Language-integrated Provenance

Stefan Fehrenbach and James Cheney
University of Edinburgh
stefan.fehrenbach@ed.ac.uk,jcheney@inf.ed.ac.uk

ABSTRACT

Provenance, or information about the origin or derivation of data, is important for assessing the trustworthiness of data and identifying and correcting mistakes. Most prior implementations of data provenance have involved heavy-weight modifications to database systems and little attention has been paid to how the provenance data can be used outside such a system. We present extensions to the Links programming language that build on its support for language-integrated query to support provenance queries by rewriting and normalizing monadic comprehensions and extending the type system to distinguish provenance metadata from normal data. The main contribution of this paper is to show that the two most common forms of provenance can be implemented efficiently and used safely as a programming language feature with no changes to the database system.

1. INTRODUCTION

A Web application typically spans at least three different computational models: the server-side program, browser-side HTML or JavaScript, and SQL to execute on the database. Coordinating these layers is a considerable challenge. Recently, programming languages such as Links (Cooper et al. 2007), Hop (Serrano 2009) and Ur/Web (Chlipala 2015) have pioneered a *cross-tier* approach to Web programming. The programmer writes a single program, which can be type-checked and analyzed in its own right, but parts of it are executed to run efficiently on the multi-tier Web architecture by translation to HTML, JavaScript and SQL. Cross-tier Web programming builds on *language-integrated query* (Meijer et al. 2006), a technique for safely embedding database queries into programming languages.

When something goes wrong in a database-backed Web application, understanding what has gone wrong and how to fix it is also a challenge. Often, the database is the primary "state" of the program, and problems arise when this state becomes inconsistent or contains erroneous data. For example, Figure 1 shows Links code for querying data

```
var agencies = table "Agencies"
 with (name:String, based_in:String, phone:String)
 from db;
var externalTours = table "ExternalTours"
 with (name:String, destination:String, type:String, price:Int)
 from db;
var q1 = query {
  for (a <-- agencies)
    for (e <-- externalTours)
    where (a.name == e.name && e.type == "boat")
      [(name = e.name,
        phone = a.phone)]
}
```

Figure 1: Links table declarations and example query

name	phone
EdinTours	412 1200
EdinTours	412 1200
Burns's	607 3000

Figure 2: Example query results

from a (fictional) Scottish tourism database, with the result shown in Figure 2. Suppose one of the phone numbers is incorrect: we might want to know *where* in the source database to find the source of this incorrect data, so that we can correct it. Alternatively, suppose we are curious *why* some data is produced: for example, the result shows EdinTours twice. If we were not expecting these results, e.g. because we believe that EdinTours is a bus tour agency and does not offer boat tours, then we need to see additional input data to understand why they were produced.

Automatic techniques for producing such explanations, often called *provenance*, have been explored extensively in the database literature (Cui et al. 2000; Buneman et al. 2001; Green et al. 2007; Glavic and Alonso 2009b). Neither conventional nor cross-tier Web programming currently provides direct support for provenance. A number of implementation strategies for efficiently computing provenance for query results have been explored, but no prior work considers the interaction of provenance with clients of the database.

We propose *language-integrated provenance*, a new approach to implementing provenance that leverages the benefits of language-integrated query. In this paper, we present two instances of this approach, one which computes *where-provenance* showing where in the underlying database a result was copied from, and another which computes *lineage*

PPDP '16, September 05 - 07, 2016, Edinburgh, United Kingdom

ISBN 978-1-4503-4148-6/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2967973.2968604

showing all of the parts of the underlying database that were needed to compute part of the result. Both techniques are implemented by a straightforward source-to-source translation which adjusts the types of query expressions to incorporate provenance information and changes the query behavior to generate and propagate this information. Our approach is implemented in Links, and benefits from its strong support for rewriting queries to efficient SQL equivalents, but the underlying ideas may be applicable to other languages that support language-integrated query, such as F# (Syme 2006), SML# (Ohori and Ueno 2011), or Ur/Web (Chlipala 2015).

Most prior implementations of provenance involve changes to relational database systems and extensions to the SQL query language, departing from the SQL standard that relational databases implement. To date, none of these proposals have been incorporated into the SQL standard or supported by mainstream database systems. If such extensions are adopted in the future, however, we can simply generate queries that use these extensions in Links. In some of these systems, enabling provenance in a query changes the result type of the query (adding an unpredictable number of columns). Our approach is the first (to the best of our knowledge) to provide type-system support that makes sure that the extra information provided by language-integrated provenance queries is used safely by client.

Our approach builds on Links's support for queries that construct nested collections (Cheney et al. 2014c). This capability is crucial for lineage, because the lineage of an output record is a *set* of relevant input records. Moreover, our provenance translations can be used with queries that construct nested results. Our approach is also distinctive in allowing fine-grained control over where-provenance. In particular, the programmer can decide whether to enable or disable where-provenance tracking for individual input table fields, and whether to keep or discard provenance for each result field.

We present two simple extensions to Links to support where-provenance and lineage, and give (provably type-preserving) translations from both extensions to plain Links. We have implemented both approaches and experimentally validated them using a synthetic benchmark. Provenance typically slows down query evaluation because more data is manipulated. For where-provenance, our experiments indicate a constant factor overhead of 1.5–2.8. For lineage, the slowdown is between 1.25 and 7.55, in part because evaluating lineage queries usually requires manipulating more data. Although we have not yet compared our approach directly to other systems, these results appear to be in a reasonable range: for example, the Perm system (Glavic and Alonso 2009b) reports slowdowns of 3–30 for a comparable form of lineage.

This paper significantly extends an earlier workshop paper (Fehrenbach and Cheney 2015). The workshop version only outlined our initial design for where-provenance in Links; this paper presents the fully-implemented system, extends it to support lineage, and gives a detailed experimental evaluation of both extensions.

2. OVERVIEW

In this section we give an overview of our approach, first covering necessary background on Links and language-integrated query based on comprehensions, and then showing how provenance can be supported by query rewriting in this framework.

Base types	O	$::=$	**Int** | **Bool** | **String**
Rows	R	$::=$	$\cdot \mid R, l : A$
Table types	T	$::=$	**table**(R)
Types	A, B	$::=$	$O \mid T \mid A$ -> $B \mid (R) \mid [A]$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, x : A$
Expressions	L, M, N	$::=$	$c \mid x \mid (l_i = M_i) \mid N.l$
		|	**fun** $f(x_i)\ N \mid N(M_i)$
		|	**var** $x = M; N$ | **if** $(L)\ \{M\}$ **else** $\{N\}$
		|	**query** $\{N\}$ | **table** n **with** $(l_i : O_i)$
		|	[] | $[N]$ | N ++ M | **empty**(M)
		|	**for** $(x$ <- $L)\ M$ | **where**$(M)\ N$
		|	**for** $(x$ <-- $L)\ M$ | **insert** L **values** M
		|	**update** $(x$ <- $L)$ **where** M **set** N
		|	**delete** $(x$ <- $L)$ **where** M

Figure 3: Syntax of a subset of Links.

We will use a running example of a simple tours database, with some example data shown in Figure 5.

2.1 Links background

We first review a subset of the Links programming language that includes all of the features relevant to our work; we omit some features (such as effect typing, polymorphism, and concurrency) that are not required for the rest of the paper. We also omit detailed discussion of the operational semantics of Links, which is presented in previous work (Lindley and Cheney 2012).

Figure 3 presents a simplified subset of Links syntax, sufficient for explaining the provenance translations in this paper. Types include base types O (such as integers, booleans and strings), table types **table**$(l_i\colon A_i)$, function types A -> B, record types $(l_i\colon A_i)$, and collection types $[A]$. In Links, collection types are treated as multisets inside database queries (reflecting SQL's default multiset semantics), but represented as lists during ordinary execution.

Expressions include standard constructs such as constants, variables, record construction and field projection, conditionals, functions and application. We freely use pair types (A, B) and pair syntax (M, N) and projections $M.1$, $M.2$ etc., which are easily definable using records. Constants c can be functions such as integer addition, equality tests, etc.; their types are collected in a signature Σ. In Links we write **var** $x = M; N$ for binding a variable x to M in a N. The semantics of the Links constructs discussed so far is call-by-value. The expression **query** $\{M\}$ introduces a query block, whose content is not evaluated in the usual call-by-value fashion but instead first *normalized* to a form equivalent to an SQL query, and then submitted to the database server. The resulting table (or tables, in the case of a nested query result) are then translated into a Links value. Queries can be constructed using the expressions for the empty collection [], singleton collection $[M]$, and concatenation of collections M ++ N. In addition, the comprehension expressions **for**(x <-- M) N and **for**(x <- M) L allow us to form queries involving iteration over a collection. The difference between the two expressions is that **for**(x <-- M) expects M to be a table reference, whereas **for**(x <- M) expects M to be a collection. The expression **where** $(M)\ N$ is essentially equivalent to **if** $(M)\ \{N\}$ **else** $\{[]\}$,

$$\frac{\Sigma(c) = A}{\Gamma \vdash c : A}\ \textsc{Const} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var} \qquad \frac{\Gamma \vdash M_i : A_i}{\Gamma \vdash (l_i \;=\; M_i)_{i=1}^n : (l_i : A_i)}\ \textsc{Record} \qquad \frac{\Gamma \vdash M : (l_i : A_i)_{i=1}^n}{\Gamma \vdash M.l_k : A_k}\ \textsc{Projection}$$

$$\frac{\Gamma, [x_i : A_i]_{i=1}^n \vdash M : B}{\Gamma \vdash \textsf{fun}\ (x_i|_{i=1}^n)\{M\} : (A_i|_{i=1}^n)\ \texttt{->}\ B}\ \textsc{Fun}$$

$$\frac{\Gamma \vdash M : (A_i|_{i=1}^n)\ \texttt{->}\ B \qquad \Gamma \vdash N_i : A_i \quad (i \in \{1,\ldots,n\})}{\Gamma \vdash M(N_i|_{i=1}^n) : B}\ \textsc{App} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \textsf{var}\ x \;=\; M; N : B}\ \textsc{Var} \qquad \frac{\Gamma \vdash M : [A] \qquad A :: \textsf{QType}}{\Gamma \vdash \textsf{query}\ \{M\} : [A]}\ \textsc{Query}$$

$$\frac{\Gamma \vdash M : [A]}{\Gamma \vdash \textsf{empty}(M) : \textsf{Bool}}\ \textsc{Empty} \qquad \frac{R :: \textsf{BaseRow}}{\Gamma \vdash \textsf{table}\ n\ \textsf{with}\ (R) : \textsf{table}(R)}\ \textsc{Table} \qquad \frac{}{\Gamma \vdash [] : [A]}\ \textsc{Empty-List} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash [M] : [A]}\ \textsc{List} \qquad \frac{\Gamma \vdash M : [A] \qquad \Gamma \vdash N : [A]}{\Gamma \vdash M\ \texttt{++}\ N : [A]}\ \textsc{Concat}$$

$$\frac{\Gamma \vdash L : [A] \qquad \Gamma, x : A \vdash M : [B]}{\Gamma \vdash \textsf{for}\ (x\ \texttt{<-}\ L)\ M : [B]}\ \textsc{For-List} \qquad \frac{\Gamma \vdash M : \textsf{Bool} \qquad \Gamma \vdash N : [B]}{\Gamma \vdash \textsf{where}\ (M)\ N : [B]}\ \textsc{Where} \qquad \frac{\Gamma \vdash L : \textsf{table}(R) \qquad \Gamma, x : (R) \vdash M : [B]}{\Gamma \vdash \textsf{for}\ (x\ \texttt{<-}\ L)\ M : [B]}\ \textsc{For-Table}$$

$$\frac{\Gamma \vdash L : \textsf{table}(R) \qquad \Gamma \vdash M : [(R)]}{\Gamma \vdash \textsf{insert}\ L\ \textsf{values}\ M : ()}\ \textsc{Insert} \qquad \frac{\Gamma \vdash L : \textsf{table}(R) \qquad \Gamma, x : (R) \vdash M : \textsf{Bool} \qquad \Gamma, x : (R) \vdash N : [(R)]}{\Gamma \vdash \textsf{update}\ L\ \textsf{where}\ M\ \textsf{set}\ N : ()}\ \textsc{Update}$$

$$\frac{\Gamma \vdash L : \textsf{table}(R) \qquad \Gamma, x : (R) \vdash M : \textsf{Bool}}{\Gamma \vdash \textsf{delete}\ L\ \textsf{where}\ M : ()}\ \textsc{Delete}$$

Figure 4: Typing rules for Links.

and is intended for use in filtering query results. The expression **empty** (M) tests whether the collection produced by M is empty. These comprehension syntax constructs can also be used outside a query block, but they are not guaranteed to be translated to queries in that case. The **insert**, **delete** and **update** expressions perform updates on database tables; they are implemented by direct translation to the analogous SQL update operations.

The type system (again a simplification of the full system) is illustrated in Figure 4. Many rules are standard; we assume a typing signature Σ mapping constants and primitive operations to their types. The rule for **query** $\{M\}$ refers to an auxiliary judgment $A :: \textsf{QType}$ that essentially checks that A is a valid query result type, meaning that it is constructed using base types and collection or record type constructors only:

$$\frac{}{O :: \textsf{QType}} \qquad \frac{[A_i :: \textsf{QType}]_{i=1}^n}{(l_i : A_i)_{i=1}^n :: \textsf{QType}} \qquad \frac{A :: \textsf{QType}}{[A] :: \textsf{QType}}$$

Similarly, the $R :: \textsf{BaseRow}$ judgment ensures that the types used in a row are all base types:

$$\frac{}{\cdot :: \textsf{BaseRow}} \qquad \frac{R :: \textsf{BaseRow}}{R, l : O :: \textsf{BaseRow}}$$

The full Links type system also checks that the body M uses only features available on the database (and only calls functions that satisfy the same restriction). The rules for other query operations are straightforward, and similar to those for monadic comprehensions in other systems. Finally, the rules for updates (insert, update, and delete) are also mildly simplified; in the full system, the conditions and update expressions are required to be database-executable operations. Lindley and Cheney (2012) presents a more complete formalization of Links's type system that soundly characterizes the intended run-time behavior.

The core language of Links we are using is a simplification of the full language in several respects. Links includes a number of features (e.g. recursive datatypes, XML literals, client/server annotations, and concurrency features) that are important parts of its Web programming capabilities but not needed to explain our contribution. Links also uses a type-and-effect system to determine whether the code inside a **query** block is translatable to SQL, and which functions can be called safely from query blocks. We use a simplified version of Links's type system that leaves out these effects and does not deal with polymorphism. Our implementation does handle these features, with some limitations discussed later.

2.2 Language-integrated query

Writing programs that interact with databases can be tricky, because of mismatches between the models of computation and data structures used in databases and those used in conventional programming languages. The default solution (employed by JDBC and other typical database interface libraries) is for the programmer to write queries or other database commands as uninterpreted strings in the host language, and these are sent to the database to be executed. This means that the types and names of fields in the query cannot be checked at compile time and any errors will only be discovered as a result of a run-time crash or exception. More insidiously, failure to adequately sanitize user-provided parameters in queries opens the door to SQL injection attacks (Shar and Tan 2013).

Language-integrated query is a technique for embedding queries into the host programming language so that their types can be checked statically and parameters are automatically sanitized. Microsoft's LINQ library, which provides language-integrated query for .NET languages, is a popular feature of C# and F#. Broadly, there are two common approaches to language-integrated query. The first approach, which we call *SQL embedding*, adds specialized constructs resembling SQL queries to the host language, so that they can be typechecked and handled correctly by the program. This is

Agencies

(oid)	name	based_in	phone
1	EdinTours	Edinburgh	412 1200
2	Burns's	Glasgow	607 3000

ExternalTours

(oid)	name	destination	type	price in £
3	EdinTours	Edinburgh	bus	20
4	EdinTours	Loch Ness	bus	50
5	EdinTours	Loch Ness	boat	200
6	EdinTours	Firth of Forth	boat	50
7	Burns's	Islay	boat	100
8	Burns's	Mallaig	train	40

Figure 5: Example input data

the approach taken in C# (Meijer et al. 2006), SML# (Ohori and Ueno 2011), and Ur/Web (Chlipala 2015). The second approach, which we call *comprehension*, uses monadic comprehensions or related constructs of the host language, and generates queries from such expressions. The comprehension approach builds on foundations for querying databases using comprehensions developed by Buneman et al. (1995), and has been adopted in languages such as F# (Syme 2006) and Links (Cooper et al. 2007) as well as libraries such as Database-Supported Haskell (Giorgidze et al. 2011).

The advantage of the comprehension approach is that it provides a higher level of abstraction for programmers to write queries, without sacrificing performance. This advantage is critical to our work, so we will explain it in some detail. For example, the query shown in Figure 1 illustrates Links comprehension syntax. It asks for the names and phone numbers of all agencies having an external tour of type *"boat"*. The keyword **for** performs a comprehension over a table (or other collection), and the **where** keyword imposes a Boolean condition filtering the results. The result of each iteration of the comprehension is a singleton collection containing the record (name = e.name,phone = a.phone).

Monadic comprehensions do not always correspond exactly to SQL queries, but under certain reasonable assumptions, it is possible to normalize these comprehension expressions to a form that is easily translatable to SQL. For example, the following query

```
var q1' = query {
  for (e <-- externalTours)
  where (e.type == "boat")
    for (a <-- agencies)
    where (a.name == e.name)
    [(name = e.name, phone = a.phone)]
}
```

does not directly correspond to a SQL query due to the alternation of **for** and **where** operations; nevertheless, query normalization generates a single equivalent SQL query in which the **where** conditions are both pushed into the SQL query's WHERE clause:

```
SELECT e.name AS name, a.phone AS phone
  FROM ExternalTours e, Agencies a
 WHERE e.type = 'boat' AND a.name = e.name
```

Normalization frees the programmer to write queries in more natural ways, rather than having to fit the query into a pre-defined template expected by SQL.

However, this freedom can also lead to problems, for example if the programmer writes a query-like expression that contains an operation, such as print or regular expression matching, that cannot be performed on the database. In early versions of Links, this could lead to unpredictable performance, because queries would unexpectedly be executed on the server instead of inside the database. The current version uses a type-and-effect system (as described by Cooper (2009) and Lindley and Cheney (2012)) to track which parts of the program must be executed in the host language and which parts may be executed on the database. Using the **query** keyword above forces the typechecker to check that the code inside the braces will successfully execute on the database.

2.3 Higher-order functions and nested query results

Although comprehension-based language-integrated query may seem (at first glance) to be little more than a notational convenience, it has since been extended to provide even greater flexibility to programmers without sacrificing performance.

The original results on normalization (due to Wong (1996)) handle queries over flat input tables and producing flat result tables, and did not allow calling user-defined functions inside queries. Subsequent work has shown how to support higher-order functions (Cooper 2009; Grust and Ulrich 2013) and queries that construct nested collections (Cheney et al. 2014c). For example, we can use functions to factor the previous query into reusable components, provided the functions are nonrecursive and only perform operations that are allowed in the database.

```
fun matchingAgencies(name) {
  for (a <-- agencies)
  where (a.name == name)
    [(name = e.name, phone = a.phone)]
}
var q1'' = query {
  for (e <-- externalTours)
  where (e.type == "boat")
    matchingAgencies(e.name)
}
```

Cooper's results show that these queries still normalize to SQL-equivalent queries, and this algorithm is implemented in Links. Similarly, we can write queries whose result type is an arbitrary combination of record and collection types, not just a flat collection of records of base types as supported by SQL:

```
var q2 = query {
  for (a <-- agencies)
    [(name = a.name,
      tours = for (e <-- externalTours)
              where (e.name == a.name)
              [(dest = e.destination, type = e.type)]
}
```

This query produces records whose second tours component is itself a collection — that is, the query result is of the type [(name:String,[(dest:String, type:Type)])] which contains a nested occurrence of the collection type constructor []. SQL does not directly support queries producing such nested results — it requires flat inputs and query results.

Our previous work on *query shredding* (Cheney et al. 2014c) gives an algorithm that evaluates queries with nested results efficiently by translation to SQL. Given a query whose return type contains n occurrences of the collection type constructor, query shredding generates n SQL queries that can be evaluated on the database, and constructs the nested result from the resulting tables. This is typically much more efficient than loading the database data into memory and evaluating the query there. Links supports query shredding and we will use it in this paper to implement lineage.

Both capabilities, higher-order functions and nested query results, are essential building blocks for our approach to provenance. In what follows, we will use these techniques without further explanation of their implementation. The details are covered in previous papers (Cooper 2009; Lindley and Cheney 2012; Cheney et al. 2014c), but are not needed to understand our approach.

2.4 Where-provenance and lineage

As explained in the introduction, provenance tracking has been explored extensively for queries in the database community. We are now in a position to explain how these provenance techniques can be implemented on top of language-integrated query in Links. We review two of the most common forms of provenance, and illustrate our approach using examples; the rest of the paper will use similar examples to illustrate our implementation approach.

Where-provenance is information about where information in the query result "came from" (or was copied from) in the input. Buneman et al. (2001) introduced this idea; our approach is based on a later presentation for the nested relational calculus by Buneman et al. (2008). A common reason for asking for where-provenance is to identify the source of incorrect (or surprising) data in a query result. For example, if a phone number in the result of the example query is incorrect, we might ask for its where-provenance. In our system, this involves modifying the input table declaration and query as follows:

```
var agencies = table "Agencies"
 with (name:String, based_in:String, phone:String)
 where phone prov default
```

The annotation `phone prov default` says to assign phone numbers the "default" provenance annotation of the form (Agencies, phone, i) where i is the object id (oid) of the corresponding row. The field value will be of type **Prov**(**String**); the data value can be accessed using the keyword **data** and the provenance can be accessed using the keyword **prov**, as follows:

```
var q1''' = query {
  for (a <-- agencies)
    for (e <-- externalTours)
    where (a.name == e.name && e.type == "boat")
      [(name = e.name,
        phone = data a.phone, p_phone = prov a.phone)]
}
```

The result of this query is as follows:

name	phone	p_phone
EdinTours	412 1200	(Agencies,phone,1)
EdinTours	412 1200	(Agencies,phone,1)
Burns's	607 3000	(Agencies,phone,2)

Why-provenance is information that explains "why" a result was produced. In a database query setting, this is usually taken to mean a *justification* or *witness* to the query result, that is, a subset of the input records that includes all of the data needed to generate the result record. Actually, several related forms of why-provenance have been studied (Cui et al. 2000; Buneman et al. 2001; Cheney et al. 2009; Glavic et al. 2013), however, many of these only make sense for set-valued collections, whereas Links currently supports multiset semantics. In this paper, we focus on a simple form of why-provenance called *lineage* which is applicable to either semantics.

Intuitively, the lineage of a record r in the result of a query is a subset L of the records in the underlying database db that "justifies" or "witnesses" the fact that r is in the result of Q on db. That is, running Q on the lineage L should produce a result containing r, i.e. $r \in Q(L)$. Obviously, this property can be satisfied by many subsets of the input database, including the whole database db, and this is part of the reason why there exist several different definitions of why-provenance (for example, to require minimality). A common approach is to define the lineage to be the set of all input database records accessed in the process of producing r; this is a safe overapproximation to the minimal lineage, and usually is much smaller than the whole database.

We identify records in input database tables using pairs such as (AgencyTours,2) where the first component is the table name and the second is the row id, and the lineage of an element of a collection is just a collection of such pairs. (Again, this has the benefit that we can use a single type for references to data in multiple input tables.) Using this representation, the lineage for `q1` is as follows:

name	phone	lineage
EdinTours	412 1200	[(Agencies,1),(ExternalTours,5)]
EdinTours	412 1200	[(Agencies,1),(ExternalTours,6)]
Burns's	607 3000	[(Agencies,2),(ExternalTours,7)]

In our system, to obtain these results we simply use the keyword **lineage** instead of **query**; for example, for `q1` we would simply write:

```
lineage {
  for (a <-- agencies)
    for (e <-- externalTours)
    where (a.name == e.name && e.type == "boat")
      [(name = e.name,
        phone = a.phone)]
}
```

Links's capabilities for normalizing and efficiently evaluating queries provide the key ingredients needed for computing provenance. For both where-provenance and lineage, we can translate programs using the extensions described above, in a way that both preserves types and ensures that the resulting query expressions can be converted to SQL queries. In the rest of this paper, we give the details of these translations and present an experimental evaluation showing that its performance is reasonable.

3. PROVENANCE TRANSLATIONS

In this section we present the key technical contributions of this paper. We present two extensions of Links: $\mathsf{Links}^{\mathsf{W}}$, which supports where-provenance in queries, and $\mathsf{Links}^{\mathsf{L}}$, which supports lineage in queries. We show that both extensions can be implemented by a type-preserving source-to-source translation to plain Links.

$$\frac{\Gamma \vdash M : \mathsf{Prov}(A)}{\Gamma \vdash \textbf{prov } M : (\mathsf{String}, \mathsf{String}, \mathsf{Int})}\ \textsc{Prov} \qquad \frac{\Gamma \vdash M : \mathsf{Prov}(A)}{\Gamma \vdash \textbf{data } M : A}\ \textsc{Data}$$

$$\frac{R :: \mathsf{BaseRow} \qquad \Gamma \vdash S : \mathsf{ProvSpec}(R)}{\Gamma \vdash \textbf{table } n \textbf{ with } (R) \textbf{ where } S : \textbf{table}(R \triangleright S)}\ \textsc{Table}$$

$$\frac{}{\Gamma \vdash \cdot : \mathsf{ProvSpec}(R)} \qquad \frac{\Gamma \vdash S : \mathsf{ProvSpec}(R)}{\Gamma \vdash S, l \textbf{ prov default} : \mathsf{ProvSpec}(R)}$$

$$\frac{\Gamma \vdash S : \mathsf{ProvSpec}(R) \qquad \Gamma \vdash M : (R) \texttt{ -> } (\textbf{String}, \textbf{String}, \textbf{Int})}{\Gamma \vdash S, l \textbf{ prov } M : \mathsf{ProvSpec}(R)}$$

$$R \triangleright \cdot = R \qquad (R, l : O) \triangleright (S, l \textbf{ prov } s) = (R \triangleright S), l : \mathbf{Prov}(O)$$

Figure 6: Additional typing rules for $\mathsf{Links}^{\mathsf{W}}$.

$$\begin{array}{rcl}
\mathfrak{W}[\![O]\!] & = & O \\
\mathfrak{W}[\![A \texttt{ -> } B]\!] & = & \mathfrak{W}[\![A]\!] \texttt{ -> } \mathfrak{W}[\![B]\!] \\
\mathfrak{W}[\![(l_i : A_i)_{i=1}^n]\!] & = & (l_i : \mathfrak{W}[\![A_i]\!])_{i=1}^n \\
\mathfrak{W}[\![[A]]\!] & = & [\mathfrak{W}[\![A]\!]] \\
\mathfrak{W}[\![\mathbf{Prov}(A)]\!] & = & (\mathsf{data} : \mathfrak{W}[\![A]\!], \mathsf{prov} : (\textbf{String}, \textbf{String}, \textbf{Int})) \\
\mathfrak{W}[\![\textbf{table}(R)]\!] & = & (\textbf{table}(\|R\|), () \texttt{ -> } [\mathfrak{W}[\![(R)]\!]]) \\
\|O\| & = & O \\
\|\mathbf{Prov}(A)\| & = & \|A\| \\
\|l_i : A_i|_{i=1}^n\| & = & l_i : \|A_i\||_{i=1}^n
\end{array}$$

Figure 7: Type translation and erasure.

3.1 Where-Provenance

$\mathsf{Links}^{\mathsf{W}}$ extends Links with support for where-provenance. The syntax shown in Figure 3 is extended as follows:

$$\begin{array}{rcl}
O & ::= & \cdots \mid \mathbf{Prov}(O) \\
L, M, N & ::= & \cdots \mid \textbf{data } M \mid \textbf{prov } M \\
 & \mid & \textbf{table } n \textbf{ with } (R) \textbf{ where } S \\
S & ::= & \cdot \mid S, l \textbf{ prov } s \\
s & ::= & \textbf{default} \mid M
\end{array}$$

We introduce the type constructor $\mathbf{Prov}(O)$, where O is a type argument of base type. We treat $\mathbf{Prov}(O)$ itself as a base type, so that it can be used as part of a table type. (This is needed for initializing provenance as explained below.) Values of type $\mathbf{Prov}(O)$ are annotated with where-provenance, where the annotation consists of a triple (R, f, i) where R is the source table name, f is the field name, and i is the row identifier. For example, 42 *#("QA", "a", 23)* represents the answer 42, of type Prov(Int) which was copied from row 23, column a, of table QA. We print the provenance of a value as a comment (following #) to indicate that it can not be directly entered into $\mathsf{Links}^{\mathsf{W}}$. The type $\mathbf{Prov}(O)$ is abstract, without a visible constructor, so only the $\mathsf{Links}^{\mathsf{W}}$ runtime can construct values of provenance type.

There are two operations on values with provenance type: **data** N extracts the data value of some expression N; similarly, **prov** N extracts its argument's where-provenance triple.

In addition, we extend the syntax of table expressions to allow a list of *provenance initialization specifications* l **prov** s. A specification s is either the keyword **default** or an expression M which is expected to be of type $(\overline{l_i : O_i})$ -> (**String**, **String**, **Int**). We have three kinds of columns: (1) regular columns with labels l_r where r is in some set of indices $\mathcal{R}$. For these columns we do not compute provenance. (2) Columns with *default* where-provenance have labels l_d where $d \in \mathcal{D}$. For these columns we compute provenance derived from their location in the database given by table name, column name, and the row's oid. (3) Columns with *external* where-provenance have labels l_e where $e \in \mathcal{E}$. For these columns we obtain provenance by calling a user-provided function with the row as input. Such user-defined provenance calculation functions have to be pure and database-executable, but they are otherwise free to do whatever they want. The envisioned use is fetching existing provenance metadata that is stored separately from the actual data.

The typing rules for the new constructs of $\mathsf{Links}^{\mathsf{W}}$ are shown in Figure 6. These rules employ an auxiliary judgment $\Gamma \vdash S : \mathsf{ProvSpec}(R)$, meaning that in context Γ, the provenance specification S is valid with respect to record type R. As suggested by the typing rule, the **prov** keyword extracts the provenance from a value of type $\mathbf{Prov}(A)$, and **data** extracts its data, the A-value. The most complex rule is that for the **table** construct. The rule for typing table references also uses an auxiliary operation $R \triangleright S$ that defines the type of the provenance view of a table whose fields are described by R and whose provenance specification is S. As for ordinary tables, we check that the fields are of base type.

We give the semantics of $\mathsf{Links}^{\mathsf{W}}$ by a translation to Links. The syntactic translation of types $\mathfrak{W}[\![-]\!]$ is shown in Figure 7. We write $\mathfrak{W}[\![\Gamma]\!]$ for the obvious extension of the type translation to contexts. The implementation extends the Links parser and type checker, and desugars the $\mathsf{Links}^{\mathsf{W}}$ AST to a Links AST after type checking, reusing the backend mostly unchanged. The expression translation function is also written $\mathfrak{W}[\![-]\!]$ and is shown in Figure 8.

Values of type $\mathbf{Prov}(O)$ are represented at runtime in Links as ordinary records with type (data: O,prov: (String, String, Int)). Thus, the keywords **data** and **prov** translate to projections to the respective fields.

We translate table declarations to pairs. The first component is a simple table declaration where all columns have their primitive underlying non-provenance type. We will use the underlying table declaration for insert, update, and delete operations. The second component is essentially a delayed query that calculates where-provenance for the entire table. (The fact that it is delayed is important here, because it means that it can be inlined and simplified later, rather than loaded into memory.) We compute provenance for each record by iterating over the table. For every record of the input table, we construct a new record with the same fields as the table. For every column with provenance, the field's value is a record with data and prov fields. The data field is just the value. The translation of table references also uses an auxiliary operation $R \triangleright_x^n S$ which, given a row type R, a table name n, a variable x and a provenance specification S, constructs a record in which each field contains data from x along with the specified provenance (if any). We wrap the iteration in an anonymous function to delay execution: otherwise, the provenance-annotated table would be constructed in memory when the table reference is first evaluated. We will eventually apply this function in a query, and the Links query normalizer will inline the provenance annotations and

$$
\begin{array}{rcl}
\mathfrak{W}[\![c]\!] &=& c\\
\mathfrak{W}[\![x]\!] &=& x\\
\mathfrak{W}[\![(l_i = M_i)_{i=1}^n]\!] &=& (l_i = \mathfrak{W}[\![M_i]\!])_{i=1}^n\\
\mathfrak{W}[\![N.l]\!] &=& \mathfrak{W}[\![N]\!].l\\
\mathfrak{W}[\![\mathsf{fun}(x_i|_{i=1}^n)\ \{M\}]\!] &=& \mathsf{fun}(x_i|_{i=1}^n)\ \{\mathfrak{W}[\![M]\!]\}\\
\mathfrak{W}[\![M(N_i)]\!] &=& \mathfrak{W}[\![M]\!](\mathfrak{W}[\![N_i]\!])\\
\mathfrak{W}[\![\mathsf{var}\ x = M; N]\!] &=& \mathsf{var}\ x = \mathfrak{W}[\![M]\!]; \mathfrak{W}[\![N]\!]\\
\mathfrak{W}[\![\mathsf{query}\ \{M\}]\!] &=& \mathsf{query}\ \{\mathfrak{W}[\![M]\!]\}\\
\mathfrak{W}[\![[]]\!] &=& []\\
\mathfrak{W}[\![[M]]\!] &=& [\mathfrak{W}[\![M]\!]]\\
\mathfrak{W}[\![M \mathbin{{+}{+}} N]\!] &=& \mathfrak{W}[\![M]\!] \mathbin{{+}{+}} \mathfrak{W}[\![N]\!]
\end{array}
$$

$$
\begin{array}{rcl}
\mathfrak{W}[\![\mathsf{if}\ (L)\ \{M\}\ \mathsf{else}\ \{N\}]\!] &=& \mathsf{if}\ (\mathfrak{W}[\![L]\!])\ \{\mathfrak{W}[\![M]\!]\}\ \mathsf{else}\ \{\mathfrak{W}[\![N]\!]\}\\
\mathfrak{W}[\![\mathsf{empty}\ (M)]\!] &=& \mathsf{empty}\ (\mathfrak{W}[\![M]\!])\\
\mathfrak{W}[\![\mathsf{for}\ (x \text{ <- } L)\ M]\!] &=& \mathsf{for}\ (x \text{ <- } \mathfrak{W}[\![L]\!])\ \mathfrak{W}[\![M]\!]\\
\mathfrak{W}[\![\mathsf{where}(M)\ N]\!] &=& \mathsf{where}(\mathfrak{W}[\![M]\!])\ \mathfrak{W}[\![N]\!]\\
\mathfrak{W}[\![\mathsf{for}\ (x \text{ <-- } L)\ M]\!] &=& \mathsf{for}\ (x \text{ <-- } \mathfrak{W}[\![L]\!].2())\ \mathfrak{W}[\![M]\!]\\
\mathfrak{W}[\![\mathsf{data}\ M]\!] &=& \mathfrak{W}[\![M]\!].\mathsf{data}\\
\mathfrak{W}[\![\mathsf{prov}\ M]\!] &=& \mathfrak{W}[\![M]\!].\mathsf{prov}\\
\mathfrak{W}[\![\mathsf{insert}\ L\ \mathsf{values}\ M]\!] &=& \mathsf{insert}\ \mathfrak{W}[\![L]\!].1\ \mathsf{values}\ \mathfrak{W}[\![M]\!]\\
\mathfrak{W}[\![\mathsf{update}\ (x \text{ <-- } L)\ \mathsf{where}\ M\ \mathsf{set}\ N]\!] &=& \mathsf{update}\ (x \text{ <-- } \mathfrak{W}[\![L]\!].1)\ \mathsf{where}\ \mathfrak{W}[\![M]\!]\ \mathsf{set}\ \mathfrak{W}[\![N]\!]\\
\mathfrak{W}[\![\mathsf{delete}\ (x \text{ <-- } L)\ \mathsf{where}\ M]\!] &=& \mathsf{delete}\ (x \text{ <-- } \mathfrak{W}[\![L]\!].1)\ \mathsf{where}\ \mathfrak{W}[\![M]\!]
\end{array}
$$

$$
\mathfrak{W}[\![\mathsf{table}\ n\ \mathsf{with}(R)\mathsf{where}\ S]\!] = (\mathsf{table}\ n\ \mathsf{with}\ (R),\ \mathsf{fun}()\{\mathsf{for}(x \text{ <-- } \mathsf{table}\ n\ \mathsf{with}\ (R))[(R \triangleright_x^n S)]\})
$$

$$
\begin{array}{rcl}
\cdot \triangleright_x^n \cdot &=& \cdot\\
(R, l : O) \triangleright_x^n \cdot &=& (R \triangleright_x \cdot), l = x.l
\end{array}
\qquad
\begin{array}{rcl}
(R, l : \mathsf{Prov}(O)) \triangleright_x^n (S, l\ \mathsf{prov\ default}) &=& (R \triangleright_x^n S), l = (\mathsf{data} = x.l, \mathsf{prov} = (n, l_d, x.oid))\\
(R, l : \mathsf{Prov}(O)) \triangleright_x^n (S, l\ \mathsf{prov}\ M) &=& (R \triangleright_x^n S), l = (\mathsf{data} = x.l, \mathsf{prov} = \mathfrak{W}[\![M]\!](x))
\end{array}
$$

Figure 8: Translation of Links$^{\mathsf{W}}$ to Links, and auxiliary operation $R \triangleright_x^n S$

normalize them along with the rest of the query.

We translate table comprehensions to comprehensions over the second component of a translated table declaration. Since that component is a function, we have to apply it to a (unit) argument.

For example, recall the example query q1'' from Section 2. The table declaration translates as follows:

```
var agencies = (table "Agencies"
   with (name:String, based_in:String, phone:String),
  fun () { for (t <-- table "Agencies"
           with (name:String, based_in:String, phone:String))
  [(name:t.name, based_in:t.based_in,
    phone=(data=t.phone,prov=("Agencies","phone",t.oid)))] })
```

The translation of the externalTours table reference is similar, but simpler, since it has no **prov** annotations. The query translates to

```
query {
  for (a <-- agencies.2())
    for (e <-- externalTours.2())
    where (a.name == e.name && e.type == "boat")
      [(name = e.name,
        phone = a.phone.data, p_prov = a.phone.prov)]
}
```

Moreover, after inlining the adjusted definitions of agencies and externalTours and normalizing, the provenance computations in the delayed query agencies.2 are also inlined, resulting in a single SQL query.

The (intended) correctness property of the where-provenance translation is that it preserves well-formedness, as follows:

Theorem 1. *For every Links$^{\mathsf{W}}$ term M:*

$$\Gamma \vdash_{\mathit{Links}^W} M : A \Rightarrow \mathfrak{W}[\![\Gamma]\!] \vdash_{\mathit{Links}} \mathfrak{W}[\![M]\!] : \mathfrak{W}[\![A]\!]$$

The proof is straightforward by induction on the structure of derivations; the only interesting cases are those for comprehensions and updates, since they illustrate the need for both the plain table reference and its provenance view.

We have modeled the definition of the where-provenance translation directly on the approach of Buneman et al. (2008) (modulo changes of notation), so we do not formally state or prove the equivalence of the two definitions.

$$
\begin{array}{rcl}
\mathfrak{D}[\![O]\!] &=& O\\
\mathfrak{D}[\![A \text{ -> } B]\!] &=& (\mathfrak{D}[\![A]\!] \text{ -> } \mathfrak{D}[\![B]\!], \mathfrak{L}[\![A]\!] \text{ -> } \mathfrak{L}[\![B]\!])\\
\mathfrak{D}[\![(l_i : A_i)_{i=1}^n]\!] &=& (l_i : \mathfrak{D}[\![A_i]\!])_{i=1}^n\\
\mathfrak{D}[\![[A]]\!] &=& [\mathfrak{D}[\![A]\!]]\\
\mathfrak{D}[\![\mathsf{table}(R)]\!] &=& (\mathsf{table}(R), () \text{ -> } \mathfrak{L}[\![[(R)]]\!])\\
\mathfrak{L}[\![O]\!] &=& O\\
\mathfrak{L}[\![A \text{ -> } B]\!] &=& \mathfrak{L}[\![A]\!] \text{ -> } \mathfrak{L}[\![B]\!]\\
\mathfrak{L}[\![(l_i : A_i)_{i=1}^n]\!] &=& (l_i : \mathfrak{L}[\![A_i]\!])_{i=1}^n\\
\mathfrak{L}[\![[A]]\!] &=& [(\mathsf{data} : \mathfrak{L}[\![A]\!], \mathsf{prov} : [(\mathbf{String}, \mathbf{Int})])]\\
\mathfrak{L}[\![\mathsf{table}(R)]\!] &=& \mathfrak{L}[\![[(R)]]\!]
\end{array}
$$

Figure 9: Doubling and lineage translations

$$
\textsc{Lineage}\quad \frac{\Gamma \vdash M : [A] \qquad A :: \mathsf{QType}}{\Gamma \vdash \mathsf{lineage}\ \{M\} : \mathfrak{L}[\![[A]]\!]}
$$

Figure 10: Additional typing rule for Links$^{\mathsf{L}}$

3.2 Lineage

Links$^{\mathsf{L}}$ adds the **lineage** keyword to Links. The syntax is extended as follows:

$$L, M, N \quad ::= \quad \cdots \mid \mathsf{lineage}\{M\}$$

The expression **lineage** $\{M\}$ is similar to **query** $\{M\}$, in that M must be an expression that can be executed on the database (that is, terminating and side-effect free; this is checked by Links's effect type system just as for **query** {M}). However, instead of executing the query normally, **lineage** {M} also computes lineage for each record in the result. If M has type $[A]$ (which must be an appropriate query result type) then the type of the result of **lineage** $\{M\}$ will be $\mathfrak{L}[\![[A]]\!]$, where $\mathfrak{L}[\![-]\!]$ is a type translation that adjusts the types of collections $[A]$ to allow for lineage, as shown in Figure 10.

The syntactic translation of Links$^{\mathsf{L}}$ types is shown in Figure 9. We write $\mathfrak{D}[\![\Gamma]\!]$ and $\mathfrak{L}[\![\Gamma]\!]$ for the obvious extensions of these translations to contexts. The translation of Links$^{\mathsf{L}}$

$$
\begin{aligned}
\mathfrak{L}[\![c]\!] &= c\\
\mathfrak{L}[\![x]\!] &= x\\
\mathfrak{L}[\![(l_i = M_i)_{i=1}^n]\!] &= (l_i = \mathfrak{L}[\![M_i]\!])_{i=1}^n\\
\mathfrak{L}[\![N.l]\!] &= \mathfrak{L}[\![N]\!].l\\
\mathfrak{L}[\![\textbf{fun}(x_i|_{i=1}^n)\ \{M\}]\!] &= (\textbf{fun}(x_i|_{i=1}^n)\ \{\mathfrak{L}[\![M]\!]\})\\
\mathfrak{L}[\![M(N_i|_{i=1}^n)]\!] &= \mathfrak{L}[\![M]\!](\mathfrak{L}[\![N_i]\!]|_{i=1}^n)\\
\mathfrak{L}[\![\textbf{var}\ x = M; N]\!] &= \textbf{var}\ x = \mathfrak{L}[\![M]\!]; \mathfrak{L}[\![N]\!]\\
\mathfrak{L}[\![\textbf{query}\ \{M\}]\!] &= \textbf{query}\ \{\mathfrak{L}[\![M]\!]\}\\
\mathfrak{L}[\![[]]\!] &= []\\
\mathfrak{L}[\![[M]]\!] &= [(\mathsf{data} : \mathfrak{L}[\![M]\!], \mathsf{prov} : [])]\\
\mathfrak{L}[\![M \mathbin{++} N]\!] &= \mathfrak{L}[\![M]\!] \mathbin{++} \mathfrak{L}[\![N]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathfrak{L}[\![\textbf{if}\ (L)\ \{M\}\ \textbf{else}\ \{N\}]\!] &= \textbf{if}\ (\mathfrak{L}[\![L]\!])\ \{\mathfrak{L}[\![M]\!]\}\ \textbf{else}\ \{\mathfrak{L}[\![N]\!]\}\\
\mathfrak{L}[\![\textbf{query}\ \{M\}]\!] &= \textbf{query}\ \{\mathfrak{L}[\![M]\!]\}\\
\mathfrak{L}[\![\textbf{empty}\ (M)]\!] &= \textbf{empty}\ (\mathfrak{L}[\![M]\!])\\
\mathfrak{L}[\![\textbf{for}\ (x \texttt{ <- } L)\ M]\!] &= \textbf{for}\ (y \texttt{ <- } \mathfrak{L}[\![L]\!])\\
&\qquad \textbf{for}\ (z \texttt{ <- } \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}])\\
&\qquad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \mathbin{++} z.\mathsf{prov})]\\
\mathfrak{L}[\![\textbf{where}(M)\ N]\!] &= \textbf{where}(\mathfrak{L}[\![M]\!])\ (\mathfrak{L}[\![N]\!])\\
\mathfrak{L}[\![\textbf{for}\ (x \texttt{ <- } L)\ M]\!] &= \textbf{for}\ (y \texttt{ <- } \mathfrak{L}[\![L]\!])\\
&\qquad \textbf{for}\ (z \texttt{ <- } \mathfrak{L}[\![M]\!][x \mapsto y.\mathsf{data}])\\
&\qquad\quad [(\mathsf{data} = z.\mathsf{data}, \mathsf{prov} = y.\mathsf{prov} \mathbin{++} z.\mathsf{prov})]\\
\mathfrak{L}[\![\textbf{lineage}\ \{M\}]\!] &= \textbf{query}\ \{\mathfrak{L}[\![M]\!]\}
\end{aligned}
$$

$$
\begin{aligned}
\mathfrak{D}[\![c]\!] &= c\\
\mathfrak{D}[\![x]\!] &= x\\
\mathfrak{D}[\![(l_i = M_i)_{i=1}^n]\!] &= (l_i = \mathfrak{D}[\![M_i]\!])_{i=1}^n\\
\mathfrak{D}[\![N.l]\!] &= \mathfrak{D}[\![N]\!].l\\
\mathfrak{D}[\![\textbf{fun}(x_i|_{i=1}^n)\ \{M\}]\!] &= (\textbf{fun}(x_i|_{i=1}^n)\ \{\mathfrak{D}[\![M]\!]\},\\
&\qquad \mathfrak{L}^*[\![\textbf{fun}(x_i|_{i=1}^n)\ \{M\}]\!])\\
\mathfrak{D}[\![M(N_i|_{i=1}^n)]\!] &= \mathfrak{D}[\![M]\!].1(\mathfrak{D}[\![N_i]\!]_{i=1}^n)\\
\mathfrak{D}[\![\textbf{var}\ x = M; N]\!] &= \textbf{var}\ x = \mathfrak{D}[\![M]\!]; \mathfrak{D}[\![N]\!]\\
\mathfrak{D}[\![[]]\!] &= []\\
\mathfrak{D}[\![[M]]\!] &= [\mathfrak{D}[\![M]\!]]\\
\mathfrak{D}[\![M \mathbin{++} N]\!] &= \mathfrak{D}[\![M]\!] \mathbin{++} \mathfrak{D}[\![N]\!]
\end{aligned}
$$

$$
\begin{aligned}
\mathfrak{D}[\![\textbf{if}\ (L)\ \{M\}\ \textbf{else}\ \{N\}]\!] &= \textbf{if}\ (\mathfrak{D}[\![L]\!])\ \{\mathfrak{D}[\![M]\!]\}\ \textbf{else}\ \{\mathfrak{D}[\![N]\!]\}\\
\mathfrak{D}[\![\textbf{query}\ \{M\}]\!] &= \textbf{query}\ \{\mathfrak{D}[\![M]\!]\}\\
\mathfrak{D}[\![\textbf{empty}\ (M)]\!] &= \textbf{empty}\ (\mathfrak{D}[\![M]\!])\\
\mathfrak{D}[\![\textbf{for}\ (x \texttt{ <- } L)\ M]\!] &= \textbf{for}\ (x \texttt{ <- } \mathfrak{D}[\![L]\!])\ \mathfrak{D}[\![M]\!]\\
\mathfrak{D}[\![\textbf{where}(M)\ N]\!] &= \textbf{where}(\mathfrak{D}[\![M]\!])\ \mathfrak{D}[\![N]\!]\\
\mathfrak{D}[\![\textbf{for}\ (x \texttt{ <- } L)\ M]\!] &= \textbf{for}\ (x \texttt{ <- } \mathfrak{D}[\![L]\!].1)\ \mathfrak{D}[\![M]\!]\\
\mathfrak{D}[\![\textbf{insert}\ L\ \textbf{values}\ M]\!] &= \textbf{insert}\ \mathfrak{D}[\![L]\!].1\ \textbf{values}\ \mathfrak{D}[\![M]\!]\\
\mathfrak{D}[\![\textbf{update}\ (x \texttt{ <- } L)\ \textbf{where}\ M\ \textbf{set}\ \mathfrak{D}[\![N]\!] &= \textbf{update}\ (x \texttt{ <- } \mathfrak{D}[\![L]\!].1)\ \textbf{where}\ \mathfrak{D}[\![M]\!]\ \textbf{set}\ N]\!]\\
\mathfrak{D}[\![\textbf{delete}\ (x \texttt{ <- } L)\ \textbf{where}\ M]\!] &= \textbf{delete}\ (x \texttt{ <- } \mathfrak{D}[\![L]\!].1)\ \textbf{where}\ \mathfrak{D}[\![M]\!]\\
\mathfrak{D}[\![\textbf{lineage}\ \{M\}]\!] &= \textbf{query}\ \{\mathfrak{L}^*[\![M]\!]\}
\end{aligned}
$$

$$
\begin{aligned}
\mathfrak{L}[\![\textbf{table}\ n\ \textbf{with}(R)]\!] &= \textbf{for}(x \texttt{ <- } \textbf{table}\ n\ \textbf{with}\ (R))[(\mathsf{data} = x, \mathsf{prov} = [(n, x.\mathsf{oid})])]\\
\mathfrak{D}[\![\textbf{table}\ n\ \textbf{with}(R)]\!] &= (\textbf{table}\ n\ \textbf{with}\ (R),\ \textbf{fun}()\{\mathfrak{L}[\![\textbf{table}\ n\ \textbf{with}(R)]\!]\})
\end{aligned}
$$

$$
\begin{aligned}
\mathfrak{L}^*[\![M]\!] &= \mathfrak{L}[\![M]\!][x_i \mapsto d2l[\![A_i]\!](x_i)|_{i=1}^n] \text{ where } x_1 : A_1, \ldots, x_n : A_n \text{ are the free variables of } M\\
d2l[\![A]\!] &: \mathfrak{D}[\![A]\!] \to \mathfrak{L}[\![A]\!]\\
d2l[\![O]\!](x) &= x\\
d2l[\![A \texttt{ -> } B]\!](f) &= f.2\\
d2l[\![(l_1 : A_1, \ldots, l_n : A_n)]\!](x) &= (l_1 : d2l[\![A_1]\!](x.l_1), \ldots, l_n : d2l[\![A_n]\!](x.l_n))\\
d2l[\![[A]]\!](y) &= \textbf{for}(x \texttt{ <- } y)[(\mathsf{data} = d2l[\![A]\!](x), \mathsf{prov} = [])]\\
d2l[\![\textbf{table}(R)]\!](t) &= t.2()
\end{aligned}
$$

Figure 11: Translation of Links$^{\mathsf{L}}$ to Links: type translation, outer translation, and inner translation and term translation

expressions to Links is shown in Figure 11. It operates in two modes: $\mathfrak{D}$ and $\mathfrak{L}$. We translate ordinary Links programs using the translation $\mathfrak{D}[\![-]\!]$. When we reach a **lineage** block, we switch to using the $\mathfrak{L}[\![-]\!]$ translation. $\mathfrak{L}[\![[M]]\!]$ provides initial lineage for list literals. Their lineage is simply empty. Table comprehension is the most interesting case. We translate a table iteration **for** $(x$ <-- $L)$ M to a nested list comprehension. The outer comprehension binds y to the results of the lineage-computing view of L. The inner comprehension binds a fresh variable z, iterating over $\mathfrak{L}[\![M]\!]$—the original comprehension body M transformed using $\mathfrak{L}$. The original comprehension body M is defined in terms of x, which is not bound in the transformed comprehension. We therefore replace every occurrence of x in $\mathfrak{L}[\![e]\!]$ by $y.\mathsf{data}$. In the body of the nested comprehension we thus have y, referring to the table row annotated with lineage, and z, referring to the result of the original comprehension's body, also annotated with lineage. As the result of our transformed comprehension, we return the plain data part of z as our data, and the combined lineage annotations of y and z as our provenance. (Handling **where**-clauses is straightforward, as shown in Figure 11.)

One subtlety here is that lineage blocks need not be closed, and so may refer to variables that were defined (and will be bound to values at run time) outside of the lineage block. This could cause problems: for example, if we bind x to a collection $[1, 2, 3]$ outside a lineage block and refer to it in a comprehension inside such a block then uses of x will expect the collection elements to be records such as $(\mathsf{data} = 1, \mathsf{prov} = L)$ rather than plain numbers. Therefore, such variables need to be adjusted so that they will have appropriate structure to be used within a lineage block. The auxiliary type-indexed function $d2l[\![A]\!]$ accomplishes this by mapping a value of type $\mathfrak{D}[\![A]\!]$ to one of type $\mathfrak{L}[\![A]\!]$. We define $\mathfrak{L}^*[\![-]\!]$ as a function that applies $\mathfrak{L}[\![-]\!]$ to its argument and substitutes all free variables $x : A$ with $d2l[\![A]\!](x)$.

The $\mathfrak{D}[\![-]\!]$ translation also has to account for functions that are defined outside lineage blocks but may be called either outside or inside a lineage block. To support this, the case for functions in the $\mathfrak{D}[\![-]\!]$ translation creates a pair, whose first component is the recursive $\mathfrak{D}[\![-]\!]$ translation of the function, and whose second component uses the $\mathfrak{L}^*[\![-]\!]$ translation to create a version of the function callable from within a lineage block. (We use $\mathfrak{L}^*[\![-]\!]$ because functions also need not be closed.) Function calls outside lineage blocks are translated to project out the first component; function calls inside such blocks are translated to project out the second component (this is actually accomplished via the A -> B case of $d2l$.)

Finally, notice that the $\mathfrak{D}[\![-]\!]$ translation maps table types and table references to pairs. This is similar to the $\mathfrak{W}[\![-]\!]$ translation, so we do not explain it in further detail; the main difference is that we just use the oid field to assign default provenance to all rows.

For example, if we wrap the query from Figure 1 in a **lineage** block it will be rewritten to this:

```
for (y_a <- agencies.2())
  for (z_a <- for (y_e <- externalTours.2())
                for (z_e <- [(data = (name = y_a.data.name,
                                      phone = y_a.data.phone),
                              prov = [])])
                where (y_a.data.name == y_e.data.name
                      && y_e.data.type == "boat")
                  [(data = z_e.data,
                    prov = y_e.prov ++ z_e.prov)])
    [(data = z_a.data, prov = y_a.prov ++ z_a.prov)]
```

Once agencies and externalTours are inlined, Links's built-in normalization algorithm simplifies this query to:

```
for (y_a <- table "Agencies" with ...)
  for (y_e <- table "ExternalTours" with ...)
  where (y_a.data.name == y_e.data.name
        && y_e.data.type == "boat")
    [(data = (name = y_a.data.name,phone = y_a.data.phone),
      prov = [("Agencies", y_a.oid), ("ExternalTours",y_e.oid)])]
```

The (again, intended) correctness property for the translation from $\mathsf{Links}^{\mathsf{L}}$ to Links is stated as follows:

THEOREM 2. *Let M be given such that $\Gamma \vdash_{\mathit{Links}^{L}} M : A$. Then:*

1. $\mathfrak{L}[\![\Gamma]\!] \vdash_{\mathit{Links}} \mathfrak{L}[\![M]\!] : \mathfrak{L}[\![A]\!]$
2. $\mathfrak{D}[\![\Gamma]\!] \vdash_{\mathit{Links}} \mathfrak{L}^*[\![M]\!] : \mathfrak{L}[\![A]\!]$
3. $\mathfrak{D}[\![\Gamma]\!] \vdash_{\mathit{Links}} \mathfrak{D}[\![M]\!] : \mathfrak{D}[\![A]\!]$

The proof of each part is straightforward by induction (notice that $\mathfrak{D}[\![-]\!]$ depends on $\mathfrak{L}[\![\Gamma]\!]$ but not vice versa). The main complication is the use of $l2s$ in $\mathfrak{L}^*[\![-]\!]$, and the cases for functions and lineage which need to use the second induction hypothesis. In the case of lineage, we use the fact that $\mathfrak{D}[\![A]\!] = \mathfrak{D}[\![\mathfrak{L}[\![A]\!]]\!]$, which follows because A :: QType so cannot involve table or function types.

Our definition of lineage is inspired by previous presentations for database query languages such as nested relational calculus (Foster et al. 2008); however, our approach differs in considering multiset rather than set semantics. Relating these definitions and is left for future work.

4. EXPERIMENTAL EVALUATION

We implemented two variants of Links with language-integrated provenance, $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$, featuring our extensions for where-provenance and lineage, respectively. Both variants build on Links with query shredding as described by Cheney et al. (2014c); they used queries against a simple test database schema (see Figure 12) that models an organization with departments, employees and external contacts. We change some of their benchmarks to return where-provenance and provenance and compare against the same queries without provenance.

Unlike Cheney et al. (2014c) our database does not include an additional id field, instead we use PostgreSQL's OIDs, which are used for identification of rows in where-provenance and lineage. We populate the databases at varying sizes using randomly generated data in the same way Cheney et al. (2014c) describe it: "We vary the number of departments in the organization from 4 to 4096 (by powers of 2). Each department has on average 100 employees and each employee has 0–2 tasks." The largest database, with 4096 departments, is 142 MB on disk when exported by `pg_dump` to a SQL file (excluding OIDs). We create additional indices on tasks(employee), tasks(task), employees(dept), and contacts(dept).

All tests were performed on an otherwise idle desktop system with a quad-core CPU with 3.2 GHz, 8 GB RAM, and a 500 GB HDD. The system ran Linux (kernel 4.5.0) and we used PostgreSQL 9.4.2 as the database engine. Links and its variants $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ are interpreters written in OCaml, which were compiled to native code using OCaml 4.02.3.

4.1 Where-provenance

To be usable in practice, where-provenance should not have unreasonable runtime overhead. We compare queries *without* any where-provenance against queries that calculate where-provenance on *some* of the result and queries that calculate *full* where-provenance wherever possible. This should give us an idea of the overhead of where-provenance on typical queries, which are somewhere in between full and no provenance.

The nature of where-provenance suggests two hypotheses: First, we expect the asymptotic cost of where-provenance-annotated queries to be the same as that of regular queries. Second, since every single piece of data is annotated with a triple, we expect the runtime of a fully where-provenance-annotated query to be at most four times the runtime of an unannotated query just for handling more data.

We only benchmark *default* where-provenance, that is table name, column name, and the database-generated OID for row identification. External provenance is computed by user-defined database-executable functions and can thus be arbitrarily expensive.

We use the queries with nested results from Cheney et al. (2014c) and use them unchanged for comparison with the two variants with varying amounts of where-provenance.

For *full* where-provenance we change the table declarations to add provenance to every field, except the OID. This changes the types, so we have to adapt the queries and some of the helper functions. Figure 13 shows the benchmark queries with full provenance. Note that for example query Q2 maps the **data** keyword over the employees tasks before comparing the tasks against *"abstract"*. Query Q6 returns the outliers in terms of salary and their tasks, concatenated with the clients with a *fake* task *"buy"*. Since the fake task is not a database value it cannot have where-provenance. $\mathsf{Links}^{\mathsf{W}}$ type system prevents us from pretending it does. Thus, the list of tasks has type [String], not [Prov(String)].

The queries with *some* where-provenance are derived from

```
table departments with (oid: Int, name: String)
table employees with (oid: Int, dept: String,
                      name: String, salary: Int)
table tasks with (oid: Int, employee: String, task: String)
table contacts with (oid: Int, dept: String,
                     name: String, client: Bool)
```

Figure 12: Benchmark database schema, c.f. Cheney et al. (2014c).

```
# Q1 : [(contacts: [("client": Prov(Bool), name: Prov(String))], ...
for (d <-- departments)
  [(contacts = contactsOfDept(d),
    employees = employeesOfDept(d),
    name = d.name)]

# Q2 : [(d: Prov(String))]
for (d <- q1())
where (all(d.employees, fun (e) {
    contains(map(fun (x) { data x }, e.tasks), "abstract") }))
  [(d = d.name)]

# Q3 : [(b: [Prov(String)], e: Prov(String))]
for (e <-- employees)
  [(b = tasksOfEmp(e), e = e.name)]

# Q4 : [(dpt:Prov(String), emps:[Prov(String)])]
for (d <-- departments)
  [(dpt = (d.name),
    emps = for (e <-- employees)
           where ((data d.name) == (data e.dept))
             [(e.name)])]

# Q5 : [(a: Prov(String), b: [(name: Prov(String), ...
for (t <-- tasks)
  [(a = t.task, b = employeesByTask(t))]

# Q6 : [(d: Prov(String), p: [(name: Prov(String), tasks: [String])])]
for (x <- q1())
  [(d = x.name,
    p = get(outliers(x.employees),
          fun (y) { map(fun (z) { data z }, y.tasks) }) ++
        get(clients(x.contacts), fun (y) { ["buy"] }))]
```

Figure 13: “allprov” benchmark queries used in experiments

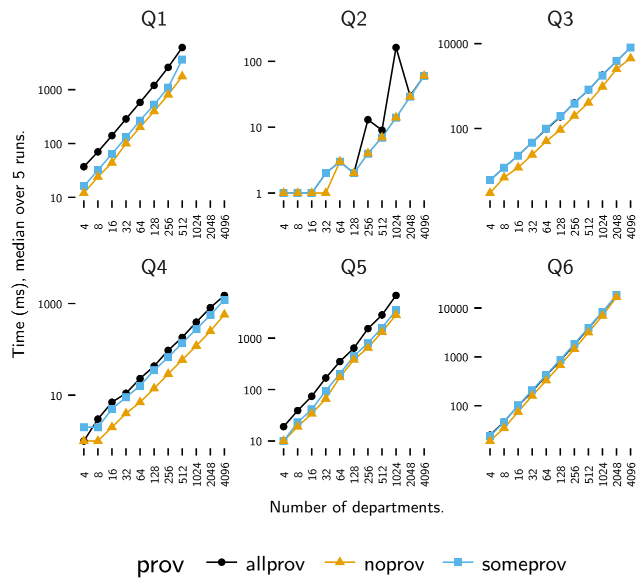

Figure 14: Where-provenance query runtimes.

Query	median runtime* in ms			overall slowdown
	allprov	someprov	noprov	(geom mean)
Q1	6068	3653	1763	2.26
Q2	60	60	60	1.52
Q3	8100	8064	4497	1.88
Q4	1502	1214	573	2.8
Q5	6778	3457	2832	1.85
Q6	17874	18092	16716	1.22

Figure 15: Median runtimes for largest dataset (Q1 at 512 departments, Q5 at 1024 departments, Q6 at 2048 departments, others 4096 departments) and geometric means of overall slowdowns

the queries with full provenance. Query Q1 drops provenance from the contacts' fields. Q2 returns data and provenance separately. It does not actually return less information, it is just less type-safe. Q3 drops provenance from the employee. Q4 returns the employees' provenance only, and drops the actual data. Q5 does not return provenance on the employees fields. Q6 drops provenance on the department. (These queries make use of some auxiliary functions which are included in the appendix.)

Setup. We have three Links^W programs, one for each level of where-provenance annotations. For each database size, we drop all tables and load a dump from disk, starting with 4096. We then run Links^W three times, once for each program in order *all*, *some*, *none*. Each of the three programs performs and times its queries 5 times in a row and reports the median runtime in milliseconds. The programs measure runtime using the Links^W built-in function serverTimeMilliseconds which in turn uses OCaml's Unix.gettimeofday.

Data. Figure 14 shows our experimental results. We have one plot for every query, showing the database size on the x-axis and the median runtime over five runs on the y-axis. Note that both axes are logarithmic. Measurements of full where-provenance are in black circles, no provenance are yellow triangles, some provenance is blue squares. Based on test runs we had to exclude some results for queries at larger database sizes because the queries returned results that were too large for Links to construct as in-memory values.

The graph for query Q2 looks a bit odd. This seems to be due to Q2 not actually returning any data for some database sizes, because for some of the (randomly generated) instances there just are no departments where all employees have the task *"abstract"*.

The table in Figure 15 lists all queries with their median runtimes with full, some, and no provenance. The time reported is in milliseconds, for the largest database instance that both variants of a query ran on. For most queries this is 4096; for Q1 it is 512, 1024 for Q5, and 2048 for Q6. Figure 15 also reports the slowdown of full where-provenance versus no provenance as the geometric mean over all database sizes, for each query. The slowdown ranges from 1.22 for query Q6 up to 2.8 for query Q4.

Interpretation. The graphs suggest that the asymptotic cost of all three variants is the same, confirming our hypothesis. This was expected, anything else would have suggested a bug in our implementation.

The multiplicative overhead seems to be larger for queries that return more data. Notably, for query Q2, which returns no data at all on some of our test database instances, the overhead is hardly visible. The raw amount of data returned for the full where-provenance queries is three to four times that of a plain query. Most strings are short names and provenance adds two short strings and a number for table, column, and row. The largest overhead is 2.8 for query Q4, which exceeds our expectations due to just raw additional data needing to be processed.

4.2 Lineage

We expect lineage to have different performance characteristics than where-provenance. Unlike where-provenance, lineage is conceptually set valued. A query with few actual results could have huge lineage, because lineage is combined for equal data. In practice, due to Links using multiset semantics for queries, the amount of lineage is bounded by the shape of the query. Thus, we expect lineage queries to have the same asymptotic cost as queries without lineage. However, the lineage translation still replaces single comprehensions by nested comprehensions that combine lineage. We expect this to have a larger impact on performance than where-provenance, where we only needed to trace more data through a query.

Figure 16 lists the queries used in the lineage experiments. For lineage, queries are wrapped in a **lineage** block. Our implementation does not currently handle function calls in lineage blocks automatically, so in our experiments we have manually written lineage-enabled versions of the functions employeesByTask and tasksOfEmp, whose bodies are wrapped in a **lineage** block. We reuse some of the queries from the where-provenance experiments, namely Q3, Q4, and Q5. Queries AQ6, Q6N, and Q7 are inspired by query Q6, but not quite the same. Queries QF3 and QF4 are two of the flat queries from Cheney et al. (2014c). Query QC4 computes pairs of employees in the same department and their tasks in a "tagged union". Again, these queries employ some helper functions which are included in an appendix.

We use a similar experimental setup to the one for where-provenance. We only use databases up to 1024 departments, because most of the queries are a lot more expensive. Query QC4 has excessive runtime even for very small databases. Query Q7 ran out of memory for larger databases. We excluded them from runs on larger databases.

Data. Figure 17 shows our lineage experiment results. Again, we have one plot for every query, showing the database size on the x-axis and the median runtime over five runs on the y-axis. Both axes are logarithmic. Measurements with lineage are in black circles, no lineage is shown as yellow triangles.

The table in Figure 18 lists queries and their median runtimes with and without lineage. The time reported is in milliseconds, for the largest database instance that both variants of a query ran on. For most queries this is 1024; for Q7 it is 128, 16 for QC4, and 512 for QF3. The table also reports the slowdown of lineage versus no lineage as the geometric mean over all database sizes. (We exclude database size 4 for the mean slowdown in QF4 which reported taking 0 ms for no lineage queries which would make the geometric

```
typename Lin(a) = (data: a, prov: [(row: Int, "table": String)]);

# AQ6 : [Lin((department: String, outliers: [Lin((name: String, ...
for (d <- for (d <-- departments)
          [(employees = for (e <-- employees)
                        where (d.name == e.dept)
                          [(name = e.name, salary = e.salary)],
            name = d.name)])
  [(department = d.name,
    outliers = for (o <- d.employees)
               where (o.salary > 1000000 || o.salary < 1000)
                 [o])]

# Q3 : [Lin((b: [Lin(String)]), e: String)]
for (e <-- employees) [(b = tasksOfEmp(e), e = e.name)]

# Q4 : [Lin((dpt: String, emps: [Lin(String)]))]
for (d <-- departments)
  [(dpt = d.name,
    emps = for (e <-- employees)
           where (d.name == e.dept)
             [(e.name)])]

# Q5 : [Lin((a: String, b: [Lin((name: String, salary: Int, ...
for (t <-- tasks) [(a = t.task, b = employeesByTask(t))]

# Q6N : [Lin((department: String, people:[Lin((name: String, ...
for (x <-- departments)
  [(department = x.name,
    people = (for (y <-- employees)
              where (x.name == y.dept &&
                     (y.salary < 1000 || y.salary > 1000000))
                [(name = y.name,
                  tasks = for (z <-- tasks)
                          where (z.employee == y.name)
                            [z.task])]) ++
             (for (y <-- contacts)
              where (x.name == y.dept && y."client")
                [(name = y.dept, tasks = ["buy"])]))]

# Q7 : [Lin((department: String, employee: (name: String, ...
for (d <-- departments)
for (e <-- employees)
where (d.name == e.dept && e.salary > 1000000 || e.salary < 1000)
  [(employee = (name = e.name, salary = e.salary),
    department = d.name)]

# QC4 : [Lin((a: String, b: String, c: [Lin((doer: String, ...
for (x <-- employees)
for (y <-- employees)
where (x.dept == y.dept && x.name <> y.name)
  [(a = x.name,
    b = y.name,
    c = (for (t <-- tasks)
         where (x.name == t.employee)
           [(doer = "a", task = t.task)]) ++
        (for (t <-- tasks)
         where (y.name == t.employee)
           [(doer = "b", task = t.task)]))]

# QF3 : [Lin((String, String))]
for (e1 <-- employees)
for (e2 <-- employees)
where (e1.dept == e2.dept && e1.salary == e2.salary
   && e1.name <> e2.name)
  [(e1.name, e2.name)]

# QF4 : [Lin(String)]
(for (t <-- tasks)
 where (t.task == "abstract")
  [t.employee]) ++
(for (e <-- employees)
 where (e.salary > 50000)
  [e.name])
```

Figure 16: Lineage queries used in experiments

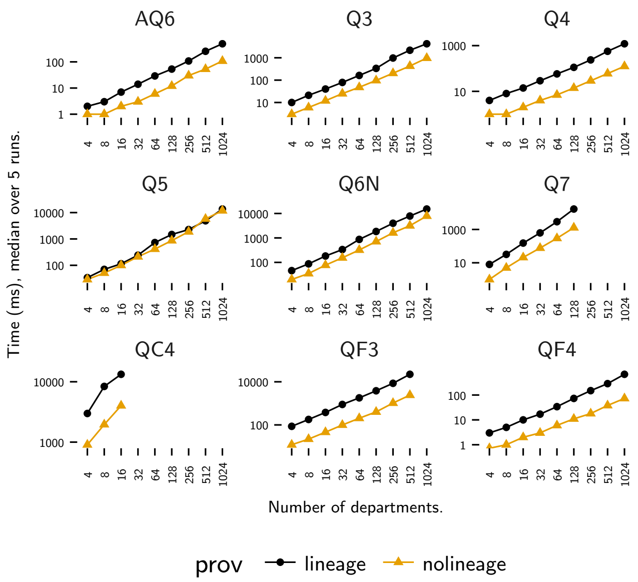

Figure 17: Lineage query runtimes.

Query	median runtime in ms lineage	nolineage	overall slowdown (geom mean)
AQ6	493	108	3.8
Q3	4234	969	3.76
Q4	1208	125	7.55
Q5	13662	11851	1.25
Q6N	15200	7872	2.38
Q7	16766	1283	4.17
QC4	13291	4021	1.53
QF3	22298	2412	6.71
QF4	682	73	6.49

Figure 18: Median runtimes at largest dataset (Q7 at 128 departments, QC4 at 16 departments, QF3 at 512 departments, others at 1024 departments) and geometric means of overall slowdowns

mean infinity.) The performance penalty for using lineage ranges from query Q5 needing a quarter more time to query Q4 being more than 7 times slower than its counterpart.

Interpretation. Due to Links multiset semantics, we do not expect lineage to cause an asymptotic complexity increase. The experiments confirm this. Lineage is still somewhat expensive to compute, with slowdowns ranging from 1.25 to more than 7 times slower. Further investigation of the SQL queries generated by shredding is needed.

4.3 Threats to validity

Our test databases are only moderately sized. However, our result sets are relatively large. Query Q1 for example returns the whole database in a different shape. Links' runtime representation of values in general and database results in particular has a large memory overhead. In practice, for large databases we should avoid holding the whole result in memory. This should reduce the overhead (in terms of memory) of provenance significantly. (It is not entirely clear how to do this in the presence of nested results and thus query shredding.) In general, it looks like the overhead of provenance is dependent on the amount of data returned. It would be good to investigate this more thoroughly. Also, it could be advantageous to represent provenance in a special way. In theory we could store the relation and column name in a more compact way, for example.

One of the envisioned main use cases of provenance is debugging. Typically a user would filter a query anyway to pin down a problem and thus only look at a small number of results and thus also query less provenance. Our experiments do not measure this scenario but instead compute provenance for all query results eagerly. Thus, the slowdown factors we showed represent worst case upper bounds that may not be experienced in common usage patterns.

Our measurements do not include program rewriting time. However, this time is only dependent on the lexical size of the program and is thus fairly small and, most importantly, independent of the database size. Since Links is interpreted, it does not really make sense to distinguish translation time from execution time, but both the where-provenance translation and the lineage translation could happen at compile time, leaving only slightly larger expressions to be normalized at runtime.

5. RELATED WORK

Buneman et al. (2001) gave the first definition of where-provenance in the context of a semistructured data model. The DBNotes system of Bhagwat et al. (2005) supported where-provenance via SQL query extensions. DBNotes provides several kinds of where-provenance in conjunctive SQL queries, implemented by translating SQL queries to one or more provenance-propagating queries. Buneman et al. (2008) proposed a where-provenance model for nested relational calculus queries and updates, and proved expressiveness results. They observed that where-provenance could be implemented by translating and normalizing queries but did not implement this idea; our approach to where-provenance in $\mathsf{Links}^{\mathsf{W}}$ is directly inspired by that idea and is (to the best of our knowledge) the first implementation of it. One important difference is that we *explicitly* manage where-provenance via the **Prov** type, and allow the programmer to decide whether to track provenance for some, all or no fields. Our approach also allows inspecting and comparing the provenance annotations, which Buneman et al. (2008) did not allow; nevertheless, our type system prevents the programmer from forging or unintentionally discarding provenance. On the other hand, our approach requires manual data and prov annotations because it distinguishes between raw data and provenance-annotated data.

$\mathsf{Links}^{\mathsf{L}}$ is inspired by prior work on lineage (Cui et al. 2000) and why-provenance (Buneman et al. 2001). There have been several implementations of lineage and why-provenance. Cui and Widom implemented lineage in a prototype data warehousing system called WHIPS. The Trio system of Benjelloun et al. (2008) also supported lineage and used it for evaluating probabilistic queries; lineage was implemented by defining customized versions of database operations via user-defined functions, which are difficult for database systems to optimize. Glavic and Alonso (2009b) introduced the Perm system, which translated ordinary queries to queries that compute their own lineage; they handled a larger sublanguage of SQL than previous systems such as Trio, and subsequently Glavic and Alonso (2009a) extended this approach to handle queries with nested subqueries (e.g. SQL's EXISTS, ALL or ANY operations). They implemented these rewriting algorithms inside the database system and showed performance improvements of up to 30 times relative to Trio. Our approach instead shows that it is feasible to perform this rewriting outside the database system and leverage the standard SQL interface and underlying query optimization of relational databases.

Both $\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ rely on the conservativity and query normalization results that underly Links's implementation of language-integrated query, particularly Cooper's work (2009) extending conservativity to queries involving higher-order functions, and previous work by Cheney et al. (2014c) on "query shredding", that is, evaluating queries with nested results efficiently by translation to equivalent flat queries. There are alternative solutions to this problem that support larger subsets of SQL, such as Grust et al.'s *loop-lifting* (2010) and more recent work on *query flattening* (Ulrich and Grust 2015), and it would be interesting to evaluate the performance of these techniques on provenance queries.

Other authors, starting with Green et al. (2007), have proposed provenance models based on annotations drawn from algebraic structures such as semirings. While initially restricted to conjunctive queries, the semiring provenance model has subsequently been extended to handle negation and aggregation operations (Amsterdamer et al. 2011). Karvounarakis et al. (2010) developed ProQL, an implementation of the semiring model in a relational database via SQL query extensions. Glavic et al. (2013) present further details of the Perm approach described above, show that semiring provenance can be extracted from Perm's provenance model, and also describe a row-level form of where-provenance. We believe that semiring polynomial annotations can also be extracted from lineage in Links, but supporting other instances of the semiring model via query rewriting in Links appears to be nontrivial due to the need to perform aggregation. In future work, we intend to increase the expressiveness of Links queries to include aggregation and grouping operations and strengthen the query normalization results accordingly.

$\mathsf{Links}^{\mathsf{W}}$ and $\mathsf{Links}^{\mathsf{L}}$ are currently separate extensions, and cannot be used simultaneously, so another natural area for investigation is supporting multiple provenance models at the same time. We have not yet investigated this and it is not clear whether it is straightforward or difficult; one possible difficulty may be the need to combine multiple type translations. We intend to explore this (as well as consider alternative models). Cheney et al. (2014a) presented a general form of provenance for nested relational calculus based on execution traces, and showed how such traces can be used to provide "slices" that explain specific results. While this model appears to generalize all of the aforementioned approaches, it appears nontrivial to implement by translation to relational queries, because it is not obvious how to represent the traces in this approach in a relational data model. (Giorgidze et al. (2013) show how to support *nonrecursive* algebraic data types in queries, but the trace datatype is recursive.) This would be a challenging area for future work.

Our translation for lineage is similar in some respects to the doubling translation used in Cheney et al. (2014b) to compile a simplified form of Links to a F#-like core language. Both translations introduce space overhead and overhead for normal function calls due to pair projections. Developing a more efficient alternative translation (perhaps in combination with a more efficient and more complete compilation strategy) is an interesting topic for future work.

6. CONCLUSIONS

Our approach shows that it is feasible to implement provenance by rewriting queries *outside* the database system, so that a standard database management system can be used. By building on the well-developed theory of query normalization that underlies Links's approach to language-integrated query, our translations remain relatively simple, while still being translated to SQL queries that are executed efficiently on the database. To the best of our knowledge, our approach is the first efficient implementation of provenance for *nested* query results or for queries that can employ *first-class functions*; at any rate, SQL does not provide either feature.

Links is a research prototype language, but the underlying ideas of our approach could be applied to other systems that support comprehension-based language-integrated query, such as F# and Database Supported Haskell. There are a number of possible next steps, including extending Links's language-integrated query capabilities to support richer queries and more forms of provenance. Our results show that provenance for database queries can be implemented efficiently and safely at the language-level. This is a promising first step towards systematic programming language support for provenance.

References

Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS 2011*, pages 153–164, 2011.

O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.

D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4):373–396, 2005.

P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comp. Sci.*, 149(1):3–48, 1995.

P. Buneman, S. Khanna, and W.-C. Tan. Why and where: A characterization of data provenance. In *ICDT 2001*, number 1973 in LNCS, pages 316–330. Springer Berlin / Heidelberg, 2001.

P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.*, 33(4):28:1–28:47, Dec. 2008.

J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, Apr. 2009.

J. Cheney, A. Ahmed, and U. A. Acar. Database queries that explain their work. In *PPDP 2014*, pages 271–282. ACM, 2014a.

J. Cheney, S. Lindley, G. Radanne, and P. Wadler. Effective quotation: Relating approaches to language-integrated query. In *PEPM 2014*, pages 15–26. ACM, 2014b.

J. Cheney, S. Lindley, and P. Wadler. Query shredding: Efficient relational evaluation of queries over nested multisets. In *SIGMOD 2014*, pages 1027–1038. ACM, 2014c.

A. Chlipala. Ur/Web: A simple model for programming the web. In *POPL 2015*, pages 153–165. ACM, 2015.

E. Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *DBPL 2009*, volume 5708 of *LNCS*, pages 36–51. Springer Berlin Heidelberg, 2009.

E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO 2006*, pages 266–296. Springer-Verlag, 2007.

Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, June 2000.

S. Fehrenbach and J. Cheney. Language-integrated provenance in Links. In *TaPP Workshop*, July 2015.

J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: queries and provenance. In *PODS*, pages 271–280, 2008.

G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell boards the ferry: Database-supported program execution for Haskell. In *IFL 2010*, pages 1–18. Springer-Verlag, 2011.

G. Giorgidze, T. Grust, A. Ulrich, and J. Weijers. Algebraic data types for language-integrated queries. In *DDFP 2013*, pages 5–10. ACM, 2013.

B. Glavic and G. Alonso. Provenance for nested subqueries. In *EDBT 2009*, pages 982–993, 2009a.

B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE 2009*, pages 174–185, 2009b.

B. Glavic, R. Miller, and G. Alonso. Using SQL for efficient generation and querying of provenance information. In *Festschrift in Honour of Peter Buneman*, volume 8000 of *LNCS*, pages 291–320. Springer Berlin Heidelberg, 2013.

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

T. Grust and A. Ulrich. First-class functions for first-order database engines. In *DBPL 2013*, 2013.

T. Grust, J. Rittinger, and T. Schreiber. Avalanche-safe LINQ compilation. *PVLDB*, 3(1):162–172, 2010.

G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD 2010*, pages 951–962, 2010.

S. Lindley and J. Cheney. Row-based effect types for database integration. In *TLDI 2012*, pages 91–102. ACM, 2012.

E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *SIGMOD 2006*, pages 706–706. ACM, 2006.

A. Ohori and K. Ueno. Making Standard ML a practical database programming language. In *ICFP 2011*, pages 307–319. ACM, 2011.

M. Serrano. Hop, a fast server for the diffuse web. In *COORDINATION*, 2009.

L. K. Shar and H. B. K. Tan. Defeating SQL injection. *IEEE Computer*, 46(3):69–77, 2013.

D. Syme. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML Workshop*, 2006.

A. Ulrich and T. Grust. The flatter, the better: Query compilation based on the flattening transformation. In *SIGMOD 2015*, pages 1421–1426. ACM, 2015.

L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3), 1996.