insertion costs μ, where $\alpha, \beta, \gamma, \delta, \iota, \lambda, \mu \in \mathbb{R}_+^\infty$. Thus we have four different assignments into $\mathbb{T}$: $[\bar{x} \mapsto \lambda, \bar{w} \mapsto \infty, t \mapsto 0, v \mapsto 0]$, $[\bar{x} \mapsto \infty, \bar{w} \mapsto \lambda, t \mapsto 0, v \mapsto 0]$, $[\bar{x} \mapsto \infty, \bar{w} \mapsto \infty, t \mapsto \mu, v \mapsto 0]$, $[\bar{x} \mapsto \lambda, \bar{w} \mapsto \infty, t \mapsto 0, v \mapsto \mu]$ one for each of the four repairs. We obtain:

Repair	Cost
[−*Class2*("Bob", "Calc II")]	$\min(\alpha + \lambda, \beta, \gamma) + \min(\alpha + \min(\delta, \lambda), \iota) + \iota + \beta$
[−*Class2*("Bob", "Bio")]	same
[+*Admin*("Calc II", "Sci")]	$\min(\mu + \beta, \gamma) + \min(\alpha + \delta, \iota + \mu) + \iota + \beta$
[+*Admin*("Bio", "Math")]	$\gamma + \mu + \alpha + \delta + \iota + \beta$

For example, taking $\lambda = \mu = \alpha = \beta = \gamma = \delta = \iota = 10$ will result in the first (or second) repair to be cheapest.

6 Conclusions

In this paper, we introduced a novel approach for computing all possible repairs for missing and wrong answers to a query on a database instance. Our algorithm is based on a new framework for reasoning over the provenance of non-answers under the CWA for queries expressible in first-order logic, including negation. This framework treats relational queries as logical structures (unions and conjunctions of statements) and applies algebraic operations to these logical structures, allowing the use of dual polynomials as a way to represent provenance.

Our approach is not limited to repairing queries with missing and wrong answers; it can also repair a database instance that does not satisfy certain integrity constraints. Since integrity constraints can be expressed in first-order logic, our algorithm is able to determine where constraints are broken. In cases of missing answers, wrong answers, and broken integrity constraints, our algorithm is able to suggest repairs for the database in the form of a set of insertions and deletions of tuples.

Our use of dual polynomials rather than boolean expressions provides a way to weigh certain tuples over others by assigning tokens different weights. Questions such as finding a minimal cost solution from an and-or-tree are NP-hard (for example by reduction from vertex cover). In the future, we aim to explore approximation techniques for these questions. Additionally, we would like to expand our approach to handle open world assumption. Currently, our approach fixes databases by introducing tuples consisting only of values from our active domain. However, it is unrealistic to always expect repairs to be derived from within the database. Thus, it is important for us to move beyond CWA and develop a working framework in OWA.

[3] Other definitions are possible and we plan to compare them in future work.

Acknowledgments. The authors would like to thank Sanjeev Khanna and Tova Milo for helpful discussions. Our work was partially supported by Penn's Center for Undergraduate Research and Fellowships as well as by NSF grants 1302212 and 1547360 and by NIH grant U01EB02095401.

References

[1] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, pages 68–79, 1999.

[2] L. Bertossi. Database repairing and consistent query answering. *Synthesis Lectures on Data Management*, 3(5):1–121, 2011.

[3] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014)*, Cologne, 2014. USENIX Association.

[4] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 143–154, New York, NY, USA, 2005. ACM.

[5] D. Burdick, R. Fagin, P. G. Kolaitis, L. Popa, and W. Tan. A declarative framework for linking entities. *ACM Trans. Database Syst.*, 41(3):17:1–17:38, 2016.

[6] D. Burdick, R. Fagin, P. G. Kolaitis, L. Popa, and W. C. Tan. Expressive power of entity-linking frameworks. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, pages 10:1–10:18, 2017.

[7] A. Chapman and H. V. Jagadish. Why not? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 523–534, New York, NY, USA, 2009. ACM.

[8] J. Chomicki. *Consistent Query Answering: Five Easy Pieces*, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[9] U. Dayal and P. A. Bernstein. On the updatability of relational views. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4*, VLDB '78, pages 368–377. VLDB Endowment, 1978.

[10] W. Fan and F. Geerts. *Foundations of Data Quality Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[11] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2):6:1–6:48, June 2008.

[12] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *Proc. VLDB Endow.*, 3(1-2):173–184, Sept. 2010.

[13] E. Grädel and V. Tannen. Semiring provenance for first-order model checking. *CoRR*, abs/1712.01980, 2017.

[14] T. J. Green and V. Tannen. The semiring framework for database provenance. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 93–99, 2017.

[15] M. Herschel. A hybrid approach to answering why-not questions on relational query results. *J. Data and Information Quality*, 5(3):10:1–10:29, Mar. 2015.

[16] M. Herschel and M. A. Hernández. Explaining missing answers to spjua queries. *Proc. VLDB Endow.*, 3(1-2):185–196, Sept. 2010.

[17] M. Herschel, M. A. Hernández, and W.-C. Tan. Artemis: A system for analyzing missing answers. *Proc. VLDB Endow.*, 2(2):1550–1553, Aug. 2009.

[18] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[19] S. Kolahi and L. V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory*, ICDT '09, pages 53–62, New York, NY, USA, 2009. ACM.

[20] D. Kossmann, E. Lo, and C. Binnig. Reverse query processing. *2007 IEEE 23rd International Conference on Data Engineering*, 00:506–515, 2007.

[21] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. Why so? or why no? functional causality for explaining query answers. *CoRR*, abs/0912.5340, 2009.

[22] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PVLDB*, 4(12):1490–1493, 2011.

[23] A. Meliou and D. Suciu. Tiresias: the database oracle for how-to queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 337–348, 2012.

[24] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering why-not queries in software-defined networks with negative provenance. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets-XII)*, Nov. 2013.

[25] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems negative provenance. In *Proceedings of ACM SIGCOMM 2014*, Aug. 2014.

GProM - A Swiss Army Knife for Your Provenance Needs

Bahareh S. Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, Qitian Zeng
Illinois Institue of Technology

Abstract

We present an overview of GProM, a generic provenance middleware for relational databases. The system supports diverse provenance and annotation management tasks through query instrumentation*, i.e., compiling a declarative frontend language with provenance-specific features into the query language of a backend database system. In addition to introducing GProM, we also discuss research contributions related to GProM including the first provenance model and capture mechanism for transaction provenance, a unified framework for answering why- and why-not provenance questions, and provenance-aware query optimization. Furthermore, by means of the example of post-mortem debugging of transactions, we demonstrate how novel applications of provenance are made possible by GProM.*

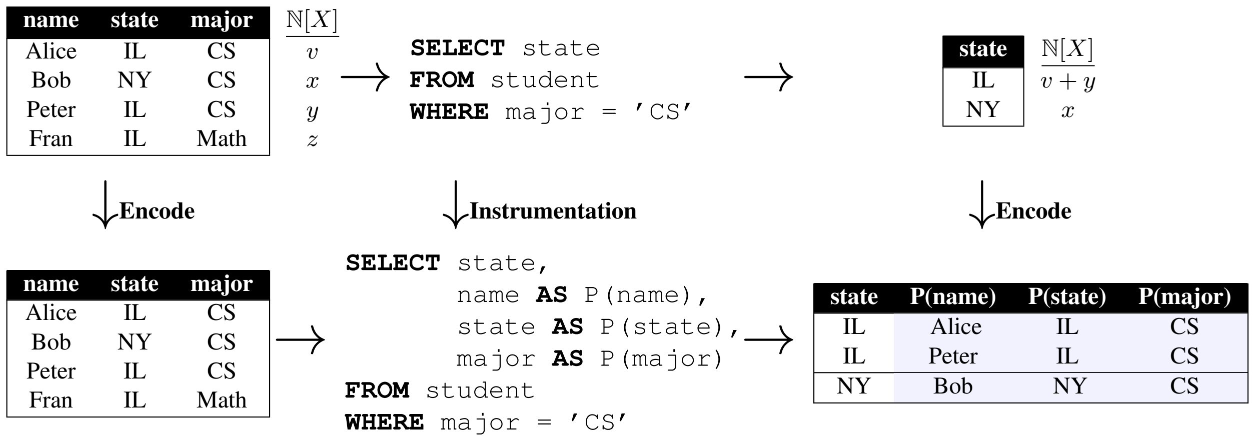
1 Introduction

Provenance, information about the origin of data and the queries and/or updates that produced it, is critical for debugging queries and transactions, auditing, establishing trust in data, and many other use cases. For example, consider a relation storing employee salaries. The relation is subjected to complex transactional updates such as calculating tax, applying tax deductions, multipling rates with working hours, and so on. How can we know whether the information in the current version of the relation is correct? If one employee's salary is incorrect, how do we know which update(s) or data caused that error? Data provenance, by providing a full record of the derivation history of data, makes it possible to identify the causes of such errors.

A persistent challenge in database provenance research has been to build efficient provenance-aware databases. That is, to design and implement systems that automatically capture provenance information for database operations and allow this information to be queried. In this work, we give an introduction to GProM (Generic Provenance Middleware), a system that enriches database backends with support for provenance. The system is available as open source software at `https://github.com/IITDBGroup/gprom`. At its core, GProM is a compiler that translates a frontend language (e.g., SQL with new language constructs for requesting and managing provenance) into queries expressed in the language of a database backend that generate a relational encoding of data annotated with provenance. Below we briefly discuss some of GProM's unique features.

- **Exploiting backend databases for provenance capture and storage**: GProM represents provenance in the data model of the backend database. To capture provenance for an operation, the system constructs a query expressed in the language of the database backend which returns this type of provenance encoding. This technique, which we refer to as *instrumentation*, enables us to exploit the advanced storage and query execution capabilities of modern database systems.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



name	state	major	$\mathbb{N}[X]$
Alice	IL	CS	v
Bob	NY	CS	x
Peter	IL	CS	y
Fran	IL	Math	z

```
SELECT state
FROM student
WHERE major = 'CS'
```

state	$\mathbb{N}[X]$
IL	$v + y$
NY	x

name	state	major
Alice	IL	CS
Bob	NY	CS
Peter	IL	CS
Fran	IL	Math

```
SELECT state,
       name AS P(name),
       state AS P(state),
       major AS P(major)
FROM student
WHERE major = 'CS'
```

state	P(name)	P(state)	P(major)
IL	Alice	IL	CS
IL	Peter	IL	CS
NY	Bob	NY	CS

Figure 1: Provenance instrumentation example: compute provenance polynomials for a query

- **On demand provenance capture for a large class of operations**: GProM supports provenance capture for queries, updates, and transactions. To the best of our knowledge it is the only system that can capture provenance for transactions. In contrast to many other systems which capture provenance eagerly for all operations no matter whether this provenance is needed or not, GProM only captures provenance if it is explicitly requested by a user or application.
- **Treating provenance requests as queries**: The user interacts with GProM through a declarative frontend language enriched with language constructs for requesting and managing provenance. Provenance requests are treated as queries which allows them to be combined with other language constructs. Thus, the full expressive power of the frontend language is available for querying provenance.
- **Low-overhead and non-invasive**: GProM was designed to minimize the performance impact for operations when no provenance is requested. Obviously, it would be impossible to reconstruct provenance for past operations unless some information is maintained. GProM relies on the temporal and auditing logging capabilities supported by many DBMS to capture sufficient information to be able to reconstruct provenance for past queries, transactions, and updates on demand. This approach has the advantage of being non-invasive, i.e., no changes to an application's SQL code are required to enable provenance capture.
- **Extensibility**: GProM was designed from the ground up with extensibility in mind - support for new provenance models, database backends, frontend languages, and optimizations can be added with ease. This has enabled us to support a large number of diverse provenance and annotation management tasks.

The remainder of this paper is organized as follows. We explain the instrumentation approach underlying GProM in Section 2. In Section 3, we give a more technical introduction to GProM. Section 4 covers major contributions to provenance research related to GProM. We discuss post-mortem transaction debugging, one of the novel applications of provenance made possible by GProM, in Section 5. We conclude in Section 6. The research on GProM is enabled by the many fundmental contributions to provenance research made by the database community. For reasons of space it is beyond the scope of this paper to acknowledge these important contributions. We refer the interested reader to one of the many excellent surveys on provenance [4–6, 8, 12].

2 Instrumentation - Exploiting a DBMS for Provenance Storage and Capture

The de facto standard for database provenance [9] is to model provenance as annotations on data and to define a query semantics that determines how annotations propagate. Under such a semantics, each output tuple t of a query Q is annotated with its provenance, i.e., a combination of input tuple annotations that explains how these inputs were used by Q to derive t. For instance, using provenance polynomials, each tuple is annotated with a, typically unique, variable representing this tuple. Under this model, annotations are propagated such that every query result tuple is annotated with a polynomial over the variables representing the input tuples in the output's provenance. The addition and multiplication operations in such polynomials encode how these inputs have been combined to derive the output. Addition represents alternative use of inputs (e.g., union or projection) and multiplication represents conjunctive use (e.g., join). Relations annotated with provenance polynomials are a specific type of K-relations, relations where tuples are annotated with elements from a commutative semiring such as the semiring of provenance polynomials (denoted as $\mathbb{N}[X]$). Relational algebra over K-relations is defined based on the addition and multiplication operations of the semiring.

For example, consider a query listing all states that have CS students. The query and example $\mathbb{N}[X]$-relations encoding the input and output are shown at the top of Figure 1. The query result `(IL)` is annotated with $v + y$ which indicates that this tuple is part of the result as long as either v or x exist in the input (students Alice or Peter). As we will discuss in Section 4.1, the semiring provenance model which originally was defined for queries can be extended to also support transactional updates. GProM targets any type of provenance or other information that can be modelled as annotations. Current database systems do not natively support the propagation of annotations through operations. There are two approaches for making a database system provenance-aware. Either we extend the system's query execution engine to support these features natively or we encode provenance annotations using the data model supported by the database and *instrument* operations to propagate provenance annotations. GProM and many other database provenance systems such as Perm [7], LogicBlox, Orchestra, and ExSPAN apply the second approach. Using a relational encoding of provenance annotations, these systems compile queries with annotated semantics into relational queries that produce this encoding of provenance annotations. We refer to this reduction from annotated to standard relational semantics as *instrumentation*. The instrumentation approach can either be implemented as a compilation process in a middleware application or as a query rewrite layer within a DBMS that instruments queries before they are passed to the system's optimizer (e.g., the *Perm* system [7] is an extension of PostgreSQL with support for capturing and querying provenance). In GProM we have opted for a middleware implementation to be able to support multiple database backends.

An example of instrumentation is shown at the bottom of Figure 1. The input query with annotated semantics is instrumented to produce a relational encoding of provenance polynomial annotations. In this example, we use an encoding pioneered in Perm [7] which represents a tuple t annotated with polynomial k as follows. The polynomial is refactored into a sum of products where variables in each monomial (individual product in the sum) are ordered based on the occurrence of the relations in the query. A polynomial normalized in this fashion is then encoded as a set of tuples where each tuple in such a set represents one monomial. A variable is represented by the values of the tuple annotated with this variable in the input. Here P denotes a renaming function which is used to create names for attributes that store provenance. For instance, an attribute `student.name` would be renamed as `prov_student_name`. Note that this is only one of the representations supported in GProM, e.g., the user can alternatively request the system to use tuple identifiers to encode variables. Furthermore, annotations are generated on the fly in this example. This, however, is not a requirement. The instrumentation approach works perfectly well for inputs which have provenance associated with them.

3 System Overview

We now give a more detailed and technical overview of GProM. Figure 2 shows the high-level architecture of GProM. A user interacts with the system by sending a query written in a frontend language using one of the system's client interfaces (e.g., using a CLI). Frontend languages are declarative query languages that have been enriched with new language constructs for requesting and querying provenance. A frontend-specific parser translates an incoming query into a more suitable internal representation, e.g., relational algebra. The result is then

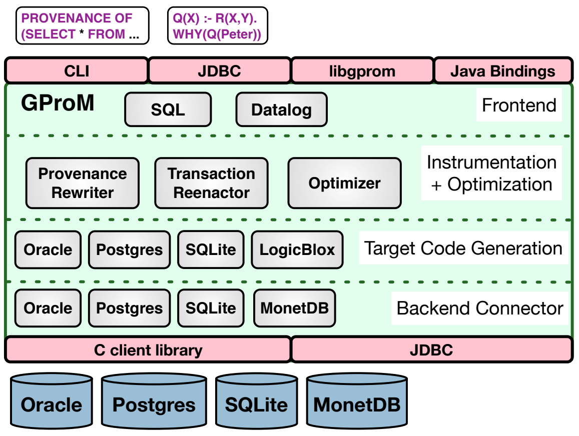
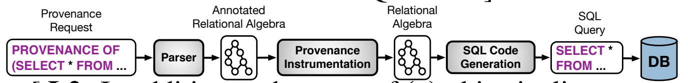
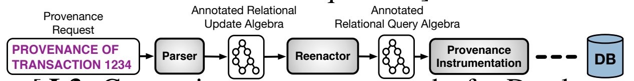


Figure 2: GProM Architecture

[**L1:** Provenance is captured using an annotated version of relational algebra which is first translated into relational algebra over a relational encoding of annotated relations and then into SQL code.]

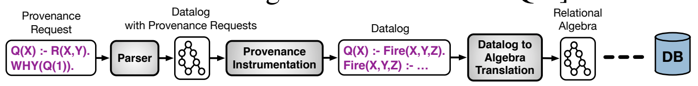


[**L2:** In addition to the steps of **(a)**, this pipeline uses *reenactment* [2] to compile annotated updates into annotated queries.]



Provenance Request
PROVENANCE OF TRANSACTION 1234
Parser
Annotated Relational Update Algebra
Reenactor
Annotated Relational Query Algebra
Provenance Instrumentation
DB

[**L3:** Computing provenance graphs for Datalog queries [10] based on a rewriting called *firing rules*. The instrumented Datalog program is first compiled into relational algebra and then into SQL.]



Provenance Request
Q(X) :- R(X,Y). WHY(Q(1)).
Parser
Datalog with Provenance Requests
Provenance Instrumentation
Datalog
Q(X) :- Fire(X,Y,Z). Fire(X,Y,Z) :- ...
Datalog to Algebra Translation
Relational Algebra
DB

Figure 3: Instrumentation pipelines for provenance: **(a) L1:** SQL queries, **(b) L2:** transactions, **(c) L3:** Datalog

processed by one or more instrumentation components which translate parts of a query containing provenance features by rewriting such parts to generate a relational encoding of provenance. Instrumentation is typically broken down into a multi-step compilation process which has the advantage that components implementing a compilation step can be utilized for multiple provenance tasks. GProM also features a generic optimizer that can be applied to any such compilation step (see Section 4.3). The output of instrumentation is then processed by a backend specific code generation module, e.g., translating relational algebra into Postgres's SQL dialect. The generated code is sent to the backend for execution using a backend connector. Connectors either use a native C-library or a JDBC driver to connect to the backend system. GProM was designed from the ground up to be as modular and extensible as possible. Most components of the system including the frontend parser, instrumentation and optimization components, code generators, and connectors are pluggable.

3.1 Frontends

So far we have implemented two frontends in GProM: 1) an SQL dialect with provenance features and 2) a Datalog frontend with support for requesting explanations (provenance) for existing and missing query answers. Importantly, the SQL dialect also supports other types of annotations such as temporal data and uncertainty. Our general philosophy in designing these language extensions was to make provenance requests proper query constructs that can be used in almost any place where regular queries are allowed. Importantly, this enables the full expressive power of the frontend language to be used for querying provenance information.

3.2 Instrumentation Pipelines

To implement a particular provenance task, e.g., compile a provenance request written in GProM's SQL dialect into Postgres's SQL dialect, the instrumentation components of GProM are arranged into a so-called *instrumentation pipeline*. Figure 3 shows some of the pipelines currently supported by GProM:

- **L1. Provenance for SQL Queries:** The pipeline from Figure **??** generates a relational encoding of provenance annotations such as the one shown in Figure 1.

- **L2. Provenance for Transactions:** Figure **??** shows a pipeline that retroactively captures provenance for transactions. In addition to the steps from Figure **??**, this pipeline uses a compilation step called *reenactment*. Reenactment translates transactional histories with annotated semantics into equivalent temporal queries with annotated semantics. We will discuss this pipeline in more detail in Section 4.1.
- **L3. Provenance for Datalog:** This pipeline (Figure **??**) produces provenance graphs that explain which successful and failed rule derivations of a Datalog program are relevant for (not) deriving a (missing) query result. A provenance request is compiled into a program that computes the edge relation of the provenance graph. This program is then translated into SQL. See Section 4.2 for a more detailed discussion.

Note that a frontend may support multiple pipelines. User requests written in the language of the frontend are automatically dispatched to the pipeline that is responsible for handling this type of request. For instance, the SQL frontend uses several pipelines including the pipelines L1 and L2 described above. If a user requests the provenance for a query then this request will be handled by pipeline L1 whereas if the user requests the provenance of a transaction then this request will be dispatched to pipeline L2.

3.3 Backends and Client Interfaces

GProM can be accessed through its native commandline shell (CLI), through a library (libgrom), using a Java API, or through the system's JDBC driver which wraps a vendor specific JDBC driver. GProM supports code generation for the SQL dialects of Oracle, Postgres, and SQLite as well as for LogiQL, LogicBlox's Datalog dialect. Backend connectors have been implemented for Oracle, Postgres, SQLite, and MonetDB.

4 Research Contributions

In addition to building a platform to support diverse provenance needs, our work on GProM has laid out the basis for future research related to data provenance. At the same time, building GProM required us to cover new ground in provenance research. In the following, we cover three major research contributions: tracking provenance of transactions, unifying why- and why-not provenance, and provenance-aware query optimization.

4.1 Provenance for Transactions and Reenactment

One major limitation of database provenance approaches is their lack of support for tracking provenance of transactional updates. This has prevented the use of provenance for applications such as debugging of transactions and auditing. For instance, consider the following scenario. A company uses a database to store mission-critical data and an attacker has compromised one of the database user accounts. Provenance for transactions would allow us to determine what data was accessed by the attacker through this account. Furthermore, it would allow us to determine what data was affected directly or indirectly by updates run by the compromised account (e.g., a reporting query returns an incorrect result because of the attacker has modified data). To address this shortcoming, we have developed *MV-semirings* (multi-version semirings), the first provenance model for transactions, and reenactment, a technique for retroactively capturing the provenance of past transactions using queries.

The MV-semiring model extends the semiring annotation framework [9] to account for tuple derivations under transactional updates [1, 2]. For any semiring $\mathcal{K}$, we can construct an MV-semiring $\mathcal{K}^{\nu}$. For instance, $\mathbb{N}[X]^{\nu}$ is the MV-version of the provenance polynomial semiring $\mathbb{N}[X]$. An annotation from an MV-semiring $\mathcal{K}^{\nu}$ is a symbolic expression over elements from $\mathcal{K}$ recording the derivation history of a tuple. These expressions use *version annotations* to enclose part of the provenance of a tuple to encode that the tuple version corresponding to this part of the provenance was processed by a certain update at a certain time. A version annotation $X^{id}_{T,\nu}(k)$ denotes that an operation of type X (update U, insert I, delete D, or commit C) that was executed at time

$\nu - 1$ (we assume a totally ordered time domain that is used to identify versions) by transaction T affected a previous version of a tuple with identifier id and previous provenance k. The nesting of version annotations in the MV-semiring annotation of a tuple records the sequence of updates that lead to the creation of the current version of the tuple. We have defined update operations and a transactional semantics for MV-semiring databases that is backward compatible to *snapshot isolation* (SI) and *read committed snapshot isolation* (RC-SI) for bag semantics databases. In the resulting semantics, each tuple in a version of a database produced by a history of transactions is annotated with its complete derivation history according to a SI or RC-SI history. Our model also supports provenance for queries, i.e., the provenance a query result cannot just be traced back to the inputs of the query, but also reaches back into the transactional history that produced these inputs. Furthermore, our model preserves a major advantage of the semiring framework: it generalizes set and bag semantics as well as other types of annotations expressible in the semiring framework such as incomplete databases. That is, we can determine the bag semantics database that is the result of a given snapshot isolation history from the annotated database for this history. We make use of this property to build a transactional debugger (see Section 5).

Example 4.1: Consider a tuple version t from an $\mathbb{N}[X]^{\nu}$-relation R (the MV-version of provenance polynomials) that was created by a SI history. Assume that in the current version of relation R, tuple t is annotated with $C^2_{T_1,6}(I^2_{T_1,4}(x_2))$. This annotation records that the tuple was produced by an insert (I) executed by transaction T_1 at time 3 and was assigned a tuple identifier 2. Transaction T_1 committed at time 5 after which this version of tuple t became visible to other transactions. This is encoded by the outer version annotation: $C^2_{T_1,6}$. Note that we assign a time stamp $\nu + 1$ to tuples created by an update or commit executed at time ν. We assign a fresh variable (x_2 in the example) to tuples created by an insert using a **`VALUES`** clause. Inserted tuples are assigned new tuple ids (id 2, shown as a superscript in the version annotation).

We have demonstrated [1, 2] that MV-semiring databases inherit many of the beneficial properties of K-relations (the semiring annotation framework) and are a strict generalization of K-relations in the following sense: given a semiring $\mathcal{K}$, the corresponding MV-semiring $\mathcal{K}^{\nu}$ is also a semiring. That means that we can apply the standard query semantics for K-relations to query a $\mathcal{K}^{\nu}$-relation. Furthermore, the K-relation corresponding to an $\mathcal{K}^{\nu}$-relation R can be extracted from R by applying a semiring homomorphism $\textsc{Unv}$ which evaluates the symbolic expression that is a $\mathcal{K}^{\nu}$ annotation by interpreting version annotation as functions from $\mathcal{K}$ to $\mathcal{K}$. Importantly, any semiring homomorphism $h : \mathcal{K}_1 \to \mathcal{K}_2$ can be lifted to a homomorphism ${\mathcal{K}_1}^{\nu} \to {\mathcal{K}_2}^{\nu}$ which in addition to queries also commutes with transactional histories. Such a lifted homomorphism replaces $\mathcal{K}_1$ elements in an annotation with $\mathcal{K}_2$ elements according to h. For example, consider how to derive a bag semantics annotation from the $\mathbb{N}[X]^{\nu}$ annotation from the example above ($C^2_{T_1,6}(I^2_{T_1,4}(x_2))$). In the K-relational model, bag semantics is modelled by annotating tuples with their multiplicity (the semiring $\mathbb{N}$ of natural numbers). We first apply $\textsc{Unv}$ to get the provenance polynomial for tuple t. Both commit and insert annotations are interpreted as the identity function for $\mathbb{N}[X]$. Thus, $\textsc{Unv}(C^2_{T_1,6}(I^2_{T_1,4}(x_2))) = x_2$. Now further assume that tuple t appears with multiplicity 2 in the input, i.e., we apply a homomorphism $\mathbb{N}[X] \to \mathbb{N}$ based on the valuation $x_2 = 2$ and get 2. That is, in the current version of relation R, the tuple t from the example appears with multiplicity 2. The interested reader is referred to [1, 2] for the definition of update operations and transactional semantics for MV-databases as well as a more formal discussion of the properties of MV-semiring structures.

We have proven that if we extend our query model with a new operator that creates version annotations, then any update, transaction, or (partial) history in our model can be equivalently expressed as a query, e.g., from an update u we can derive a query $\mathbb{R}(u)$ which returns the same database state as the original update u (if executed over the same input). We call such queries *reenactment queries*. The equivalence of an operation and its reenactment query under annotated semantics has an important implication: instead of computing provenance eagerly during transaction execution we can compute it retroactively by running reenactment queries. Since our model generalizes bag-semantics snapshot isolation, we can use reenactment to recreate a database state valid at a particular time by simply running a query - including database states that were only visible within one transaction.

We have implemented support for transaction provenance in GProM based on reenactment. The instrumentation pipeline implementing transaction provenance is shown in Figure **??**. In addition to supporting provenance capture for past transactions using the auditing logging and time travel capabilities of modern DBMS, this pipeline also allows a hypothetical sequence of updates to be evaluated using reenactment.

Example 4.2: Assume a user is interested in evaluating the effect of a hypothetical update over the current version of a bag semantics relation `Emp(name, salary)` which increases the salary of all employees by \$500 if their current salary is less than \$1000. For simplicity, assume that the user is not interested in provenance (we use semiring $\mathbb{N}$ instead of $\mathbb{N}[X]^{\nu}$). This request is expressed using GProM's **`REENACT`** statement:

```
REENACT(UPDATE Emp SET salary = salary + 500 WHERE salary < 1000;);
```

To reenact this update over the current version of relation `Emp`, GProM would construct a reenactment query which returns the new state of `Emp` produced by the hypothetical update. This state is computed as a union between the set of tuples that would not be updated (do not fulfill the update's condition) and the updated versions of tuples that fulfill the update's condition (we have to increase their salary by `500`):

```
SELECT * FROM Emp WHERE NOT(salary < 1000)
UNION ALL
SELECT name, salary + 500 AS salary, b FROM Emp WHERE salary < 1000;
```

4.2 Unifying Why and Why-not Provenance

The problems of explaining why a tuple is in the result of a query or why it is missing from the result, i.e., why and why-not provenance, have been studied extensively. However, these two problems (computing provenance and explaining missing answers) have been treated mostly in isolation. An important observation is that for queries with negation, the two problems coincide: to explain why a tuple t is not in the result of a query Q, we can equivalent ask why t is in the result of $\neg Q$. Thus, a provenance model for queries with negation should naturally be able to support why-not questions. While there are extensions of the semiring model for set difference, which encodes a form of negation, the problem is that in general $\neg Q$ may not be safe. Thus, to unify the two worlds of why and why-not provenance we need a provenance model that permits unsafe queries. We have introduced in [10] a graph-based provenance model for first-order (FO) queries expressed as non-recursive Datalog queries with negation. We apply the closed world assumption to deal with unsafe queries.

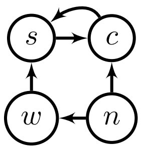
Our approach for computing provenance according to this model is based on the observation that typically only a part of provenance, which we call an *explanation*, is actually relevant for answering a user's provenance question about the existence or absence of a result. An explanation for a why (why-not) question should justify the existence (absence) of a result as the success (failure) to derive the result through the rules of the query. Furthermore, it should explain how the existence (absence) of tuples in the database caused the derivation to succeed (fail). The main driver of our approach is a rewriting of Datalog rules that captures successful and failed rule derivations. This rewriting replaces the rules of a program with so-called *firing rules*. To efficiently compute an explanation, we generate a Datalog program consisting of a set of firing rules that computes the relevant part of the provenance bottom-up. Evaluating this program over a given instance returns the egde relation of the explanation (provenance graph).

We have implemented this approach in GProM as Pipeline L3 (shown in Figure **??**). The user provides a why or why-not question and the corresponding Datalog query as an input. For this pipeline, we use GProM's Datalog frontend, which provides language constructs for expressing provenance requests. For instance, `WHY(Q(a))` would instruct GProM to explain why `Q`(a) is a query result. The system instruments the input program (query) to capture provenance relevant to the user question based on the firing rule rewriting mentioned above. This program is translated into relational algebra and the resulting algebra expression is then translated into SQL and sent to the backend databases to compute the edge relation of the explanation for the provenance question. Based on this edge relation, we render a provenance graph e.g., the graphs shown in Figure 5 and 6.

```
r1 : Q(X,Y) :- Train(X,Z),
               Train(Z,Y),
               not Train(X,Y).
```

Relation **Train**

fromCity	**toCity**
new york	washington dc
new york	chicago
chicago	seattle
seattle	chicago
washington dc	seattle



Result of query **Q**

X	**Y**
washington dc	chicago
new york	seattle
seattle	seattle
chicago	chicago

Figure 4: Example train connection database and query

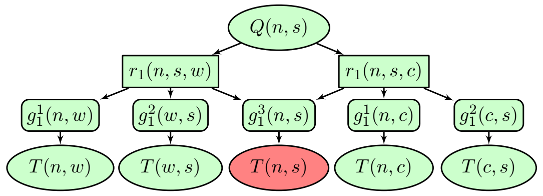


Figure 5: Provenance graph for `WHY(Q(n,s))`

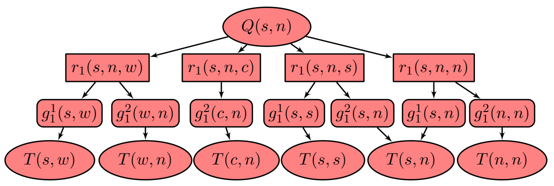


Figure 6: Provenance graph for `WHYNOT(Q(s,n))`

Example 4.3: Consider the train connections relation shown in Figure 4 and Datalog query r_1 that computes which cities can be reached with exactly one transfer, but not directly. A user might wonder why it is possible to reach Seattle from New York with one intermediate stop but not directly (`WHY(Q(n,s))`) or why it is not possible to reach Seattle from New York in the same fashion (`WHYNOT(Q(s,n))`). The provenance graph in Figure 5 explains, for the question `WHY(Q(n,s))`, how Seattle can be reached from New York via one intermediate hop, but not directly. Here, we use the following abbreviations: T = Train, n = New York, s = Seattle, w = Washington DC, and c = Chicago. In the example instance, there are two ways to reach Seattle from New York in this fashion: stopping either in Washington DC or in Chicago. These options correspond to two successful derivations of rule r_1 with X=n, Y=s, and Z=w (or Z=c, respectively). The provenance graphs produced by GProM contain three types of nodes: tuple nodes (ovals), rule nodes (rectangles), and goal nodes (rounded rectangles). The color of a node denotes its success (green) or failure (red), e.g., $\mathtt{Q}(n,s)$ is labelled successful, because this tuple exists in the query result. In Figure 5 there are two rule nodes denoting the two successful derivations of $\mathtt{Q}(n,s)$ by rule r_1. The provenance graph for question `WHYNOT(Q(s,n))` (Figure 6) explains why it is not true that New York can be reached from Seattle with exactly one transfer, but not directly. The tuple $\mathtt{Q}(s,n)$ is missing from the query result, because all potential ways to derive this tuple through r_1 have failed. In this example, there are four failed derivations - each choosing one of the four cities present in the database as an intermediate stop. For instance, we cannot reach New York from Seattle via Washington DC (the first failed rule derivation from the left in Figure 6), because there exists no connection from Seattle to Washington DC (a tuple node $\mathtt{T}(s,w)$ in red), and WashingtonDC to New York (a tuple node $\mathtt{T}(w,n)$ in red). Note that the goal $\neg\mathtt{T}(s,n)$ is successful and, thus, is not part of the explanation (successful goals do not contribute to failed derivations).

4.3 Provenance-aware Query Optimization

The instrumentation approach implemented in GProM has the distinct advantage that it does not require any changes to the backend database system. However, because of the intrinsic complexity and unusual structure of instrumented queries, even sophisticated database optimizers are often producing suboptimal plans for such queries. DBMS optimizers have to trade optimization time for query performance. Thus, optimizations that do not benefit common workloads are typically not considered. To address this problem, we have developed a heuristic and cost-based optimization framework for instrumentation pipelines [16]. A unique feature of our optimizer is that it is plan-space and query language agnostic and, thus, can be applied to optimize any instrumentation pipeline in GProM. Our experimental results demonstrate that this approach is quite effective, improving performance by several orders of magnitude for diverse provenance tasks.

Recall that an instrumentation pipeline is a multistep compilation process. To optimize this process we can either target an intermediate language that is the result of a compilation step or the compilation step itself.

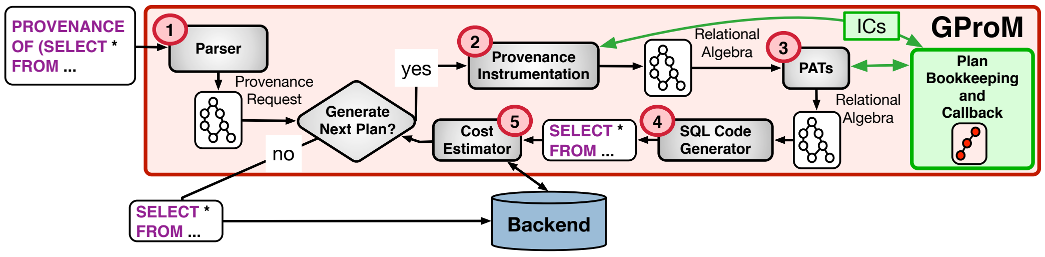


Figure 7: Optimizer workflow

As an example for the first type of optimization, consider a compilation step that outputs relational algebra, e.g., Pipelines L1 and L3 from Figure 3. We can optimize the generated algebra expression using algebraic equivalences before passing it on to the next stage of the pipeline. In [16], we focus on relational algebra since it is an intermediate language used by almost all pipelines supported by GProM. We investigate algebraic equivalences that are beneficial for instrumentation, but which are usually not applied by database optimizers. We call this type of optimizations *provenance-specific algebraic transformations* (PATs). For instance, pull up projections that create provenance annotations and remove unnecessary duplicate elimination and window operators. To be able to support rules whose conditions depend on non-local information and to simplify definition of rules, we infer properties such as candidate keys for the algebra operators of a query. For example, a duplicate elimination operator δ is redundant if its input relation is duplicate free, i.e., if it has at least one super key. Whether this is the case depends not only the operator itself, but also on its context: the subtree below the operator in this case. One of the properties we infer is a set $keys$ of super keys for an operator's output. Given this property, a PAT rule that removes duplicate eliminations is trivially expressed as: rewrite $\delta(R)$ as R if $keys(R) \neq \emptyset$.

For the second type of optimization mentioned above consider the compilation step from Pipeline L1 that translates relational algebra with annotated semantics into relational algebra. If we know two equivalent ways of translating an operator with annotated semantics into relational algebra, then we can choose the translation that maximizes performance. We refer to this type of optimizations as *instrumentation choices* (ICs). For instance, we introduce two ways for instrumenting an aggregation for provenance capture: 1) using a join [7] to pair the aggregation output with provenance from the aggregation's input; 2) using window functions (SQL's **`OVER`** clause) to directly compute the aggregation functions over inputs annotated with provenance.

Since some PATs are not always beneficial and for some ICs there is no clearly superior choice, there is a need for *cost-based optimization* (CBO). We have developed a CBO for instrumentation pipelines that can be applied to any pipeline no matter what compilation steps and intermediate languages are used. This is made possible by decoupling the plan space exploration from actual plan generation. Figure 7 shows how our cost-based optimizer is integrated with GProM. Our CBO treats an instrumentation pipeline as a blackbox function which it calls repeatedly to produce backend dialect queries (plans). Plans are sent to the backend for planning and cost estimation. We refer to one execution of the pipeline as an iteration. It is the responsibility of the pipeline's components to signal to the optimizer the existence of optimization choices (called *choice points*) through the optimizers callback API. The optimizer responds to a call from one of these components by instructing it which of the available options to choose. We keep track of which choices had to be made, which options exist for each choice point, and which options were chosen. This information is sufficient to enumerate the plan space by making different choices during each iteration. Our approach provides great flexibility in terms of supported optimization decisions, e.g., we can choose whether to apply a PAT or select which ICs to use. Adding an optimization choice only requires adding a few LOC to inform the optimizer about the availability of options.

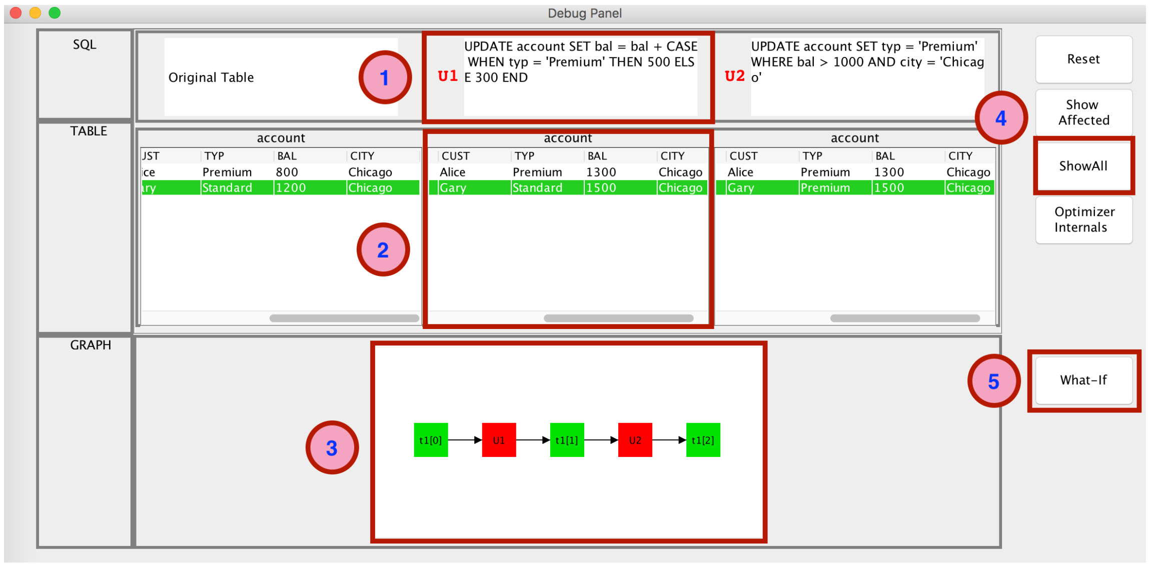
4.4 Further Contributions and Research that Utilizes GProM

In addition to the three major contributions outlined above, GProM has been the basis of many additional research thrusts. In [15], we have demonstrated how to improve interoperability between GProM and other provenance-aware systems. Specifically, we have extended Pipeline L1 from Section 3.2 to translate provenance generated by GProM into the W3C PROV standard format (`https://www.w3.org/TR/prov-overview/`) and how to propagate provenance imported as PROV through queries. Reenactment enables changes to data to be virtualized - instead of running an update we can instead just record the update statement and evaluate its effect in a non-destructive manner using reenactment. Based on this idea we have presented our vision of *provenance-aware versioned dataworkspaces* (*PVDs*) [13] which are virtual copies of a database with non-linear version histories (like the ones supported by version control systems) which can be used for exploratory purposes. We have identified historical what-if queries [3] as another use case for reenactment. Using reenactment, we can efficiently determine the effect of hypothetical changes to past update operations on the current database state. For instance, we can answer queries such as *"How would the revenue of our company be affected if we would have charged 10% interest for account overdraws instead of 7%"*. In [11] we have introduced an approximate summarization technique for why and why-not provenance extending our previous work on Datalog provenance in GProM. Using a sampling-based method, we overcome the main roadblock for explaining missing answers - the prohibitively large size of why-not provenance for databases of realistic size.

5 Post-mortem Debugging of Transactions

Aside from providing a solid platform for research on provenance and related fields, GProM and the research fueling the system have also enabled novel applications of provenance that would not have been possible before. In this section, we introduce postmortem debugging of transactions as an important example for this type of application. Debugging transactions and understanding their execution is of immense importance for developing OLAP applications, to trace causes of errors in production systems, and to audit the operations of a database. Debugging transactions, just like debugging of parallel programs, is hard because errors may only materialize under certain interleavings of operations. This problem is aggravated by the wide-spread use of lower isolation levels. Nonetheless, even for serializable histories an error may only arise for some execution orders. To debug an error, we have to reproduce the interleaving of operations which lead to the error. This problem can be addressed by supporting post-mortem debugging for transactions, i.e., enabling a user to retroactively inspect transaction executions to understand how the statements of a transaction affected the database state. While there are debuggers for procedural extensions of SQL, e.g., Microsoft's T-SQL Debugger (`http://msdn.microsoft.com/en-us/library/cc645997.aspx`), these debuggers treat SQL statements as black boxes, i.e., they do not expose the dataflow within an SQL statement. Even more important, they do not support post-mortem debugging of transaction executions within their original environment.

Supporting post-mortem debugging for transactions is quite challenging, because past database states are transient and the dataflow within and across SQL statements is opaque. While temporal databases provide access to past database versions, this is limited to committed versions. In [14], we present a non-destructive, post-mortem debugger for transactions that relies on GProM's reenactment techniques to reproduce the intermediate states of relations seen by the operations of a transaction. The approach uses provenance to expose data-dependencies between tuple versions and to explain which statements of a transaction affected a tuple version. Advanced debuggers for programming languages allow code to be changed during a debugging session to test a potential fix for a bug. We exploit the fact that GProM supports reenactment of hypothetical transactions to support such what-if scenarios, i.e., changes to a transaction's SQL statements. Being based on GProM's reenactment functionality, our approach uses the temporal database and audit logging capabilities available in many DBMS and does not require any modifications to the underlying database system nor transactional workload.



```
UPDATE account SET bal = bal + CASE
 WHEN typ = 'Premium' THEN 500 ELS
E 300 END
```

```
UPDATE account SET typ = 'Premium'
WHERE bal > 1000 AND city = 'Chicag
o'
```

account

JST	TYP	BAL	CITY
ice	Premium	800	Chicago
ry	Standard	1200	Chicago

account

CUST	TYP	BAL	CITY
Alice	Premium	1300	Chicago
Gary	Standard	1500	Chicago

account

CUST	TYP	BAL	CITY
Alice	Premium	1300	Chicago
Gary	Premium	1500	Chicago

Figure 8: Screenshot of the Debugger GUI

```
UPDATE account
SET bal = bal + CASE WHEN typ = 'Premium'
                THEN 500 ELSE 300 END

UPDATE account SET typ = 'Premium'
WHERE bal > 1000 AND city = 'Chicago';
```

(a) Transaction T

account

cust	typ	bal	city
Alice	Premium	800	Chicago
Gary	Standard	1200	Chicago

(b) Before T

account

cust	typ	bal	city
Alice	Premium	1300	Chicago
Gary	Premium	1500	Chicago

(c) After T

Figure 9: Example transaction

Example 5.1: Transaction T shown in Figure 9 adds a bonus to bank accounts ($500 for premium customers and $300 for standard customers) and gives premium status to all accounts whose balance is larger than $1000. Figure 9 also shows the state of the account relation before and after the update. For instance, after the execution of Transaction T, Gray's account enjoys premium status. However, Gary has only received a $300 bonus, the bonus for standard accounts. Our debugger can be used to inspect the internal states of the transaction that lead to this behaviour. Figure 8 shows a screenshot of the debugger's GUI for Transaction T. We show one column for each of T's operations plus a column for the initial states of the relations accessed by T. Each such column shows the SQL code of the statement (①) and the relation (②) modified by the statement (the version created by the statement). For each tuple version, we show which transaction created that version. In Figure 8, the user has selected Gary's account. Thus, the debugger shows a provenance graph (③) for this tuple and highlights the updates that affected it. From the intermediate states of the relations and the provenance graph it is immediately clear that the bonus payment was applied when Gary's status was still standard. Our debugger supports two types of what-if scenarios by clicking the "What-if" button (⑤): 1) the user can edit the data in a table and 2) the user can modify, delete, or add an update statement.

6 Conclusions and Future Work

We give a comprehensive overview of GProM (**G**eneric **Pro**venance **M**iddleware). GProM is a fully implemented system that we hope will serve as a platform for future research on provenance and annotation management as well as a framework for building provenance-aware applications. The diverse types of research threads and applications for which we have employed GProM, demonstrate the potential of our system and the feasibility of its modular, extensible design. We also give an overview of research contributions related to GProM, most notably, a provenance model for transactions and reenactment, capturing why and why-not provenance for

Datalog queries, and provenance-aware query optimization. The query instrumentation technique that is at the heart of GProM is applicable for a wide range of use cases that are not necessarily all related to provenance. For instance, we have extended GProM to evaluate temporal queries and to compute uncertainty annotations. GProM provides solid support for classical applications of provenance and has enabled novel applications. We discuss post-mortem debugging of transactions as one exciting use case of the system. Finally, we highlight interesting future work and discuss ongoing research efforts that benefit from GProM.

Acknowledgements. This work was supported by NSF Award #1640864. Opinions, findings and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We would like to thank our collaborators without whom this work would not have been possible and the Oracle corporation for generously supporting this work through multiple gifts.

References

[1] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Reenactment for read-committed snapshot isolation. In *CIKM*, pages 841–850, 2016.

[2] B. Arab, D. Gawlick, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Using reenactment to retroactively capture provenance for transactions. *TKDE*, 2017 (to appear).

[3] B. Arab and B. Glavic. Answering historical what-if queries with provenance, reenactment, and symbolic execution. In *TaPP*, 2017.

[4] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[5] S. B. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludäscher, T. McPhillips, S. Bowers, and J. Freire. Provenance in Scientific Workflow Systems. *IEEE Data Engineering Bulletin*, 32(4):44–50, 2007.

[6] B. Glavic and K. R. Dittrich. Data Provenance: A Categorization of Existing Approaches. In *Proceedings of the 12th GI Conference on Datenbanksysteme in Buisness, Technologie und Web*, pages 227–241, 2007.

[7] B. Glavic, R. J. Miller, and G. Alonso. Using SQL for efficient generation and querying of provenance information. *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman*, pages 291–320, 2013.

[8] M. Herschel, R. Diestelkämper, and H. B. Lahmar. A survey on provenance: What for? what form? what from? *The VLDB Journal*, 26(6):881–906, 2017.

[9] G. Karvounarakis and T. Green. Semiring-annotated data: Queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.

[10] S. Lee, S. Köhler, B. Ludäscher, and B. Glavic. A sql-middleware unifying why and why-not provenance for first-order queries. In *ICDE*, pages 485–496, 2017.

[11] S. Lee, X. Niu, B. Ludäscher, and B. Glavic. Integrating approximate summarization with provenance capture. In *TaPP*, 2017.

[12] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2–3):99–241, 2010.

[13] X. Niu, B. Arab, D. Gawlick, Z. H. Liu, V. Krishnaswamy, O. Kennedy, and B. Glavic. Provenance-aware versioned dataworkspaces. In *TaPP*, 2016.

[14] X. Niu, B. Glavic, S. Lee, B. Arab, D. Gawlick, Z. H. Liu, V. Krishnaswamy, F. Su, and X. Zou. Debugging transactions and tracking their provenance with reenactment. *PVLDB*, 10(12):1857–1860, 2017.

[15] X. Niu, R. Kapoor, D. Gawlick, Z. H. Liu, V. Krishnaswamy, V. Radhakrishnan, and B. Glavic. Interoperability for provenance-aware databases using PROV and JSON. In *TaPP*, 2015.

[16] X. Niu, R. Kapoor, B. Glavic, D. Gawlick, Z. H. Liu, V. Krishnaswamy, and V. Radhakrishnan. Provenance-aware query optimization. In *ICDE*, pages 473–484, 2017.

Supporting Data Provenance in Data-Intensive Scalable Computing Systems

Matteo Interlandi, Tyson Condie
Microsoft

Abstract

Debugging data processing logic in Data-Intensive Scalable Computing (DISC) systems is a difficult and time consuming effort. Data provenance support is a key building block in libraries that aim to provide debugging support for data processing pipelines. In this paper we report our experience in building Titian: a data provenance system targeting the Apache Spark framework. Our focus here is to analyze the design choices and trade offs that we and others made. Ultimately, we believe there is still more work to do before reaching a widespread adoption of data provenance outside the research community.

1 Introduction

Data-Intensive Scalable Computing (DISC) systems, like Apache Hadoop [1] and Apache Spark [2], are being used to analyze massive quantities of data. These DISC systems expose a programming model for authoring data processing logic, which is compiled into a Directed Acyclic Graph (DAG) of data-parallel operators. The root DAG operators consume data from an input source (e.g., HDFS), while downstream operators consume the intermediate outputs from DAG predecessors. Scaling to large datasets is handled by partitioning the data and assigning tasks that execute the operators on each partition.

Given its distributed and large-scale nature, debugging data processing logic in DISC environments can be daunting. DISC systems expose a batch model of execution: applications are run in the cloud, and the results, including notification of runtime failures, are sent back to users upon completion. Therefore, debugging is mostly done *post-mortem* and the primary source of debugging information is an execution log. Another common debugging pattern is *trial-and-error* iterations, where developers *selectively replay* a portion of their data processing logic on input samples or subsets of intermediate data leading to erroneous results. Trial-and-error debugging is often a slow and error prone process inasmuch as each iteration is executed afresh, and users have to be manually filter records after each iteration. Only recently, a set of tools and libraries [4–6, 8] have started to arise for helping users in interactively identifying the subset of data leading to failures, or to optimize trial-and-error runs in a principled and automatic way. All these tools can be unlocked by a DISC system equipped with the following capabilities: (1) *scalable fine-grained data provenance* (also referred to data lineage) *capturing* introducing low overhead on the runtime; (2) *interactive provenance query capabilities* enabled within the same host framework; and (3) *a flexible and simple to use API* allowing to seamlessly move between provenance and data records without triggering any re-computation.

Newt [10] and RAMP [7] are two frameworks supporting data provenance in DISC systems (Hadoop in this specific case). Unfortunately, none of them satisfy all three requirements: Newt is an external library whereby the dataflow operators of the target DISC system are wrapped with fine-grained provenance capturing logic downpouring provenance information to a storage layer (MySQL). While this design allows some level of portability, we found that interactiveness is somehow restricted because a different system has to be used at provenance query time. Conversely, RAMP tags data records with provenance information which are propagated downstream and collected into HDFS. This design requires modification to DISC internals while only a limited number of tracing queries are supported efficiently. Furthermore, scalability is suboptimal in both system as a consequence of the overheads due to the transfer of provenance information between different sub-systems (Newt) and downstream as tags (RAMP).

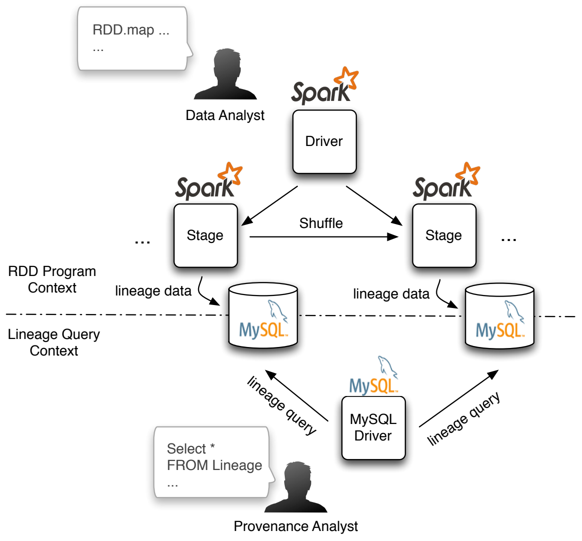
The main focus of this paper is to summarize our experience in building *Titian* [9]: a fine-grained provenance framework specifically designed for the popular Apache Spark system, and satisfying the above three requirements. Differently than RAMP and Newt, Titian was implemented with usability and scalability in mind: provenance capturing and querying is performed on the same host system whereby users can employ the same interface (i.e., RDD transformations [11]) to interactively author DISC programs and query provenance information. By tightly integrating Titian within Spark, both provenance information can be captured with low overhead, and data records can be accessed without any re-computation.

While these design choices made possible the development of several debugging functionalities and tools on top of Titian [4–6, 8], we believe several limitations and open questions remain. Specifically, we found that targeting Titian to Spark's low-level (RDD) API makes it difficult to port to newer versions of Spark. Moreover, mapping data provenance to high-level operations (e.g., SQL, Machine Learning) is non-trivial, and would require a complete overhaul of the framework. A more detailed discussion on these and other issues can be found in Section 5.

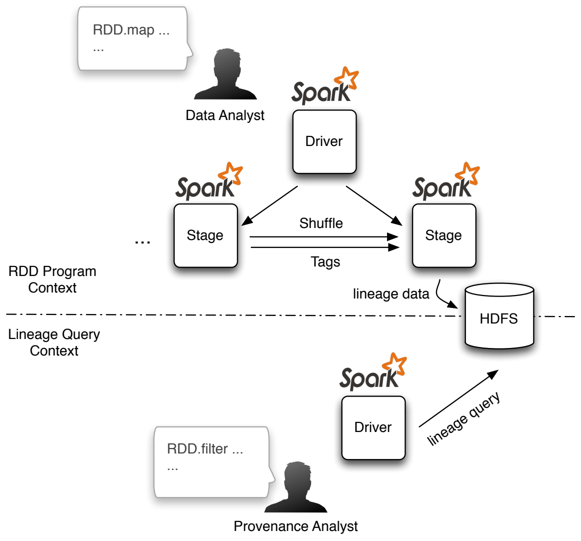
In the remainder of the paper we will first discuss our porting of Newt and RAMP over Apache Spark, and related usability and scalability difficulties (Section 3); then we will introduce Titian design and detailed implementation (Section 4). The paper will end with a discussion on the lesson learned and future directions. To properly put everything into context, we next briefly introduce Apache Spark.

2 Background: Apache Spark

Spark is a JVM-based DISC system exposing a programming model based on Resilient Distributed Datasets (RDDs) [11]. The RDD abstraction provides *transformations* (e.g., map, reduce, filter, group-by, join, etc.) and *actions* (e.g., count, collect) that operate on datasets partitioned over a cluster of nodes. A typical Spark program executes a series of transformations ending with an action that returns a result value (e.g., the record count of an RDD) to the Spark *driver program*. A driver program could be a user operating through the Spark terminal, or it could be a standalone Scala program. In either case, RDDs lazily evaluate transformations by returning a new RDD object that is specific to the transformation operation on the target input RDD(s). Actions trigger the evaluation of an RDD, and all RDD transformations leading up to it. Internally, Spark translates a series of RDD transformations into a DAG of *stages*, where each stage contains some sub-series of transformations until a *shuffle step* is required (i.e., data must be re-partitioned). `reduceByKey`, `groupBy` and `join` are common *stage-breaking* RDD transformations requiring data re-partitioning. The Spark scheduler is responsible for executing each stage in topological order according to data dependencies. Stage execution is carried out by *tasks* that perform the work (i.e., sequence of transformations) of a stage on each input partition. Records composing each data partition are presented to a task in the form of an iterator i.e., Spark follows the record-at-a-time dataflow model of databases. The final output stage evaluates the action that triggered the execution. The action result values are collected from each task and returned to the driver program, which can eventually initiate another series of transformations ending with an action. For fault-tolerance, within stages data is materialized on



(a) With Newt, a MySQL cluster is co-located with the DISC (i.e., Spark) one. The data analyst authors and submits her / his program through the Spark Driver. Spark with Newt instrumentation generates data provenance data and stores it into local MySQL instances. Once the job is completed, a provenance analyst queries the generated provenance graph using the MySQL Driver interface.



(b) With the RAMP instrumentation, provenance records are propagated downstream and saved into HDF. Data and provenance analysts use the same RDD interface. However interactiveness is limited since two different contexts are required.

Figure 1: The Newt and RAMP system approach for DISC provenance.

persistent memory and re-partitioned. Spark additionally allows programmer to *cache* RDDs in memory. When a cached RDD is scheduled for execution, Spark's *BlockManager* eagerly retrieves the saved RDD data instead of triggering the evaluation of preceding RDDs. This mechanism is particularly useful for programs containing iterations.

3 Data Provenance in DISC: RAMP and Newt

Our initial work in adding fine-grained provenance support to Spark leveraged RAMP and Newt designs. During this exercise, we encountered a number of issues, including scalability (the sheer amount of fine-grained provenance data that could be captured and used for tracing), job overhead (the per-job slowdown incurred from data provenance capture), and usability (both provide limited support for provenance queries). Newt operates externally to the target DISC system, making it more general than Titian. However, the Newt design prevents a unified programming environment, in which both the data and its provenance can be queried in the same runtime. RAMP is more tightly integrated into the target DISC system (e.g., Hadoop MapReduce), providing better scalability, but, like Newt, the RAMP design lacks a unified solution for data and provenance analysis.

In the following we present a qualitative analysis of our experience in porting Newt and RAMP over Spark. For a more quantitative evaluation, we refer interested readers to [9]. Figure 1 pictorially summarizes the Newt and RAMP instrumentation of Spark.

3.1 Newt

Newt is a generic framework designed for collecting and querying fine-grained data provenance information from any framework executing data transformations as logical operators. When porting Newt to Spark, we avoided modifications to Spark runtime so that it could be leveraged in different versions of Spark. Newt follows the *agent* model whereby target data transformations are instrument to capture provenance information on operator inputs and operator outputs. Newt provides a simple `addInput` and `addOutput` API accepting data records as input and generating timestamped unique provenance IDs (using hashing). IDs and timestamps are streamed to a *Newt client* and saved into a local log file. Once program execution is complete, Newt uses the timestamp values to infer the temporal order of outputs IDs relative to inputs provenance IDs to reconstruct the original input-output relationships between records. All reconstructed input-output relationship pairs are then loaded into an indexed *association table* stored in a MySQL distributed cluster co-located with the Spark cluster.

Capturing. Within Spark stages, the instrumentation for collecting data provenance using the Newt API is fairly straightforward: we created two special `map` RDD transformations that generate input and output provenance associations for each Spark stage using the `addInput` and `addOutput` API provided by Newt. These provenance associations are then pushed by the Newt client to MySQL once the input partition has been processed to completion. However, transformations such as `reduceByKey` and `join` require a shuffle step, during which all records are materialized. The simple timestsamp-based API of Newt is not effective in generating minimal provenance data because all input records end up being associated with each output record. Newt does provide the ability to add an optional *tag* to each record (for instance to tag records belonging to the same group-by key) so that only records with same tag are linked in the input-output association table. However tags need to be propagated through the shuffle step, which could be supported by either (1) modifying the target program so that shuffle operations (i.e., `reduceByKey`) accept as input pairs in the form $(key, (value, tag))$; or (2) modifying the Spark internals in order to make propagations of tags transparent to users. The latter is the approach that RAMP (and Titian) took, and will be explained in Section 3.2.

Querying. Provenance queries are executed in Newt as a series of SQL joins executed over the association tables stored in the MySQL cluster. Queries are issued from a *MySQL Driver* node: a logically separated entity from the Spark Driver. We found this approach suboptimal from a usability perspective because: (1) users have to setup a MySQL cluster together with the DISC system cluster; (2) analysts needs to be proficient on two systems in order to debug their programs; (3) for the system to be able to properly compose tracing query plans, stored association tables need to be explicitly linked to form a dependency graph, mirroring the position they occupy in the original input program; and (4) populating indexed MySQL tables starting from log files is time consuming (on the order of minutes to hours depending on the data volume). The latter two points make the Newt design difficult to use in interactive sessions.

Additionally, the raw intermediate data is not available in the Newt design, making it only relevant for tracing back to the input datasets; further limiting its use in a (stepwise) debugger like setting. We did not explore checkpointing (saving) the intermediate data in our Newt to Spark port since we had already hit a scalability bottleneck when capturing the provenance alone.

Lessons Learned. Summarizing, from our initial experience of adding data provenance to Spark through Newt we concluded the following:

- To minimize the overhead of tapping record pipelines, it is better by default to *capture provenance information at stage boundary only*, and eventually give to users the ability to manually inject additional capturing points if necessary.
- Without any modification to Spark internals or user code, Newt timestamp-based approach is not able to produce minimal input-output relationship tables for transformations requiring shuffling. As a consequence, capturing and querying have higher overheads wrt more optimal solutions where only minimal

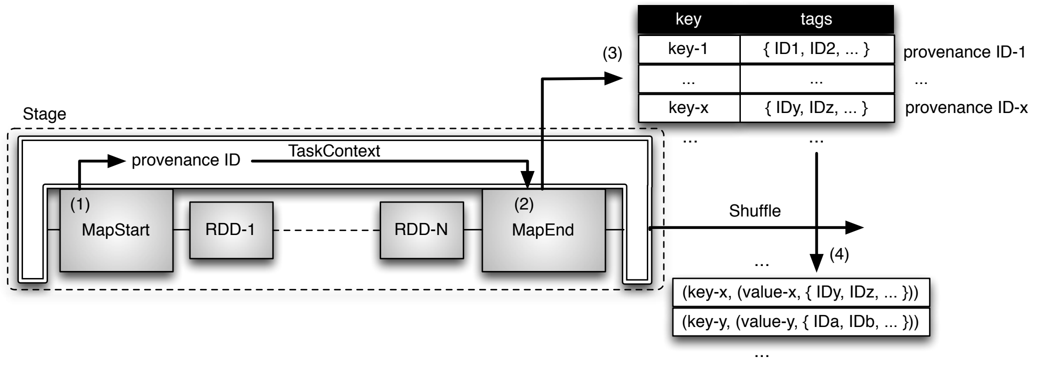


Figure 2: In RAMP Provenance IDs are propagated downstream as record TAGS. IDs are first added to the TaskContext (1). `MapEnd` pulls IDs from the context (2) and populate a local (to each task) hash table (3). Finally, collected provenance identifiers are attached to the proper key-value pair (4).

Figure 3: RAMP approach of propagating provenance through the TaskContext and the shuffle step.

provenance information are saved.

- Data provenance capturing and querying have to be executed on the same (DISC) system if we want to allow interactivity both to end users and to higher-level debugging toolkits.
- Intermediate data records (i.e., records stored into shuffle files) linked to provenance IDs have to be made accessible by users in order to provide better insight into intermediate RDD transformations.

3.2 RAMP

The initial RAMP implementation was specifically tailored for MapReduce workflows, but its design can easily be ported over Spark. Unlike Newt, RAMP integrates with the target DISC system (e.g., Spark) to efficiently tag records with chains of (nested) provenance IDs that are propagated with the data records through the dataflow to the final record outputs. These provenance chains can be seen as materializing backward tracing joins between (virtual) association tables at the final program output. Indeed, with such design, backward tracing provenance queries can be answered very efficiently. Conversely, forward queries are difficult and inefficient to support.

Capturing. When a Spark program is submitted for execution (i.e., when an action is called on an RDD), the RDD dependencies are analyzed and two new `map` RDDs (`rampMapStart` and `rampMapEnd`) are injected into the program before and after (respectively) any RDDs that translate into a single stage. For input datasets that reside in Hadoop HDFS, Spark uses a `HadoopRDD` that emits the data record along with an *offset* of the record in the input file. We inject a `rampMapStart` RDD that consumes these records and uses the partition identifier and offset as provenance ID. Since RAMP does not timestamp records, input provenance IDs need to be propagated downstream to the stage output, where associations are made with the relevant output provenance IDs. We implement provenance propagation by adding the provenance ID of the current input record to the Spark *TaskContext*, as shown in Figure 3 (1). Since records are evaluated one-at-a-time, each provenance ID is relevant to all output records sent to `rampMapEnd` at the stage output[1] (2). `rampMapEnd` is responsible for associating each output record key with all relevant input provenance IDs (3). All the provenance IDs accumulated into local hash tables have to be propagated as well, but this time through the shuffle step. To achieve this the *ExternalSorter* component in Spark had to be modified so that records can be written in the form $(key, (value,$

[1]Note that Newt does not make this assumption, and instead relies on timestamps to infer (indeed overestimate) input-to-output associations.

provenance ID) in the shuffle file, where provenance ID contains all the IDs accumulated into the has table under key (4), i.e., previously stored provenance IDs are nested and used as a new provenance ID. Note that only nested IDs need to be added to each record, since the hash of the key can be recomputed on the fly.

In the following stage, the *ShuffleReader* and *Aggregator* components had to be changed to make them aware of the new record format. Apache Spark computes aggregates (a similar argument holds for joins) by iterating over all records while updating the aggregate value linked to each key. The final aggregate values are then surfaced to the successive RDD transformations as iterator. Hence, after the shuffling step provenance information are generated in two phases. In the first phase, while the aggregator iterates through records, IDs are removed so that aggregation can be computed as in regular Spark. The provenance IDs (each composed by the sequence of IDs coming from the previous stage `rampMapStart`) are then used together with the key to populate a new local hash table, as in `rampMapEnd`. Once the aggregation phase is completed, we store the generated hash table containing all provenance IDs into a buffer in the TaskContext. In the second phase, when the aggregated records are emitted as iterator, a `rampReduceStart` RDD previously injected in the workflow pulls the current provenance ID from the buffer, appends a new provenance ID to it, and stores the updated provenance chain into the TaskContext so that the `rampMapEnd` transformation at the end of the stage can use it. The process repeat if other stages follow. The `rampMapEnd` in the final stage materialize all nested provenance IDs in HDFS.

Querying. Provenance queries are implemented in RAMP by traversing and unnesting HDFS residing provenance IDs. For this task, external scripts can be used, but interestingly also the DISC system itself. This later approach is closer to our target of providing provenance querying capabilities withing the DISC framework. Note however that in both cases, only backward tracing queries are supported efficiently, while forward tracing requires to scan and unroll all provenance IDs. Additionally, RAMP model allows to access the (map) input and (reduce) output data records connected with provenance IDs but no intermediate records.

Lessons Learned. The RAMP porting provided us the baseline for the Titian implementation.

- The offloading of provenance information to HDFS-stored files is simple and effective, and allows to use the same DISC system for provenance querying.
- The approach of propagating nested provenance IDs introduces not negligible overheads into the runtime, moreover redundant information are added to shuffle files.
- Both backward and forward tracing queries should be efficiently supported. Additionally, provenance queries should be expressible in a declarative language leaving the system to decide how to properly execute (and optimize) them.

4 Titian

Titian integrates with the Spark runtime to provide efficient data provenance support. Submitted programs are rewritten to include lineage capturing (map) transformations at stage boundaries that generate provenance association tables, which are stored directly into Spark's in-memory store (BlockManager). Additionally, partition identifiers are propagated with data records through shuffle steps in order to optimize tracing queries.

Provenance information is exposed as RDDs, on which all Spark native transformations, including some additional tracing capabilities, can be applied. This approach makes program execution and debugging a continuous process carried out interactively using the same Spark context. Figure 5 describes the Titian design.

4.1 Capturing

The entry point of Titian is the `LineageContext` that wraps the original `SparkContext` to enable data provenance capabilities. When a program is submitted for execution, Titian rewrites the program by injecting

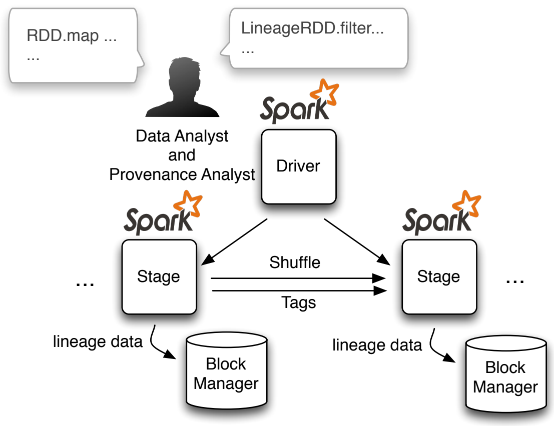


Figure 4: With Titian, users can author DISC programs and analyze their provenance trace interactively within the same (Spark) context.

Figure 5: Titian design for DISC provenance.

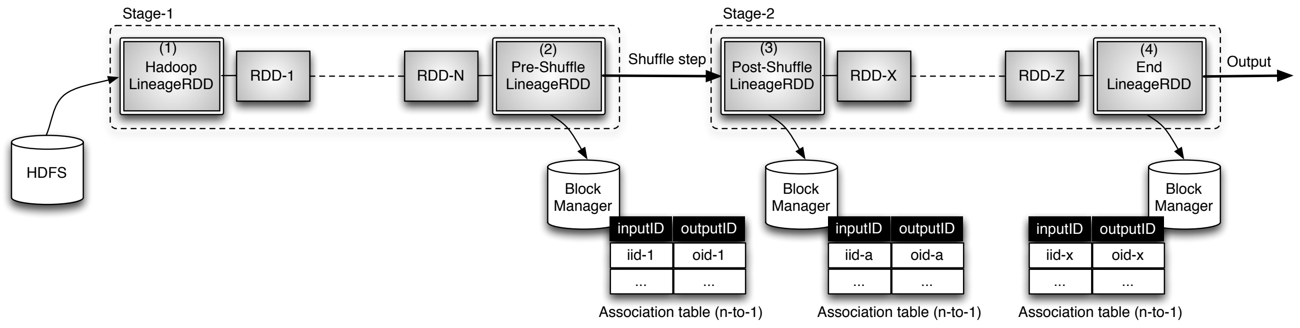


Figure 6: A two stage program instrumented with Titian.

LineageRDD transformations at stage boundaries, as shown in Figure 6. LineageRDDs can be classified into four types based on where they are positioned in the program.

(1) Program Input. Titian adds a HadoopLineageRDD or a ParallelizeLineageRDD, based on whether input records are fetched from HDFS or directly from the driver. The implementation of these LineageRDDs is similar to the `rampMapStart` transformation previously described in Section 3.2. For each input data record, an unique INPUTID (such as the filename, partition id and offset in an HDFS file) is propagated to the stage output using the TaskContext.[2]

A stage ends when one of the following operations occur: *reduction* (i.e., `reduceByKey`), *generic aggregation* (e.g., `groupBy`), or *co-grouping* operators (e.g., `join`, `union`, etc.). The differences between the three are both in the implementation and in the output format. Spark injects a combiner step when a reduction operation is requested, while generic aggregation is performed without a combiner. In both cases, the successive stage starts with an RDD iterating over a sequence of pairs where the first element is the grouping key, while the second element is the related aggregate value. Conversely, co-grouping (similarly to generic aggregation) uses no combiner, and outputs a sequences of pairs composed by the grouping key and an iterator over all the values. Join results are produced by flattening the co-group output. Next, we describe how we take these output results, generate identifiers for them, and associate them with the relevant input record identifier.

[2]Input records follow at 1-to-1 mapping, avoiding the need to generate association tables at this point.

(2) Pre-Shuffle. Before the shuffle step, Titian injects a proper LineageRDD based on the operation the program is implementing: i.e., PreReduceLineageRDD in case of `reduceByKey`, PreGroupLineageRDD for `groupBy`, and PreCoGroupLineageRDD for a transformation requiring co-grouping. Pairs of ids (INPUTID, OUTPUTID) are generated and buffered in memory by each operator, where INPUTID is the tag of the record propagated from some other LineageRDD upstream, and OUTPUTID is the hash of the key of the record. Additionally, Titian attaches to each shuffled record a partition ID so that post-shuffle records can be efficiently joined with pre-shuffle records i.e., the partition ID indicates which pre-shuffle partitions need to be considered; without this information, we would need to join with all pre-shuffle partitions, which is expense for tracing starting from a small set of output records.

(3) Post-Shuffle. In the successive stage following the shuffle step, Titian rewrites the program workflow by substituting Spark default ShuffleReader with one of the three ReduceShuffleReader, GroupShuffleReader, and CoGroupShuffleReader. Additionally, a PostReduceLineageRDD, PostGroupLineageRDD and PostCoGroup LineageRDD are added to the program based on its semantics. ReduceShuffleReader and GroupShuffleReader follows the same logic of the previously introduced RampReduceStart. Instead, when co-grouping, Spark does not aggregate values by key, but instead it directly returns an iterator over key-value pairs, in which the values are iterators of all the records having the same key. Because of this, each of this records contain the partition ID previously attached during the pre-shuffle phase. PostCoGroupLineageRDD implementation therefore (1) unrolls each record-iterator and save the partition identifier of each record in an in-memory buffer; (2) generates a new unique identifier and stores it into the TaskContext; and (3) emits the new key-value pair without partition identifiers into the successive RDD transformations.

(4) Program Output. At the end of each program Titian injects an EndLineageRDD creating a pair (INPUTID, OUTPUTID) for each data record. Additionally, EndLineageRDD attaches OUTPUTID to the record so that users can, at query time, connect records to provenance IDs. Since the input-output relationship is 1-to-n (e.g., due to a flat map transformation), an association table is explicitly materialized in the BlockManager.

Buffering and Lineage Storage. In order to reduce the overhead of capturing lineage, Titian stores provenance information in-memory into Spark's BlockManager. Additionally, to reduce the number of objects created at runtime, we have extended Spark's worker nodes runtime (Executors) with a pool of *bytebuffers* of different length. When a partition is scheduled for execution, at the first call each LineageRDD initializes a local in-memory buffer by fetching a bytebuffer from one of the pools. Each input-output pair of provenance IDs generated by the LineageRDD is then stored into the local buffer. When a partition is completed, local buffers are asynchronously materialized in the BlockManager after that a LineageRDD-specific compression logic is executed over the provenance records. Specifically, we store pre- and post-shuffle provenance data in a nested format in order to not waste memory space over redundant information. In fact, pre- and post-shuffle LineageRDD operators creates a sequence of (INPUTID, OUTPUTID) pairs in which the same OUTPUTID can appear multiple times based on how many times values with the same key exist. The trade off is that such technique increases the overhead of accessing data at query time since records need to be unnested, but other techniques such as *targeted joins* (described in Section 4.2) can be used to speed-up query processing.

4.2 Querying

Once the DISC program with provenance capturing enabled is completed, Titian surfaces provenance data as LineageDataRDDs: a specific RDD equipped with provenance-specific operations such as `TraceBackward` or `TraceForward` additionally to regular RDD transformations. To enable provenance queries, upon program completion, Titian (1) labels all LineageRDDs injected into the original program as cached; and (2) generates a dependency graph formed by all such LineageRDDs now surfaced as LineageDataRDD references. With (1), Titian is basically instrumenting Spark to fetch the provenance data available in the BlockManager when an

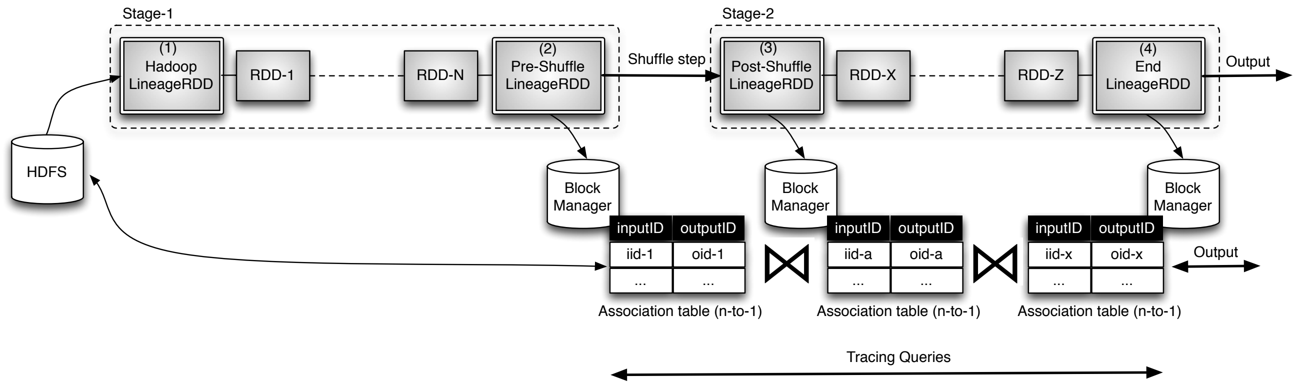


Figure 7: Tracing queries are implemented in Titian as a sequence of (distributed) joins between association tables.

action requires to compute a LineageDataRDD[3]; since provenance data is exposed as RDDs, we simply leverage Spark to query it, without asking users to manually specify the dependencies between associations tables (as for instance Newt and RAMP require).

Starting from a target LineageDataRDD of interest, analyst can submit tracing queries using the `TraceBackward` or `TraceForward` transformations.[4] Furthermore, regular RDD transformations such as `filter` can be used to remove out all provenance records that are not of interest (for instance by specifying the OUTPUTIDs of the records that need to be traced). Taking the backward direction as example, when a user call the method `traceBackard()` over a (filtered) LineageDataRDD reference, Titian's *QueryPlanner* schedules a left semi-join between the current LineageDataRDD and the preceding one in the dependency graph. If users prefer to execute more than one step backward, `traceBackward(numSteps)` can be called where `numSteps` specifies the numbers of joins to be executed. Finally, a full trace backward up to the initial input IDs can be executed by calling `fullTraceBackward()`. Figure 7 is a replica of Figure 6 where we show how joins are executed using the association tables generated from the two-stages program.

Query Planning and Optimizations. When a tracing query is issued for evaluation (e.g., after an action is invoked on a LineageDataRDDs reference returned from a `traceBackword` call), Titian generates a query plan that attempts to minimize data movement, and unrolling nested records only when necessary. Titian's QueryPlanner avoids Spark's shuffle join implementation when tracing within a stage i.e., from the stage output to the stage input (or vice-versa), since the respective associations already join locally. A shuffle join is needed when tracing between shuffle steps. However, instead of using Spark's native shuffle join, we implemented a *direct shuffle join* that uses the partition identifier information to directly traces to only the relevant partitions on the other side of the shuffle step. For example, if we are tracing back from a record with key "foo", then it might be the case that not all pre-shuffle steps generated such a key, in which case the partition identifiers will inform the direct shuffle to avoid joining those partitions with the "foo" records. Additionally, indexes are create at capture time to speed-up the joining process when tracing backward. More details on query planning and optimizations can be found in [9].

Accessing Original Data Records. Titian enables users to inspect the data records that each provenance ID is linked to, including intermediate data, without introducing any overhead in the capturing phase. In fact, while input and output records are directly connected to the related provenance ID by construction, intermediate

[3]Note that in Spark caching is usually request at program-time *before* an RDD is scheduled for execution. Titian instead asks Spark to cache LineageRDD *after* they are computed. Indeed LineageRDDs internally materialize provenance data into the BlockManager, i.e., without using Spark support. Since LineageRDD are manually saved, Spark is unaware of the checkpoints and therefore fault tolerance mechanisms work as usual.

[4]The tracing transformations are simply syntactic sugar over native Spark (filtered) joins.

records (i.e., records produced as output of a intermediate stage and consumed by successively scheduled stages) can be retrieved by directly accessing (saved) shuffle files. Spark, in general, maintains shuffle files in cluster memory for fault tolerance. Such files survive after the execution of the target program and therefore can be read by provenance queries. This is possible exclusively because one unique context is used for both data and provenance analysis.

Accessing the data linked to a set of provenance IDs is possible in Titian by calling the `showData` method over the target LineageDataRDD object. Underneath, Titian's query planner issues a join between the LineageDataRDD and the related object reference pointing to the data. For example, a call to `showData()` over an HadoopLineageDataRDD object will issue a join between the provenance records of the HadoopLineageDataRDD object, and the input dataset stored in HDFS. Similarly, a call to `showData()` over a PreShuffleLineageDataRDD issues a join with the shuffle file. Note that the Titian framework automatically maintains the reference to all (intermediate) files and objects storing data records that can be eventually requested by users during tracing.

5 Considerations and Future Directions

Newt, RAMP, and Titian provided different contributions over the state of the art. Newt showed that a *portable*, external library can be developed such that (different) DISC systems can be easily instrumented to capture provenance. Higher-level tooling can then target such library and seamlessly work over different DISC platforms. In our experiments Newt however showed *poor scalability*. This is both the result of its portable design, and the choice of using MySQL as lineage capturing backend. This later choice of separating the query subsystem from the DISC system, lowers its usability and makes features such as visualization of intermediate data difficult to achieve.

RAMP chose a more integrated design with the host DISC system at the cost of less portability. Propagating full provenance information downstream introduces however major overhead on the running program. Lastly, while provenance queries are supported in RAMP using the same DISC language, the original RAMP design did not consider forward tracing queries, but only (already materialized) backward queries.

Titian implementation merges the advantages of both the Newt and RAMP approaches: Titian integrates the provenance capturing infrastructure within the host DISC system, but instead of propagating tags, it materialized input-output association tables in memory. Titian's integration with Spark does not limit Spark's scaling capabilities (we experimented with dataset up to 1TB) with an average overhead of 30%. Titian choice of surfacing provenance data as RDD, greatly improves usability by allowing users to employ the same Spark Context for both program authoring and provenance analysis. Additionally, the optimizations implemented in Titian brings interactive speed evaluation of tracing queries. However, we think that still many open questions remain before truly reaching the goal of having a industry-ready data provenance library for DISC systems. We next sketch few possible directions for future work on the three dimensions of *portability*, *usability* and *scalability*.

Portability. We found Titian tailored integration with Spark low-level (RDD) API both a blessing and a curse: upgrading Titian to newer versions of Spark is in general not easy; similarly enabling fine-grained provenance capturing over Spark's graph, ML, or relational API is not trivial and requires re-implementing major parts of the framework. We are starting to see more applications requiring mixed programming models. While Spark does provide a unifying infrastructure to execute mixed applications, to our knowledge no solution exists which is able to trace data provenance effectively end-to-end through such mixed programs. The problem is even exacerbated when instead of a single system (Spark), multiple specialized frameworks are used to implement a data analytics pipeline. While each framework may have the ability to provide some provenance information, unifying it in a single interactive session and language is an open problem.

Usability. While we started the exploration of high-level tooling exploiting provenance for debugging DISC program [4–6, 8], we think that much work still have to be done in the field. For instance, in [6] we introduced a system for automatically detecting the minimum set of failures inducing inputs given a user-provided test function. It would be interesting to see the application of similar techniques over different domains beyond software engineer (e.g., outliers detection or data cleaning), and generating proper domain-specific explanations or actions, beyond a minimum set of evidences of a failure.

Scalability. From our experience with Titian, we found that capturing fine-grained provenance data can generate extremely large provenance graphs; in the same order of the original input data. Instead of focusing on integrating the capturing infrastructure as close as possible to the data source, another possible solution to achieve better scalability is to exploit application information to summarize the provenance data [3], or, alternatively, to push provenance queries into the capturing phase, so that only a subset of the provenance is actually captured. These approaches may lower the generality of the type of analysis that can be carried on over the provenance graph, but for certain applications (or scales) could be the right solution.

References

[1] Apache Hadoop. `http://hadoop.apache.org`

[2] Apache Spark. `http://spark.apache.org`

[3] R. Diestelkämper, M. Herschel and P. Jadhav. Provenance in DISC Systems: Reducing Space Overhead at Runtime. *TaPP*, 2017.

[4] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein and M. Kim. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. *ICSE*, 2016.

[5] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim. Debugging Big Data Analytics in Spark with BigDebug. *SIGMOD*, 2017.

[6] M. A. Gulzar, M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim. Automated Debugging in Data-Intensive Scalable Computing. *SoCC*, 2017.

[7] R. Ikeda, H. Park and J. Widom. Provenance for Generalized Map and Reduce Workflows. *CIDR*, 2011.

[8] M. Interlandi, M. A. Gulzar, M. Kim and T Condie. Optimizing Interactive Development of Data-Intensive Applications. *SoCC*, 510-522, 2016.

[9] M. Interlandi, A. Ekmekji, K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. Millstein and T Condie. Adding data provenance support to Apache Spark *VLDBJ*, 2017.

[10] L. Dionysios, D. Soumyarupa and Y. Kenneth. Scalable Lineage Capture for Debugging DISC Analytics *SOCC*, 2013.

[11] M. Zaharia, M .Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *NSDI*, 2012.

Data Center Diagnostics with Network Provenance

Ang Chen*, Chen Chen◇, Yang Wu†, Andreas Haeberlen†,
Limin Jia‡, Boon Thau Loo†, Wenchao Zhou♯

*Rice University, ◇Recruit Institute of Technology, †University of Pennsylvania,
‡Cargenie Mellon University, ♯Georgetown University

Abstract

Diagnosing problems in data centers has always been a challenging problem due to their complexity and heterogeneity. Among recent proposals for addressing this challenge, one promising approach leverages provenance*, which provides the fundamental functionality that is needed for performing fault diagnosis and debugging—a way to track direct and indirect causal relationships between system states and their changes. This information is valuable, since it permits system operators to tie observed symptoms of a faults to their potential root causes. However, capturing provenance in a data center is challenging because, at high data rates, it would impose a substantial cost. In this paper, we introduce techniques that can help with this: We show how to reduce the cost of maintaining provenance by leveraging structural similarities for compression, and by offloading expensive but highly parallel operations to hardware. We also discuss our progress towards transforming provenance into compact actionable diagnostic decisions to repair problems caused by misconfigurations and program bugs.*

1 Introduction

Data center diagnostics has always been a challenging problem: with changing workloads, complex protocols, and a heterogeneous mix of devices, system faults can happen in data centers for many different reasons, including design flaws, software bugs, and security vulnerabilities. The recent move towards software-defined networking (SDN) has further added to this complexity: networks are now fully programmable. This provides the operator with great flexibility and power, but, at the samew time, a buggy controller program can introduce subtle malfunctions into the network that can be very difficult to find.

The research community has recognized this challenge [19], and has responded by developing a line of new diagnostic tools [20, 26, 27]. One approach, in particular, is based on *provenance* – a concept that was originally developed in the database community [4] but that has recently found new uses in the networking domain [9,52,59,61]. Provenance provides the fundamental functionality required for performing fault diagnosis and debugging—the capability to "explain" the existence (or change) of system state. Provenance is a form of metadata that tracks direct and indirect causal relationships between system states and state changes. Such information is of great value, especially in complex systems like data centers, since it permits system operators to tie observed faults to their potential root causes, and to assess the damage that these faults may have caused.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Provenance has been successfully applied to a wide range of areas, including distributed systems. In earlier work [8], we have already discussed some of the challenges and sketched generalizations of the provenance model that can help to address them. This paper focuses on our experience in adopting provenance for diagnosing data centers, and the unique challenges we faced during this process. First of all, given the ever-increasing amount of data processed and stored at data centers, the storage and computation overhead for provenance maintenance is high. This is especially important when provenance is used to track *data-plane* events – that is, the flow of actual network packets, as opposed to merely control messages. It is not unusual for the data plane to process data at a rate of several terabits per second, whereas control-plane messages are much less frequent.

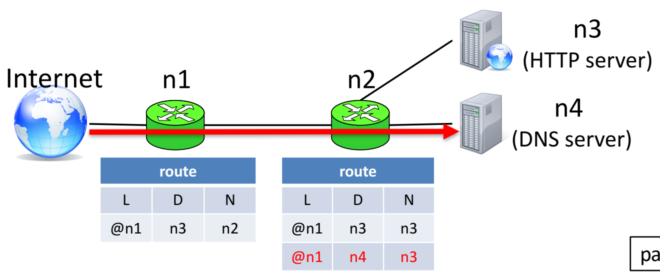
We consider several techniques that could reduce the overhead of maintaining provenance. Besides traditional content-level compression techniques, such as gzip, we explore compression opportunities that leverage the structure of provenance trees [10]. We observe that the provenance of different packets share significant similarities in their structures, presenting opportunities for provenance compression across different provenance trees. For example, whenever a new packet traverses the network, its entire provenance tree is created and maintained. However, it is not hard to realize that *all* the packets in the same flow (i.e., packets with same source and destination) would take the same route and therefore have almost identical provenance trees. Therefore, we can achieve massive storage savings if we remove the observed redundancy when maintaining provenance. Maintaining provenance with security guarantees adds another layer of overhead—cross-nodes communications need to be cryptographically signed and acknowledged to be resilient to tampering of provenance [59]. To amortize the cost of these cryptographic operations, we leverage the use of Merkle Hash Trees (MHTs) and offload embarrassingly parallel operations to hardware such as FPGAs [7].

Another challenges is that provenance, at least in its raw state from runtime recording, often times is still hard for human users to understand. In fact, provenance trees can be quite large for system admins to reason about. For example, in an average-sized SDN, the provenance graph of why a specific packet p arrived at server s can easily contain hundreds of vertices. Identifying the root cause of an observed system problem is still a non-trivial task even for experts. To address this challenge, we consider approaches that distill from provenance compact and actionable information or suggestions for system admins. We explore an approach called differential provenance [9] that repairs misconfigurations through differential analysis on the provenance for working and non-working execution instances. The result of such differential analysis pinpoints the exact root causes that caused two execution instance to diverge. Meta provenance [50] takes a step further and reason about not only causality among data but also dependencies on program code. We treat the code as another kind of data, where syntactic elements of the program is captured as meta tuples and its semantics as meta rules. Such capability of reasoning dependencies of execution results on different components in a program empowers us to automatically repair bugs in the program. Given that there are literally an infinite number of possible way to modify a program, it is infeasible to use meta provenance to repair *all* possible bugs. We focus on bugs that do not require significant rewrite of the program.

In the remainder of the paper, we briefly introduce the concept of provenance and our system model in Section 2. Section 3 presents our recent progress in reducing the provenance maintenance overhead when applying provenance to data center networks. Section 4 further discusses our effort in transforming provenance into compact actionable diagnostic decisions. Finally, we present related work in Section 5 and conclude the paper in Section 6 with a discussion on future research directions in applying provenance to system diagnosis.

2 Background

Provenance is a general concept that has been applied to many kinds of systems written in many different languages. For ease of exposition, we will use a declarative language—specifically, *network datalog (NDlog)* [30]—to explain our approach. The main reason is that, in this language, it is relatively easy to introduce the causality relations that provenance needs to track.



route

L	D	N
@n1	n3	n2

route

L	D	N
@n1	n3	n3
@n1	n4	n3

Figure 1: Example scenario.

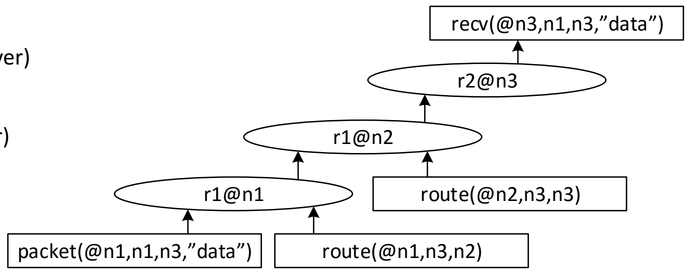


Figure 2: Example provenance.

In NDlog, the state of a node (switch, controller, or server) is modeled as a set of *tables*, each of which contains a number of *tuples*, such as configuration state or network events. For instance, an SDN switch might contain a table called `flowtable`, where each tuple represents a flow entry. Tuples can be manually inserted or programmatically derived from other tuples; in the former case, we refer to them as *base tuples*, and in the latter case as *derived tuples*.

NDlog programs consist of *rules* that describe how tuples should be derived from each other. For example, the rule `r1: a(@X,P):-b(@X,Q),Q=2*P` says that a tuple `a(@X,P)` should be derived on node `X` whenever a) there is also a tuple `b(@X,Q)` on that node, and b) `Q=2*P`. The ID after the `@` symbol specifies the node on which the tuple resides, and the `:-` symbol is the derivation operator. Rules may include tuples from different nodes: for instance, `r2: c(@X,P):-c(@Y,P)` says that tuples in table `c` on node `Y` should be sent to node `X`.

In NDlog, the provenance of a tuple is easy to see, even syntactically: if a tuple (say, `a(@X,5)`) was derived using rule `r1` from above, then it must be the case that all the preconditions in `r1` were true (here, `b(@X,10)`), and all the constraints in `r1` were satisfied (in this case, `10=2*5`). This concept can be applied recursively to explain the existence of the preconditions until a set of base tuples (such as configuration settings or packets from external links) is reached that cannot be explained further. The result is a *provenance tree*, in which vertices represent tuples or rule derivations that generate these tuples, and edges represent direct causality.

Example Provenance. Figure 1 shows a (very simple) illustrative scenario with a network that connects two servers (one DNS server `n4` and one HTTP server `n3`) to the Internet. Here, switch `n1` and `n2` each have a routing table – each routing entry records the next hop towards a given destination. Each node runs an NDlog program consisting of two rules:

```
r1: packet(@N,S,D,Data) :- packet(@L,S,D,Data), route(@L,D,N).
r2: recv(@L,S,D,Data)   :- packet(@L,S,D,Data), D==L.
```

Rule `r1` forwards a packet to the next hop, based on the destination of the packet and the local routing information. Rule `r2` receives a packet if the packet's destination is identical to the local address.

Figure 2 sketches the provenance of an HTTP packet that was sucessfully delivered at the HTTP server `n3`. It reads roughly as follows: the packet arrived at switch `n1` and was routed to switch `n2` because it matched a routing entry `route(@n1,n3,n2)` installed at `n1`. The packet was further forwarded to and delivered at `n3`. Due to a faulty routing entry on switch `n2` (highlighted in red in Figure 1), DNS requests are misrouted to the HTTP server. To diagnose this problem, the operator can similarly query the provenance of a DNS packet, which would reveal that DNS requests were misrouted by `n3` after they arrive `n2` due to the faulty routing entry `route(@n1,n4,n3)`.

Application of Provenance in Diagnosing Distributed Systems. In prior work, we have already applied provenance in several common diagnostic scenarios. ExSPAN [61], our first solution, is a general-purpose system that can maintain and query provenance for large distributed systems. SNP [59] added security features that can prevent an adversary from tampering with the provenance to cover her tracks; this made it possible to use provenance not just for diagnostics but also for forensics. Y! [52] introduced the concept of *negative* provenance, which can explain not only why a certain event did occur, but also why an event *failed* to occur.

3 Provenance Maintenance

The potentially large maintenance cost has been a long-standing challenge in provenance research [6] [24]. This becomes particularly challenging for systems like data centers that deal with frequent and high-volume data packets. When there are streams of incoming events, the provenance maintenance overhead, in terms of both computation and storage overhead, can quickly become prohibitively expensive. We present in this section two mechanisms for harnessing the maintenance overhead of provenance in data centers.

3.1 Provenance Compression through Program Analysis

Our first approach in reducing the maintenance overhead of provenance is based on the observation that there is much redundancy when maintaining provenance in a network. For example, when two packets share the same source and destination addresses, their provenances are almost identical: they share all the intermediate traversed nodes, and only differ at the packet payload. This suggests opportunity for significant storage reduction.

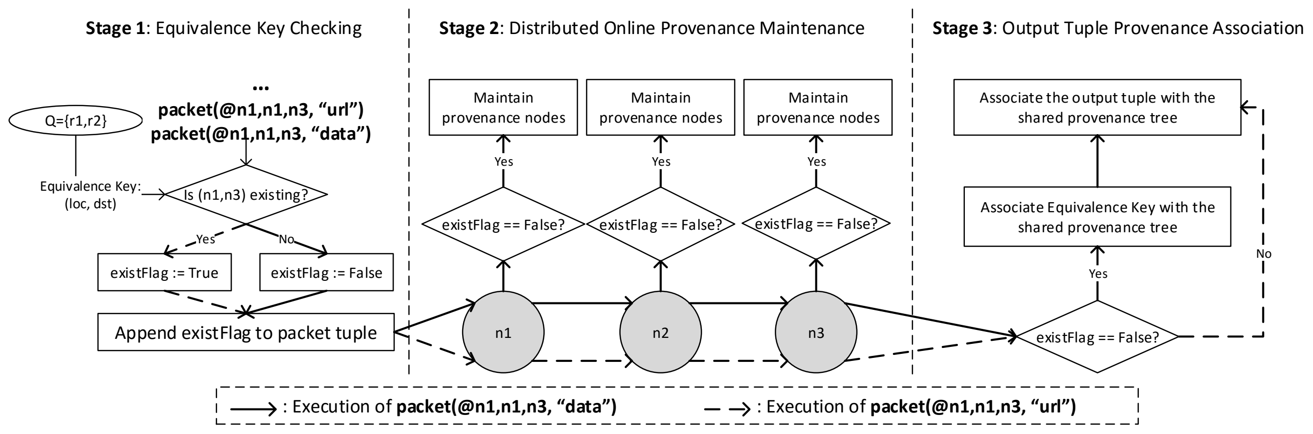
Based on this observation, we introduce an equivalence-based, online provenance compression mechanism, reducing the storage overhead of provenance effectively and efficiently [10]. This mechanism (1) groups provenance trees into equivalence classes, so that the storage of provenance trees of the same equivalence class could be compressed by sharing one representative provenance tree; and (2) enables efficient equivalence identification at runtime through the value comparison of *equivalence keys*—a subset of attributes of the input event identified by static analysis at compile time. Our experimental results demonstrate that this compression technique allows for significant—often orders of magnitude—storage reduction.

Provenance equivalence. One of our contributions is to show that the equivalence of provenance trees can be determined by comparing the values of a subset of attributes of the input tuple, if the program is written as a *Distributed Event-driven Linear Program'* (DELP). Intuitively, a DELP is an NDlog program that satisfy the following constraints:

- **Rules are event-driven.** Each rule *r* can be specified in the form: $[head]\ :-[event], [conditions]$, where $[event]$ is a body relation designated by the programmer, and $[conditions]$ are all non-event body atoms.
- **Rules are linearly dependent.** For consecutive rules r_i and r_{i+1}, the head relation *hd* of r_i is exactly the event relation in r_{i+1}.
- **Events don't have side-effects.** For each head relation *hd* in any rule r_i, *hd* is not used as a non-event relation in another rule r_j.

In a typical network application, events are high-speeding streaming relations that are often not materialized; non-event relations represent the network states. For example, in the packet forwarding program, the events are the `packet` tuples that flow through the network, and the *route* relation is a non-event relation, and is either updated manually or through a network routing protocol. In either case, it changes slowly compared to the fast-rate incoming packets. The distinction between events and non-event relations is provided by the programmer or is decided through profiling during program execution.

For DELP programs, we define the equivalence relation between two provenance trees *tr* and *tr'* as follows: *tr* and *tr'* are equivalent, if and only if they only differ at two nodes: the root node that represents the output tuple and the leaf node that represents the input tuple. This definition implies that two equivalent provenance trees are structurally identical too. For example, in Figure 1, if we insert another packet *packet*$(@n1, n1, n3, \text{“}data2\text{”})$, it will generate another provenance tree that is equivalent to the tree in Figure 2 – the only difference is the payload of the leaf *packet* tuple and the root *recv* tuple.



Stage 1: Equivalence Key Checking
Stage 2: Distributed Online Provenance Maintenance
Stage 3: Output Tuple Provenance Association
Q={r1,r2}
...
packet(@n1,n1,n3, "url")
packet(@n1,n1,n3, "data")
Equivalence Key: (loc, dst)
Is (n1,n3) existing?
Yes
No
existFlag := True
existFlag := False
Append existFlag to packet tuple
Maintain provenance nodes
existFlag == False?
n1
n2
n3
Associate the output tuple with the shared provenance tree
Associate Equivalence Key with the shared provenance tree
: Execution of packet(@n1,n1,n3, "data")
: Execution of packet(@n1,n1,n3, "url")

Figure 3: An example execution of the packet forwarding program in Section 2. The program is first triggered by *packet*$(@n1, n1, n3,$ "data"), followed by *packet*$(@n1, n1, n3,$ "url").

Based on the definition, the equivalence of two provenance trees can be determined through direct comparison: if all vertices other than the leaf tuple and the root tuple are identical, the two trees are equivalent. However, such comparison is inefficient, especially when provenance trees are large and the incoming tuples arrive at a fast rate, as is often the case with today's data centers.

We identified that, for DELP programs, equivalence classes can be determined by a subset of attributes in the events. We call these attributes *equivalence keys*, and show that they can be identified through static analysis of the program that generates the provenance trees. For example, the packet forwarding program in Section 2 is a DELP, and the equivalence keys are L, D of the *packet* tuple.

Online provenance compression. Once the equivalence keys are identified, we adopt an online provenance compression scheme that compresses equivalent (distributed) provenance trees. In our compression scheme, the execution of a DELP, triggered by an event tuple *ev*, is composed of three stages. Figure 3 presents an example execution of the provenance compression scheme, consisting of two packets traversing the network topology (from $n1$ to $n3$) in Figure 1. *packet*$(@n1, n1, n3,$ "data") is first inserted for execution (represented by the solid arrows), followed by *packet*$(@n1, n1, n3,$ "url") (represented by the dashed arrows). The three stages of online compression are logically separated with vertical dashed lines.

- **Stage 1: Equivalence keys checking.** Upon receiving an input event ev, the runtime system first checks whether the value of ev's equivalence keys have been seen before, by checking a hash table that stores all seen equivalence keys. If ev's equivalence keys have a value that already exists in the hash table, a Boolean flag $existFlag$ will be created and set to $True$. This $existFlag$ will accompany ev throughout the execution, notifying downstream nodes to avoid maintaining the concrete provenance tree. Otherwise, $existFlag$ is set to $False$. For example, in Figure 3, when the first packet $packet(@n1, n1, n3,$ "data") arrives, its equivalence keys ($n1$, $n3$) have never been encountered before, so its $existFlag$ is $False$. But $existFlag$ of the second packet $packet(@n1, n1, n3,$ "url") is set to $True$.
- **Stage 2: Online provenance maintenance.** For each rule r triggered in the execution, we selectively maintain the provenance information based on $existFlag$'s value. if $existFlag$ is $False$, the provenance nodes are maintained locally. Otherwise, no provenance information is maintained at all. For example, in Figure 3, when *packet*$(@n2, n1, n3,$ "data") triggers rule $r1$ at node $n2$, the $existFlag$ is $False$. Therefore, we maintain its provenance. In comparison, we simply execute $r2$ without recording any provenance information for *packet*$(@n2, n1, n3,$ "url").
- **Stage 3: Output tuple provenance maintenance.** For the execution whose $existFlag$ is $True$, we need to associate its output tuple to the shared provenance tree maintained by previous execution. To do this, we use a hash table *hmap* to store the reference to the shared provenance tree, wherein the key is the hash

System	Goal	Information offered	Capabilities				
			Secure evidence	**Supports forensics**	**Covers entire Internet**	**Fine-grained entities**	**Fine-grained traces**
Tulip [32]	Fault localization	Loss, delay, reordering	×	×	✓	✓(Routers)	✓(Packets)
NetPolice [57]	Traffic differentiation detection	Loss	×	×	✓	× (ISPs)	× (Flows)
SPIE [46]	IP traceback	Backward routes	×	✓	✓	✓(Routers)	✓(Packets)
NetSight [21]	Network debugging	Packet histories	×	✓	×	✓(Routers)	✓(Packets)
Netdiff [33]	ISP performance benchmarking	Delay	×	×	✓	× (ISPs)	✓(Packets)
Paris-traceroute [3]	Load-balancer detection	Load-balanced routes	×	×	✓	✓(Routers)	✓(Packets)
HAL [17]	Packet attestation	Packet transmissions	✓	✓	×	✓(Links)	✓(Packets)
AudIt [2]	Performance accountability	Loss, delay	×	×	✓	× (ISPs)	✓(Both)
SPP	Single network-level primitive	All of the above	✓	✓	✓	✓(Routers)	✓(Packets)

Table 12: Comparison between SPP and some existing diagnostic and forensic primitives.

value of the equivalence keys, and the value is a pointer to the shared provenance tree. We then associate each output tuple of the same equivalence class to this shared representative provenance tree, by looking up the equivalence keys' values in $hmap$.

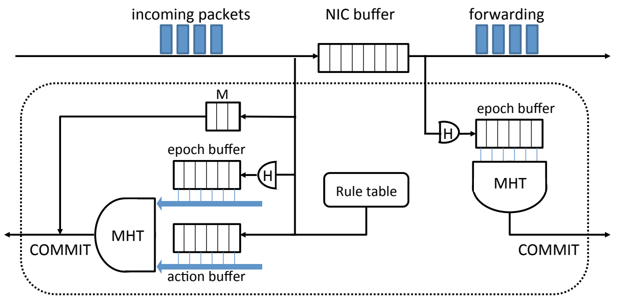
Our evaluation shows that the equivalence-based, online compression techniques achieve orders of magnitude reduction in storage and significant reduction in query latency, with only negligible network overhead added to each monitored network application at runtime.

3.2 Hardware-enabled Secure Provenance Collection

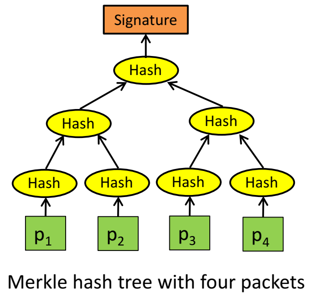
Another aspect of the challenge is to maintain provenance over high-speed traffic with security guarantees. Modern routers can process packets at Gigabits per second, so maintaining provenance, even without security guarantees, can result in non-trivial overhead; when we need to additionally achieve security guarantees, heavy-weight cryptographic operations are necessary, further amplifying the overhead. In [7], we designed Secure Packet Provenance (SPP), which addresses these challenges when maintaining per-packet provenance for high-speed Internet traffic. We used two key techniques to lower the overhead. First, we designed in SPP a lightweight protocol that can collect per-packet provenance data with tamper-proof evidence. The design avoids extensive use of heavy-weight cryptographic operations; instead, it uses Merkle Hash Trees (MHT), a data structure that allows efficient signatures over a large batch of packets. This greatly improves performance by reducing the processing logic mostly to hashing. Second, SPP offloaded the hashing and MHT construction to NetFPGAs to utilize its high parallelism, further improving the efficiency to maintain provenance over high-speed traffic.

Our key design of the SPP primitive is shown in Figure 4a. It consists of several modules: a) a *hash module*, which hashes each incoming packet at line speed, and stores the hash values at an *epoch buffer*; b) the *epoch buffer* temporarily holds packet hashes in the most recent batch, and sends the hashes to the *MHT constructor* for hash tree construction; c) the *MHT constructor* builds the MHT over the most recent epoch of packet hashes, and produces a top-level hash as the final commitment of the current batch; and d) the *loss detection* buffer (shown as M in the figure) identifies lost packets in the epoch, and feeds the information back to the sender so that the sender and receiver can "agree" on which packets have been successfully delivered. We have further sketched the MHT algorithm using an example of four packets in Figure 4b.

We have demonstrated that SPP is able to collect per-packet provenance data at line rate, and that it incurs low overheads in terms of storage and bandwidth. Moreover, with SPP, we have shown that many challenging tasks for Internet diagnostics and forensics can be approximated on top of this primitive with very few lines of code. Table 12 shows a comparison between SPP and other identified diagnostic and forensic tasks in the literature, and that network provenance can be a rather useful primitive that, once enabled, benefits many existing applications.



(a) The dataflow in SPP.



(b) An example Merkle Hash Tree.

Figure 4: The design of the SPP primitive.

4 Provenance for Diagnostics

Collecting and maintaining provenance is just the first step towards diagnosing data centers. Provenance often times is still hard for human users to understand. In fact, provenance trees can be quite large for system admins to reason about. For example, in an average-sized SDN, the provenance graph of why a specific packet p arrived at server s can easily contain hundreds of vertices [9, 52]. To address this challenge, we introduce two complementary approaches to distill from provenance compact and actionable suggestions for system admins.

4.1 Root Cause Detection with Differential Provenance

Our first approach explores the potential of repairing faults caused by misconfigurations through differential analysis across provenance trees [9]. The key observation is that misconfigurations, especially the subtle ones that need help of diagnostic tools, usually only affect a subset of traffic/nodes or only manifest themselves sporadically. Therefore, an operator typically has collected both working and non-working instances of similar traffic or service. Contrasting the provenance for the working and non-working instances is likely to provide more insights than studying only the non-working instance: it very much resembles the human debugging procedure—if we can understand why the two provenance are different and how to align them, we can likely identify the root cause of the problem. We call this approach differential provenance [9].

One challenge in differential provenance is that a small, initial difference can have a significant magnifying effect, for example, packets may take two completely different path and arrive at different destinations due to a small difference in the routing table of a gateway router. Therefore, a superficial "tree-diff" approach is likely to generate unusable results; our case studies have confirmed that the differences between working and non-working provenance trees can be even larger than the original provenance trees. To address this challenge, differential provenance identifies the first diverging point in the working and non-working provenance. We "roll back" the network execution to that point, change the mismatched tuple(s) on the non-working provenance tree to the correct version, and then "roll forward" the execution. We repeat this process until the two provenance are completely aligned.

Differential provenance has been demonstrated to be quite effective in a number of case studies on repairing software-defined networks and Hadoop MapReduce jobs. Our results show that it can always pinpoint exactly the induced misconfigurations to be the root cause of observed network problems.

4.2 Automated Program Patch with Meta Provenance

Differential provenance by itself is still fundamentally unable to help when a fault is caused by a bug in the program code: it can answer questions about the *data* in the network – such as "show me the configuration entries that have caused this packet to be dropped" – but it treats the *code* as given and immutable, so it cannot answer questions such as "show me the line in the controller program that has caused this packet to be dropped".

To address this limitation, we have generalized traditional provenance to *meta provenance* [49,50]. Our idea is, essentially, to treat the code as another kind of data: we represent the syntactic elements of the program as a special class of tuples, which we call *meta tuples*, and we capture the semantics of the programming language (e.g., a language for SDN controllers, such as Pyretic [36]) with a set of special rules, which we call *meta rules*. For instance, an NDlog rule could be represented as meta tuples for the head and for each precondition, and one of the meta rules for NDlog could say that the tuple at the head of a rule is derived whenever all of the preconditions hold at the same time.

Once we have a meta model for a given language, we use a form of negative provenance [51,52] at the meta level to ask why some conditions did not hold. For instance, if the operator observes that a certain packet has been dropped and failed to reach a certain server, she can generate the negative provenance of the packet's arrival at that server (which is essentially asking "Why did this packet not reach that particular server?"). Negative provenance will then generate a recursive explanation, whose leaves include not just changes to data, but also changes to code; in this process, we use solvers such as Z3 [12] to find concrete values for the changes and thus generate repairs. Unlike positive provenance trees, negative provenance trees are typically extremely large, or even infinite—there are almost always many different ways to "fix" a given problem, ranging from small tweaks to rewriting the entire program. With the expectation that it is infeasible to fix programs that require significant or even complete rewrite, we prioritize modifications that minimize changes to the original program.

Once potential modifications are identified, we use "backtesting" to further examine each of them and evaluate whether it is actually a "correct" modification. That is, given the same system configurations and external inputs, the modified program should rectify faulty execution traces, and avoid altering the progress of execution traces that are previously correct. We leverage multi-query optimization [16,31] from the database literature to speed up the backtesting, which enables us to validate multiple repairs in a single run.

We have applied meta provenance to subsets of three different SDN controller languages: NDlog [30], Pyretic [36], and Trema [48]. Results from several case studies show that our system can generate high-quality repairs for realistic bugs, typically in less than one minute.

5 Related Work

Provenance. Provenance is a concept initiated from the database community [4], but it has recently been applied in several other areas, including network systems [52, 58–61], storage systems [37], cloud computing platforms [23,40], and natural language processing [13]. Our work is mainly related to projects that use network provenance for diagnostics in distributed systems. In this area, ExSPAN [61] was the first system to maintain network provenance at scale; SNP [59] added integrity guarantees in adversarial settings, DTaP [60] a temporal dimension, and Y! [52] support for negative events. These systems generate an explanation on why some data exist or do not exist, but provide limited support for further diagnosis such as providing suggestions for system repair. There have been proposals for automating system repair based on provenance. Huang et al. [22] and Meliou et al. [34,35] focus on instance-based explanations for missing answers, that is, how to obtain the missing answers by making modifications to the value of base instances (tuples); Why-Not [5] and ConQueR [47] provide query-based explanations for SQL queries, which reveal over-constrained conditions in the queries and suggest modifications to them.

System debugging and repair. Network debugging can be achieved by static analysis, e.g., as in Batfish [15], Header Space Analysis [26], NetPlumber [25], VeriFlow [27], Libra [55] rcc [14], or dynamic testing, e.g., as in Minimal Causal Sequence analysis [43], DEMi [42], OFRewind [53], and ATPG [54]. Some domain-specific languages, e.g., NetKAT [1], Flowlog [38], Kinect [28], can enable verification of specific classes of SDN programs. The software engineering community has used genetic programming [29] and symbolic execution [39], for system repair; they are designed to fix small programs, or to propose specific kinds of fixes. ClearView [41] mines invariants in programs, correlates violations with failures, and generates fixes at runtime; ConfDiagnoser [56] compares correct and undesired executions to find suspicious predicates in the program; and Sidiroglou et al. [44] runs attack vectors on instrumented applications and then generates fixes automatically. These systems primarily rely on heuristics, whereas provenance-based approaches can track causality and can thus pinpoint specific root causes.

6 Future Directions

Timing causality. Faulty temporal behaviors can happen in distribute systems. For instance, suppose that a virtual machine takes unusually long to boot, because a misconfigured machine is overloading the shared storage backend. Existing tools, such as DTaP [60] or Dapper [45], offer little help with this scenario – in fact, the actual root cause (the misconfigured machine) would not even appear in the explanation! Because existing tools only capture *functional* causality, that is, events which directly contributed to the occurrence of the observed symptom, and will miss root causes that contributed only in terms of timing. We are currently studying ways to generalize provenance to track not only functional causality but also *temporal* causality, which is any event that has contributed to the timing of the observed symptom, regardless of whether it is functionally-related. This should allow provenance to explain faulty temporal behaviors.

Causality networks. Existing work on provenance only explains why a *single* event did or did not come about. While this is a useful starting point for diagnosis, it does not capture a range of other types of causality between events. For instance, operators may be interested to know why two events e_1 and e_2 always happen at the same time, or why e_1 and e_1 never happen at the same time, or even why only one of these events happens. To capture such complex, *inter-event* causality, we would need to extend the existing provenance abstraction to allow for reasoning about multiple events at the same time. Currently, we are looking at the possibility to extend provenance to causal networks [18], which encodes the inter-dependencies among events, and then using causal networks to answer queries of the more complex form. This advanced form of provenance may be helpful to process diagnostic queries that cannot be easily answered with the current provenance tree abstraction.

Probabilistic causality. Another intriguing direction is to develop models and efficient evaluation for probabilistic provenance. Unlike traditional provenance models, the inputs and derivations themselves are set to true or false based on a probabilistic distribution. This is motivated by multiple scenarios. First, the use of machine learning and model-based prediction means that a significant portion of causality relationships are inherently probabilistic in nature. Second, in some environments, the derived provenance themselves can be subjected to probabilistic distributions, for example, on a lossy communication channel, the likelihood of links being up or packets being transmitted correctly may themselves be probabilistic. Finally, in the case of applications written in a non-declarative language, the input-output dependencies themselves may be inferred probabilistically. A key challenge is to maintain provenance for *all* possible combinations of variable assignment. Inspired by prior work on probabilistic databases, a potential solution is to develop a provenance model based on the *possible-worlds semantics* [11]. An associated challenge is potential of state explosion when evaluation probabilistic provenance. One possible approach to address this challenge is to explore tradeoffs between accuracy and performance (e.g., query latency, and communication overhead).

References

[1] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *Proc. POPL*, 2014.

[2] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker. Loss and delay accountability for the Internet. In *Proc. ICNP*, 2007.

[3] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In *Proc. IMC*, 2006.

[4] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc. ICDT*, pages 316–330, 2001.

[5] A. Chapman and H. V. Jagadish. Why not. In *Proc. SIGMOD*, 2009.

[6] A. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proceedings of ACM SIGMOD*, pages 993–1006, 2008.

[7] A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. One primitive to diagnose them all: Architectural support for internet diagnostics. In *Proc. Eurosys*, 2017.

[8] A. Chen, Y. Wu, A. Haeberlen, B. T. Loo, , and W. Zhou. Data provenance at internet scale: Architecture, experiences, and the road ahead. In *Proc. of CIDR*, 2017.

[9] A. Chen, Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. The Good, the Bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of ACM SIGCOMM 2016*, Aug. 2016.

[10] C. Chen, H. T. Lehri, L. K. Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou. Distributed provenance compression. In *Proc. SIGMOD*, 2017.

[11] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: Diamonds in the dirt. *Commun. ACM*, 52(7):86–94, July 2009.

[12] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, 2008.

[13] D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. In *Proc. of VLDB Endowment*, 2017.

[14] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proc. NSDI*, Boston, MA, May 2005.

[15] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proc. NSDI*, 2015.

[16] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing one thousand queries with one stone. 5(6):526–537, 2012.

[17] A. Haeberlen, P. Fonseca, R. Rodrigues, and P. Druschel. Fighting cybercrime with packet attestation. Technical Report MPI-SWS-2011-002, Max Planck Institute for Software Systems, July 2011.

[18] J. Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British journal for the philosophy of science*, 56(4):843–887, 2005.

[19] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown. Where is the debugger for my software-defined network? In *Proc. HotSDB*, 2012.

[20] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. NSDI*, 2014.

[21] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proc. NSDI*, 2014.

[22] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *Proc. VLDB Endow.*, 1(1), Aug. 2008.

[23] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. In *Proc. of VLDB Endowment*, 2015.

[24] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *Proceedings of ACM SIGMOD*, pages 951–962, 2010.

[25] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. NSDI*, 2013.

[26] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. NSDI*, 2012.

[27] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *Proc. NSDI*, 2013.

[28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *Proc. NSDI*, 2015.

[29] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. ICSE*, 2012.

[30] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Commun. ACM*, 52(11):87–95, Nov. 2009.

[31] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. SIGMOD*, 2002.

[32] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *Proc. SOSP*, 2003.

[33] R. Mahajan, M. Zhang, L. Poole, and V. Pai. Uncovering performance differences among backbone ISPs with Netdiff. In *Proc. NSDI*, 2008.

[34] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *Proc. SIGMOD*, 2011.

[35] A. Meliou, W. Gatterbauer, and D. Suciu. Bringing Provenance to its Full Potential using Causal Reasoning. In *Proc. TaPP*, 2011.

[36] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proc. NSDI*, 2013.

[37] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX ATC*, 2006.

[38] T. Nelson, A. D. Ferguson, M. J. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proc. NSDI*, 2014.

[39] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. ICSE*, May 2013.

[40] H. Park, R. Ikeda, and J. Widom. RAMP: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12), 2011.

[41] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, et al. Automatically patching errors in deployed software. In *Proc. SOSP*, 2009.

[42] C. Scott, A. Panda, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *Proc. NSDI*, 2016.

[43] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting blackbox SDN control software with minimal causal sequences. In *Proc. SIGCOMM*, 2014.

[44] S. Sidiroglou and A. D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 2005.

[45] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. In *Google Technical Report*, 2010.

[46] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, B. Schwartz, S. Kent, and W. Strayer. Single-packet IP traceback. *IEEE/ACM Trans. Netw.*, 10(6):721–734, 2002.

[47] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *Proc. SIGMOD*, 2010.

[48] Trema: Full-Stack OpenFlow Framework in Ruby and C. https://trema.github.io/trema/.

[49] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated network repair with meta provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets'15)*, Nov. 2015.

[50] Y. Wu, A. Chen, A. Haeberlen, W. Zhou, and B. T. Loo. Automated bug removal for software-defined networks. In *Proc. NSDI*, 2017.

[51] Y. Wu, A. Haeberlen, W. Zhou, and B. T. Loo. Answering why-not queries in software-defined networks with negative provenance. In *Proceedings of the 12th ACM Workshop on Hot Topics in Networks (HotNets-XII)*, Nov. 2013.

[52] Y. Wu, M. Zhao, A. Haeberlen, W. Zhou, and B. T. Loo. Diagnosing missing events in distributed systems negative provenance. In *Proceedings of ACM SIGCOMM 2014*, Aug. 2014.

[53] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling record and replay troubleshooting for networks. In *Proc. USENIX ATC*, 2011.

[54] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *Proc. CoNEXT*, 2012.

[55] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proc. NSDI*, 2014.

[56] S. Zhang and M. D. Ernst. Automated diagnosis of software configuration errors. In *Proc. ICSE*, 2013.

[57] Y. Zhang, Z. M. Mao, and M. Zhang. Detecting traffic differentiation in backbone ISPs with NetPolice. In *Proc. IMC*, 2009.

[58] Y. Zhang, A. O?Neil, M. Sherr, and W. Zhou. Private network provenance. In *Proc. of VLDB Endowment*, 2017.

[59] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure Network Provenance. In *Proc. SOSP*, 2011.

[60] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB'13)*, Aug. 2013.

[61] W. Zhou, M. Sherr, T. Tao, X. Li, B. T. Loo, and Y. Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proc. SIGMOD*, 2010.

34th IEEE International Conference on Data Engineering 2018

ICDE PARIS 2018

April 16-20, 2018, Paris, France

http://icde2018.org/

http://twitter.com/icdeconf #icde18

Call for Participation

The annual IEEE International Conference on Data Engineering (ICDE) addresses research issues in designing, building, managing, and evaluating advanced data-intensive systems and applications. It is a leading forum for researchers, practitioners, developers, and users to explore cutting-edge ideas and to exchange techniques, tools, and experiences.

Conference Sessions:

- Keynotes
- Research Papers
- Industrial Papers
- Demos
- Panels
- Tutorials
- Lightning Talks (new track)
- Posters (ICDE & TKDE)

General Chairs:

Malu Castellanos (Teradata, USA)
Ioana Manolescu (Inria, France)

Affiliated Workshops:

- **Context in Analytics: Challenges, Opportunities and Trends**
- **Data Engineering meets the Semantic Web (DESWeb2018)**
- **The Joint Workshop of HardBD (International Workshop on Big Data Management on Emerging Hardware) and Active (Workshop on Data Management on Virtualized Active Systems)**
- **Data Engineering meets Intelligent Food and COoking Recipe 2018 (DECOR 2018)**
- **Emerging Data Engineering Methods and Approaches for Precision Medicine (DEPM 2018)**
- **Accelerating In-Memory Databases (AIMD 2018)**

IEEE computer society   IEEE Advancing Technology for Humanity

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name ____________________

IEEE Member # ____________________

Mailing Address ____________________

____

Country ____________________

Email ____________________

Phone ____________________

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
Renmin University of China
Beijing 100872, China
duyong@ruc.edu.cn

TCDE Chair

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903