A Conceptual Model and Predicate Language for Data Selection and Projection Based on Provenance

David W. Archer and Lois M. L. Delcambre

Department of Computer Science, Portland State University
Portland, OR 97207 USA

{darcher, lmd @cs.pdx.edu}

Abstract

Writing relational database queries over current provenance databases can be complex and error-prone because application data is typically mixed with provenance data, because queries may require recursion, and because the form in which provenance is maintained requires procedural parsing not easily framed in query syntax. As a result, it is often difficult to write queries that select (rows or columns of) data based on provenance. In this paper, we contribute a conceptual model and a predicate language for use in relational algebra that allows the user to write simple, non-recursive queries to select data and attributes based on provenance. Our model also includes novel data and provenance features, including multi-valued attributes, that are useful for data curation settings. We show that our predicate language supports a broad class of queries that select application data based on provenance. We also show how selection of data with our language extensions can be emulated with an existing graph database system and its associated query language.

1. Introduction

Current approaches for provenance of data in relational databases [1-4,6,8] extend the schema to represent provenance directly. As a consequence, a user needs to know which part of the schema is data and which is provenance. To pose a query, the user must mix the query fragment necessary to access and parse the provenance information with the query fragment that accesses the data. Another consequence is that user manipulations or queries may corrupt system-maintained provenance data.

We argue that there is a need to introduce *orthogonality* in provenance databases. That is, we need to keep user data separate from system-maintained provenance in order to prevent clutter and simplify query writing; and we need to keep mechanisms for *manipulating* data from affecting provenance, while retaining the ability for users to *interrogate* both data and provenance together.

Another problem is that since provenance information is typically stored "one step at a time" in current systems, queries that interrogate provenance typically require transitive closure across a possibly unknown number of materialized query answers in order to trace derivation. Such queries are likely difficult for an end-user to write in current query languages.

These problems make it difficult for an end user to pose queries that select data by its provenance (e.g., "Which tuples in this relation were derived from source X?"). Current literature [1-4] does not address this issue, instead often emphasizing queries that result in provenance information (e.g., "What is the provenance of this tuple?"). However, anecdotal evidence from several domains tells us that users often want to select data based on its provenance and either use it immediately (for example in reports), or further analyze it and combine it with other data.

In this paper, we contribute a conceptual model for relational data and its provenance that supports the principle of orthogonality, yet enables simpler syntax for queries that select rows or columns of data based on provenance (as well as data values). The data portion of our model supports relational data with multi-valued attribute values, addressable by relational algebra; data manipulation operators extended with operators to express user confidence in data; and operators for copying data from place to place. Note that users of our model may write regular relational queries against the data without having to consider provenance.

The provenance portion of our model includes full details on how data were derived (inserted, updated, copied, deleted, and re-inserted, as well as queried) and confirmed or doubted, by whom, when, and in what order. The model provides distinct provenance at multiple granularities, including relations, tuples, schema attributes, and attribute values. Our provenance model also supports multiple histories for data, because for example in settings we address, a tuple or an attribute value might be derived from a query result, and then later also inserted by a data manipulation language (DML) operation. That is, we support multiple inser-

tion of tuples and values, as well as multiple values for each attribute.

The provenance portion of our model is addressable by a predicate language for use with the select and project operators of relational algebra. This language allows selection of tuples or attributes based on characteristics in their provenance. Our predicate language enables posing certain classes of transitive provenance queries declaratively in relational algebra, without recursion or complex syntax. The predicate language we define for the selection operator can also be used in conjunction with the usual predicate language that allows users to select tuples based on their attribute values. The predicate language we define for the projection operator can also be used in conjunction with the usual approach of listing attribute names to project.

In this paper, we evaluate the applicability of our predicate language at the conceptual level. We do this by defining an important class of provenance-related queries from settings examined elsewhere in the literature as well as settings we are familiar with. We write queries to answer typical provenance questions from this class using relational algebra and our predicate language. We evaluate the feasibility of implementing our predicate language by showing that it can be mapped to an existing graph query language, GraphQL[5].

In Section 2, we present our conceptual model. Section 3 presents our predicate language for selection and projection of data based on provenance. Section 4 evaluates our predicate language. In Section 5, we discuss related work, and in Section 6, we outline future work.

2. Conceptual Model

In our conceptual model for data and provenance, a *database slice* consists of a finite set of relations with multi-valued attributes. A database slice is a complete instance of a database, with its associated schema, at a particular time. A slice is created whenever an operator is applied to the database. We chose a simple, non first normal form structure (with multiple values per attribute but without nested structure) because in data curation settings, users may need to retain more than one value for an attribute of a tuple, either because they are unsure about the correct value, or because more than one value may apply. The top of Figure 1 shows example relation D with one tuple and two attributes, in a database slice. In this example, the schema of relation D includes the attributes "Name" and "ID". The sole tuple has attribute value "Bob" as its Name, and two values, "8" and "9", for its ID. A *dataloaf* consists of a totally ordered set of database slices, and a set of *external sources* from which data may be inserted into database slices. The ordering of database slices is chronological, denoting the order of applied operations.

We refer to the most recently created database slice in the dataloaf as the *Now* database slice, because it represents the current state of the modelled database. The initial slice in a dataloaf is created when a data definition language (DDL) operation is performed to create its first relation. At the time of its creation, this initial slice is the Now database slice in the dataloaf.

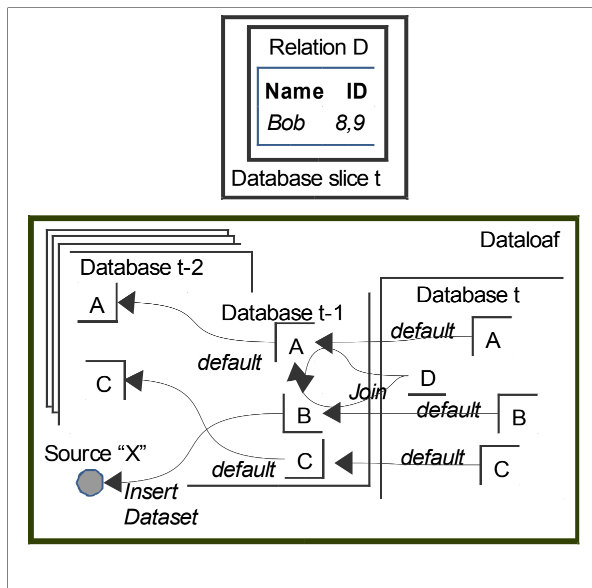

Figure 1. Dataloaf with database slices, relations, external sources, and relation provenance links.

From that point onward, every operator in our conceptual model takes the Now slice in the dataloaf as input, and induces a new Now slice. In the case of a query composed of several operators, only one new slice is created. Each new slice is a copy of the former Now slice, with *components* (relations, tuples, attributes, and attribute values) modified or added as prescribed by the operation performed. When describing operators in our model, we sometimes refer to this new slice as the *result* database slice.

An example dataloaf, showing three database slices and one external source, is at the bottom of Figure 1.
In the figure, we begin at the database slice labeled t-2, where relations A and C exist as the result of prior operations. An insertion operation of a complete relation, B, from an external source, X, is performed on slice t-2, resulting in slice t-1. A query (in this case, consisting of a single Join operator) takes relations A and B as input from slice t-1, and induces slice t, which includes the new relation, D, resulting from the Join operation. At each step, the unaffected contents of the Now slice are copied forward into the result slice, so that the result slice is a complete version of the modeled database.

The *data definition language (DDL)* in our model includes operators for creation of relations, attributes,

and external sources, as well as deletion of relations and attributes. The *data manipulation language (DML)* includes operators for insertion, deletion, and copying of whole relations, individual tuples, and individual attribute values, along with expressions of confirmation and doubt in values. The *query language* consists of Select, Project, Join, and Union operators extended to support multi-valued attribute values. The operators in our language are shown in Figure 2.

Data Definition Language Operators
Create Relation, *External Source*, or Attribute
Delete Relation or Attribute
Data Manipulation Language Operators
Insert Relation, Tuple, or Value[1]
Copy Relation, Tuple, or Value
Delete Tuple or Value
Confirm or Doubt Value
Query operators
Select, Project, Join, Union

Figure 2. Operators for our conceptual model. Operators shown in italics are extensions beyond the relational model.

These operators affect data in the expected way, but also induce provenance relationships among components and external sources. The operators Confirm and Doubt represent user expressions of confidence (or lack thereof) in attribute values. Although these operators do not change data values, our model records these user expressions as part of data provenance.

Operators induce *provenance links* from components in the result database slice to components from which they were derived in the Now slice, or to external sources. Each provenance link originates from one component (the descendant), and has one or more destination components (the ancestors) of the same type (or of type external source), depending on the operation that induces the link. Provenance links have properties, including a *type* (that indicates the operation or query applied, or *default,* or *renew*). We call provenance links labeled with applied operations *action links.* When a result database slice is created from the Now database slice, a *default* provenance link is induced from each unmodified component in the result database slice to its prior version in the Now database slice. In addition, an action link is induced from each newly derived component in the result database slice to each component in the Now database slice or external source that contributes to its presence.

Each component has a boolean attribute called *Expired*. Upon creation, a component's Expired attribute is set to False. When components are deleted, they are copied into result database slices with Expired set to True, and connected by *default* links to their corresponding Now slice components[2]. Deleted components are not available for use by operators, except for re-insertion via DML of a deleted component. Such a re-insertion results in a *renew* link from the re-inserted component in the result database slice to the deleted component in the Now database slice, along with the appropriate action link from the re-inserted component to an external source or database component from which it was inserted or copied.

Figures 3 through 10 show a running example of operations on a dataloaf. Throughout, we show provenance links for relations (densely dotted), tuples (dashed), and data values (solid). We omit default links for clarity. We show only the Now database slice and the result database slice at each step, omitting earlier slices (and links to those) for clarity. Figure 3 shows initial population of a dataloaf with a relation and an external source. Figure 4 shows dataloaf evolution on insertion of a tuple with a single attribute value (i.e., *Name*, with value "*Bob*"). Figures 5 and 6 show the effect of inserting multiple data values for an attribute of a tuple. Figure 7 shows insertion of a second relation as a single operation. Figure 8 shows execution of a simple query over two relations, resulting in a new relation. Figures 9 and 10 show deletion and re-insertion of a data value. (Although the example shows re-insertion from the original source, our model supports re-insertion from a different source as well.)

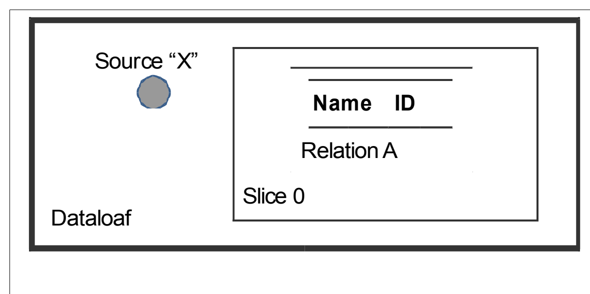

Figure 3: Dataloaf with one slice after operations
Create Source(Name="X");
Create Relation(Name="A", Attributes={"Name", "ID"});

1 Multiple insertions of the same relation, tuple, or value are allowed in our language. This is an extension to the relational model.

2 Deleted data is retained, but distinguished from non-deleted data.

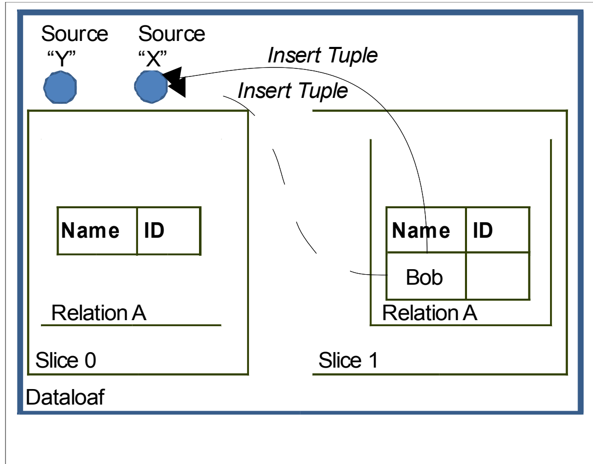

Figure 4. Dataloaf after Insert Tuple(Source="X", Relation="A",{"Name"="Bob"}).

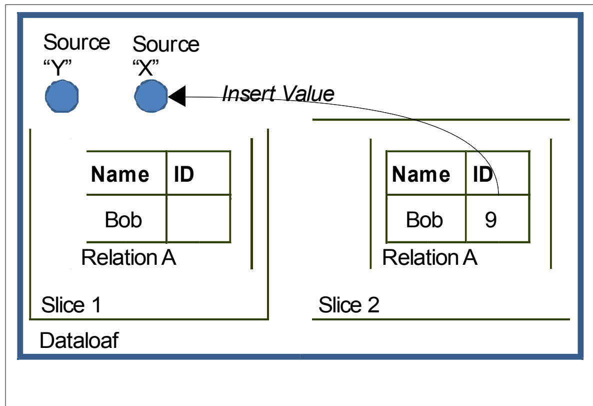

Figure 5. Dataloaf after Insert Value(Source="X", Relation="A", Tuple=1, Attribute="ID", Value="9")

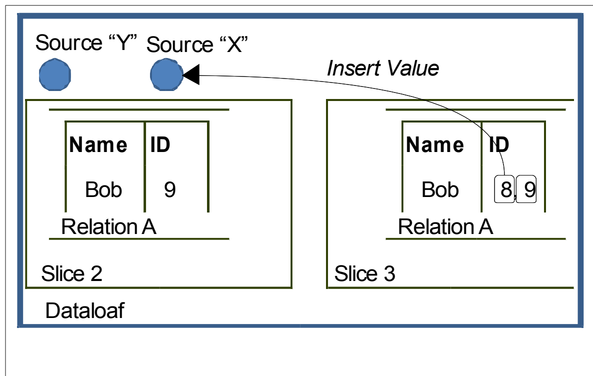

Figure 6. Dataloaf after Insert Value(Source="X", Relation="A", Tuple=1, Attribute="ID", Value="8")

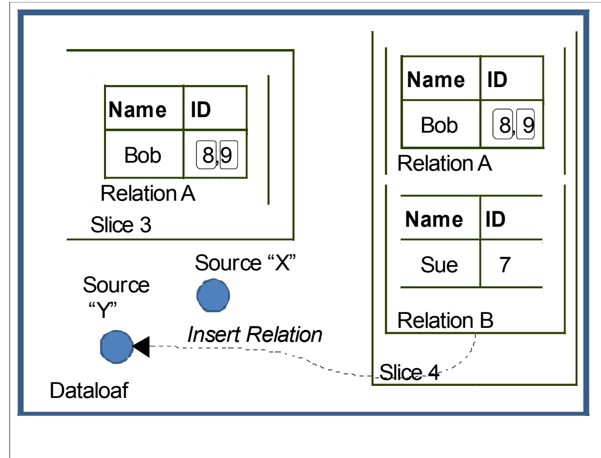

Figure 7. Dataloaf after Insert Relation(Source="Y", Relation="B", {{"Name"="Sue", "ID"="7"}}). Action links omitted for attributes, tuples, and attribute values.

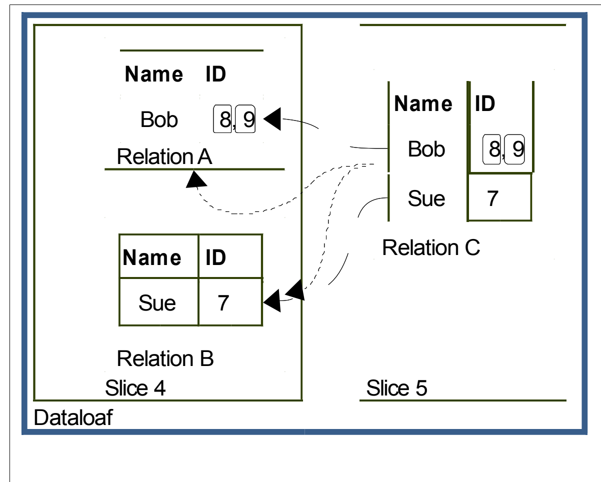

Figure 8. Dataloaf after of $C = A \cup B$. Relations A and B omitted in result slice. Action links omitted for attributes and attribute values.

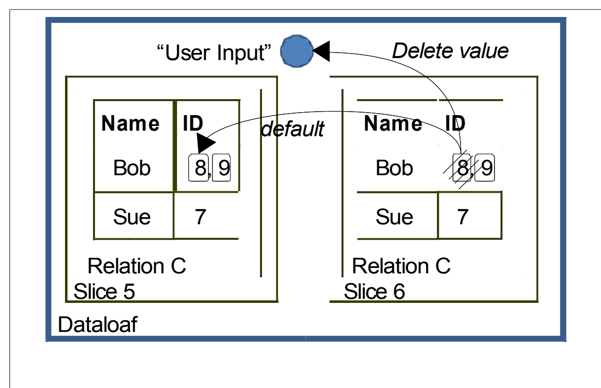

Figure 9. Dataloaf after Delete Value(Src="User input", Relation="C", Tuple=1, Attribute="ID", Value="8"). Default links omitted except for affected value. Non-participating relations omitted. Deleted value shown hashed.

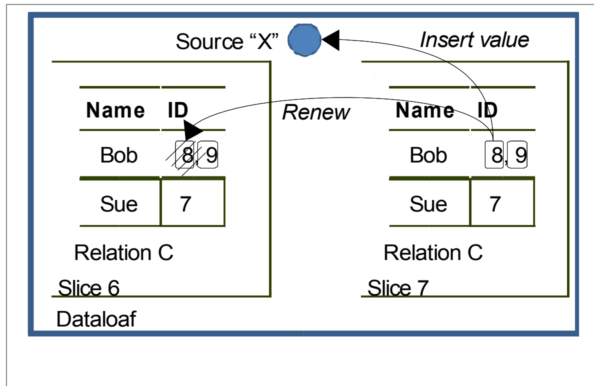

Figure 10. Dataloaf after Insert Value(Source="X", Relation="C", Tuple=1, Attribute="ID", Value="8")

We define *provenance graphs* over a dataloaf that include both data and provenance of a selected component. A provenance graph presents an intuitive picture of the derivations comprising a component's history. In a provenance graph, vertices represent data components[3] or external sources, and edges represent provenance links between them. Provenance graphs are directed and acyclic, but are not in general trees, because multiple components may be ancestors of some component, and a component may be an ancestor of multiple components. Figure 11 shows the provenance graph for the data value "8" in relation "C" in Figure 10. The "dot" notation used in the figure indicates relation first, then tuple ID, then attribute, then attribute value.

We envision that a user might browse the provenance of a component in a dataloaf by selecting it from the Now database slice, using a menu to produce its provenance graph, and then viewing it on a display. By inspection, a user could see components and external sources that contributed to the selected component's provenance, along with the operations applied in deriving the component. We expect that an interface supporting such inspection would allow the user to select links (edges) or components (vertices) and be presented with more detail (such as who initiated an action) as desired. This kind of browsing may enable the user to formulate queries. For example, discovering a particular external source in the provenance of a tuple might prompt the user to query about which other tuples in a relation have provenance including that source.

The action links in our model encode how ancestor data combine to form descendant data. DML and DDL operations contribute single action links with single ancestor terminals. The Where-provenance of components derived by these operations is precisely the external sources or internal database locations from which they were drawn.

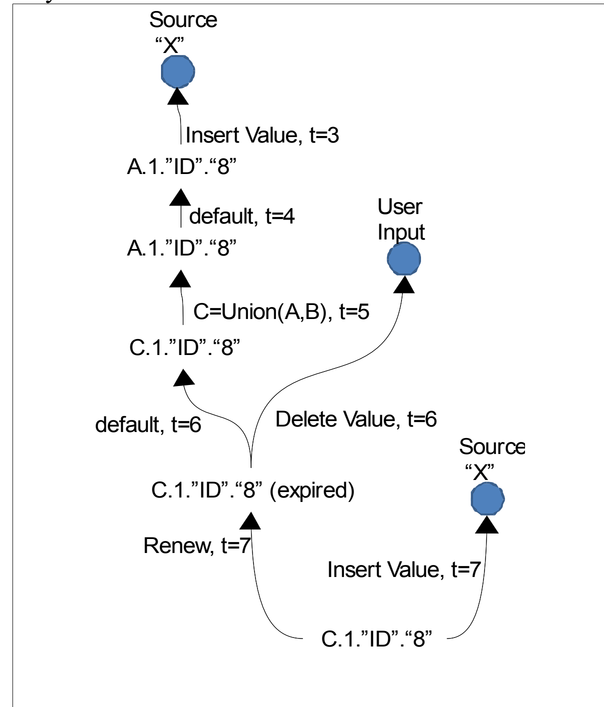

Figure 11. Provenance graph for value "8" in example dataloaf from Figures 3-10.

This definition of Where-provenance matches the implicit provenance semantics of DML as shown by Buneman et al. [21], extended by our inclusion of external data sources in provenance. Our definition of provenance also extends that of Buneman et al. by describing How (the operation history involved), and When (the order of operations). We also extend their work to include provenance of schema attributes.

For queries, we encode *conjunctive* provenance, where the conjunction of the existence of multiple ancestors gives rise to a descendant, using action links with more than one terminal ancestor. We encode *disjunctive* provenance using multiple action links originating at a descendant. Each link contributes a (possibly conjunctive) term to the disjunction. As with DML and DDL, the manner in which provenance "propagates" through individual queries follows the implicit Where-provenance semantics of Buneman et al. [21]. Queries may result in complex provenance, but this provenance can always be described in a "sum-of-products" form. Thus a query may give rise to multiple action links, each of which may have more than one terminal ancestor. Note that when attribute values arise spontaneously from queries, e.g., from constants used in a query expression, they may have no provenance links.

3 All components represented in a provenance graph are of the same type as the selected component, or are external source referents.

This method of encoding provenance allows derivation of other provenance representations, for example those defined by Green et al. [2], Widom et al. [6], and Buneman et al. [1]. Figure 12 shows an example of the provenance of result tuples from the union of two self-joins. The input relation is shown at the top of the figure, followed by the query expression and the result relation. Provenance expressions for the result tuples from selected other models in the literature [1, 6, 2] are shown below the result relation, followed by comparable provenance graphs from our model. The (simplified) leftmost result in the middle of Figure 12 demonstrates the Why-provenance model of Buneman et al. [1]. In this model, provenance consists of the set of sets of tuples that contribute to the presence of a result. Centered is the result from the Trio provenance model [6]. In this model, provenance shows the plurality of groups of tuples, each of which independently causes a result's presence, with each tuple represented once in each group. On the right are provenance polynomials of Green et al. [2]. In this model, provenance also includes how many times a tuple contributes to a group, shown here by exponents.

We have omitted the following from the example MMP provenance graphs at the bottom of the figure for clarity: provenance of relations, attributes, and attribute values; labels on the action links; demarcation of the Now and result database slices involved; and representation of relation R in the result slice, along with related default links. We also use shorthand notation in the graphs to make them easier to read. The notation "S.d", for example, indicates tuple d in relation S.

Consider the provenance graph for tuple d in the result relation S. Action links for d's provenance are shown as solid lines. The leftmost action link has two terminal ancestors, representing a conjunctive contribution to d of a joined with a, as does the next link to the right. The rightmost solid edge shows a conjunctive contribution of a and c to d. In "sum-of-products" form, we can read this provenance graph as, "the provenance of d is aa + aa + ac", equivalent to the provenance semi-ring [2] representation $2a^2 + ac$. Suppose now we perform another operation, inserting a duplicate of tuple d into relation S from an external source called X. This operation would add to the provenance graph of d an external source vertex labeled X, and an action link from d to X, labeled "Insert Tuple". We would read the augmented provenance for d as aa + aa + ac + X. Representation of this kind of provenance was not addressed by Green et al. [2], because in that work, provenance polynomials refer to identifiers of tuples within a database, but do not refer to external sources.

3. Predicate language

While studying domains where relational data and its provenance are both of interest, for example in data curation settings [7-9], we found that many questions asked by domain experts involve transitive rather than "one-step" relationships in provenance graphs. That is, users want to select data items not only by characteristics of immediate ancestors, but by characteristics of ancestor components (as well as actions deriving them, or combinations of the two) anywhere along a *provenance path* in their provenance graph.

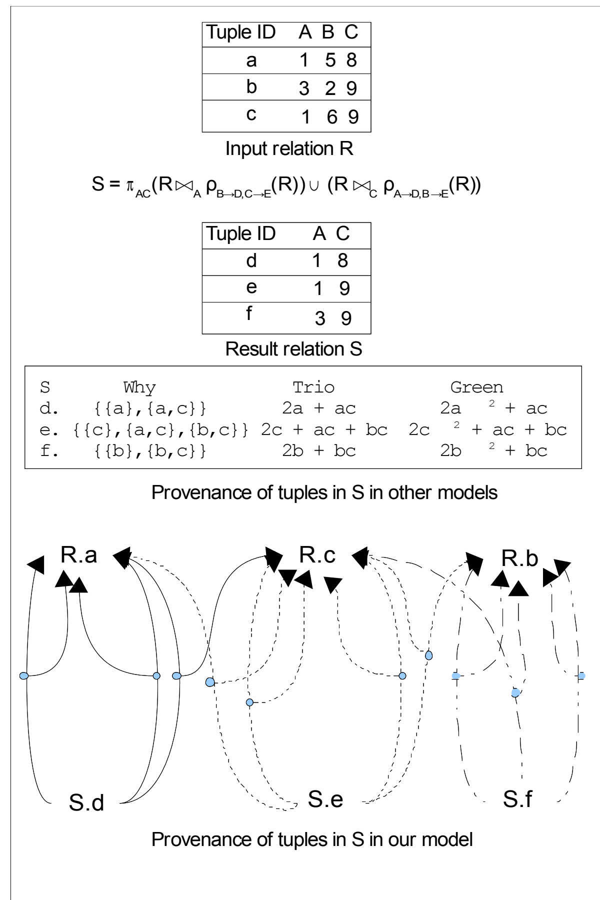

Figure 12: Comparison of provenance representations

A provenance path is a path of finite length in a provenance graph. A language for selecting data based on its provenance should be able to describe the provenance characteristics of data of interest in terms of a pattern, or *motif,* that can be used to identify paths that have those characteristics. Inherent in path motifs is the notion that the precise structure of a path of interest may not be known, and need not be specified. Instead, a user may specify the presence in a path of certain components (vertices) or actions (links) with particular properties (labels). For example, a user might be

interested in data from a particular source, without knowing the full history of the data.

For the language we define in this paper, we assume that action links are labelled with type (operation), identity of the user applying the operation, and the time at which the operation was applied. We assume that all relations in the database have unique names, and that attributes within relations are uniquely named. Our predicate language does not require names for tuples.

```
selectionPredicate ::=
   TUPLE HAS <predicateQualifier>    |
   SOME DATA VALUE IN TUPLE HAS
     <predicateQualifier>                 |
   A VALUE FROM ATTRIBUTES {nameset}
     IN TUPLE HAS <predicateQualifier>
projectionPredicate ::=
   ATTRIBUTE HAS <predicateQualifier> |
   SOME DATA VALUE IN ATTRIBUTE HAS
     <predicateQualifier>
predicateQualifier ::=
   A PATH WITH (<pathQualifier>)       |
   A PATH WITH (<pathQualifier>) [AND|OR]
     <predicateQualifier>
pathQualifier ::=
   A <component>[4] (<cQualSet>)        |
   AN OPERATION (<aQualSet>)        |
   A SOURCE (<sQualSet)                  |
   NOT <pathQualifier>                    |
    <pathQualifier> [BEFORE|AND|OR] <pathQualifier>
aQualSet ::= <aQual> | <aQual> [AND|OR] <aQualSet>
cQualSet ::= <cQual> | <cQual> [AND|OR] <cQualSet>
sQualSet ::= <sQual> | <sQual> [AND|OR] <sQualSet>
aQual ::=  WITH ACTION = <constant>     |
           WITH ACTION = A QUERY       |
           BY USER = <constant>           |
           WHERE TIME <cCmp> <constant>
cQual ::=  IN DATASET <cCmp> <constant> |
           WITH A VALUE <cCmp> <constant> |
           THAT IS EXPIRED
sQual ::=  WITH NAME <cCmp> <constant>
component ::=  tuple | attribute | value
cCmp ::=    =  |  >  |  <  |  ≥  |  ≤  |  ¬=
```

Figure 13: Syntax of Predicate Language

We define a grammar for a language that expresses predicates usable in the projection and selection operators of relational algebra. Figure 13 shows this grammar in BNF form. Our grammar is intentionally verbose in order to make predicate semantics clear. A *projectionPredicate* is a predicate for use in the projection operator, while a *selectionPredicate* is used in the selection operator. The selectionPredicate structure offers three options. A user may select tuples by their tuple provenance, by the provenance of any data value describing the tuple, or by the provenance of data values from only specifically named attributes describing the tuple. The projectionPredicate offers two similar options.

4 Component type in a PathQualifier must agree with the component type specified in the selectionPredicate or projectionPredicate, e.g., *value* must be used if the predicate specifies “...SOME DATA VALUE IN...”

4. Evaluation of Our Predicate Language

Applicability

Provenance selection queries [10] select data from input relations based on their provenance information. Such queries exhibit closure over a data model including both data and provenance. This paper focuses on predicates for provenance selection queries. We subdivide this category of queries as shown in Figure 14.

	Provenance Criteria for Selection Based on...			
Component to Select	Ancestor Data	Derivation Actions	Both	**Selection Criteria**
Tuples	S1	S4	N/A	Single
	S2	S5	S7	Unordered
	S3	S6	S8	Ordered
Attributes	S9	S12	N/A	Single
	S10	S13	S15	Unordered
	S11	S14	S16	Ordered

Figure 14: Provenance Selection Queries. Subdivisions labels match examples below.

Predicates can be used to select tuples or schema attributes (corresponding to the selection and projection operators). The selection criteria can be properties of ancestor data (such as the identity of the dataset to which data belongs, the value of data, whether or not the data has been deleted, or the name of an external source), actions used in derivation (such as the type of operation, the user who applied the operation, or the time it was applied), or both. A single criterion, or a set of criteria combined using AND, OR, and NOT, or a chronologically ordered set of criteria can be specified. We provide example provenance questions for selected subdivisions of the class, numbered as shown in Figure 14, along with syntax for a matching selection predicate in our language. For selection predicates, the resulting query would be of the form $D_O = \sigma_{predicate}(D_I)$, with input relation D_I and output relation D_O. For projection predicates, the resulting query would be of the form $D_O = \pi_{predicate}(D_I)$. We show the query output resulting from application of examples S1-S5 to relation C in the database slice shown in Figure 10.

S1. Which tuples were derived from source "X"?
Predicate:
`tuple has a path with (a source with name = "X")`
Returns: (Bob, {8,9})

S2. Which tuples have at least one data value derived from relation "A" or relation "B"?
Predicate:
some data value in tuple has
a path with (a value in relation = "A")
or a path with (a value in relation = "B")
Returns: (Bob, {8,9}) , (Sue, 7)

S3. Which tuples contain data derived from data in relation "A" that later appeared in relation "C"?
Predicate:
some data value in tuple has
a path with (a value in relation = "A"
before a value in relation = "C")
Returns: (Bob, {8,9})

S4. Which tuples are derived from tuples that were inserted at least once between timestamps "4" and "7"?
Predicate:
tuple has a path with (an operation
with action = "INSERT"
and where time >= "4" and where time < "7")
Returns: (Sue, 7)

S5. Which tuples were both derived by a query and inserted directly (without an intervening query)?
Predicate:
tuple has
a path with (an operation with action = "INSERT"
and not an operation with action = a query)
and a path with (an operation with action = a query)
Returns: nothing

S6. Which tuples had values derived from data inserted between dates "D" and "E" by user "Y", and later deleted?
Predicate:
some data value in tuple has
a path with (an operation with (action = "INSERT"
and where time > "D" and where time < "E"
and by user = "Y")
before a value that is expired)

S12. Which attributes were derived by a query posed by user "Y"?
Predicate:
attribute has a path with (an operation
with action = a query and by user = "Y")

Feasibility

We evaluate the feasibility of implementing our predicate language by showing how predicate evaluation can be emulated with an existing logical data model and query language. Since we represent provenance in our conceptual model as graphs, and since our predicates describe paths to search for in those graphs, we chose a graph data model and graph query language as our substrate. Although the graph model we chose is more expressive than our predicate language, we use it only to show the feasibility of implementing our predicate language.

He and Singh [5] define a formal language for describing graph motifs, and a query language, GraphQL, that takes these motifs as inputs and retrieves graphs that contain them from an existing database of graphs. Their formal language supports descriptions of simple, fixed graph motifs, as well as paths, cycles, and general repetitive motifs. A graph pattern is a description in this language of a graph's connectivity, along with a set of criteria for matching selected edges and vertices in the pattern. He and Singh also define a Selection operator that takes as input a database of graphs and a graph pattern, and produces graphs from the database that match the pattern.

A selectionPredicate or projectionPredicate in our language specifies that tuples (respectively attributes) in the relation in the Now database slice named in a selection (respectively projection) operator will be selected for output based on whether they satisfy the specified predicateQualifier. We describe the selection operator, noting that the description also applies to the projection operator.

We prepare for evaluation of the predicate against a slice in a dataloaf by generating a database of provenance graphs to be searched and converting the selection predicate into a set of path motifs to search for in the generated graphs. All tuples in the input relation to the Select operator are candidates. The selectionPredicate indicates whether the provenance of the tuples, or the provenance for part or all of their attribute values will be searched. The relevant provenance graphs are then generated and included in the graph database.

A tuple is selected for output if the logical expression in the predicateQualifier is satisfied by at least one of its relevant provenance graphs. A predicateQualifier is satisfied by a graph if its logical expression evaluates to True after all path motifs specified by pathQualifiers in the predicateQualifier have been searched for in the graph. Each pathQualifier can be described using a single path motif. Component QualSets describe required properties of component vertices in a motif. Action QualSets describe required properties of edges in a motif. Source Qualsets describe properties of external source vertices in a motif. Negation is supported, in that a pathQualifier can specify that no paths in a graph may satisfy the specified motif. A pathQualifier may be compound, allowing it to describe a combination of vertex and edge requirements in a motif.

Consider example S6. The selectionPredicate specifies that provenance of all attribute values of tuples in

the input relation should be searched for matches to the specified provenance path description. Thus the graph database created for predicate evaluation consists of the provenance graphs for all attribute values in all tuples in the input relation. A single path motif is created using the description in the predicateQualifier. In this example, the motif consists of a path beginning at the vertex representing a Now slice attribute value, and consisting of any connected path (including paths of zero length), followed by a vertex with Expired = True, followed by a connected path (including paths of zero length), followed by an edge labeled "INSERT" with a timestamp between values "D" and "E" and with a user equal to "Y". We use GraphQL's selection operator to search for this motif in the provenance graphs. If at least one such graph is found among the graphs for the attribute values of a tuple, the tuple is selected for output.

5. Related work

Existing provenance databases [1,3,8] use relational query languages (e.g., SQL, relational algebra, or Datalog) with recursion to pose provenance queries. Although several commercial relational database systems support SQL-99 transitive query features, the syntax is complex and likely error-prone, requiring multiple subqueries and construction of views.

The Trio [6] architecture supports the built-in function Lineage() as part of its TriQL query language [12]. Lineage() is applied to two relations and produces tuples from the first relation that have tuples from the second in their lineage. Because Trio records provenance at the tuple level only, TriQL cannot address subdivisions S9-S16 in Figure 14 (attribute selection queries). Lineage() also does not appear to support provenance selection queries involving actions. As a result, Lineage() does not support subdivisions S4-S8 in Figure 14. Nor can Lineage() express predicates that check for specific ordering of ancestor relations in a tuple's provenance. Thus Trio's Lineage() function can express predicates only for portions of S1 and S2 from Figure 14. In contrast, our approach can pose queries from all subdivisions of the class.

The Provenance Management Framework developed by Microsoft Research and Wright State University [10] includes a provenance algebra with an operator, *provenance_context()*, for selecting relations based on workflow provenance. However, the authors do not explore applying *provenance_context()* to relational data.

PASS [13] is a provenance-aware file system. The PASS team discusses the suitability of various query languages (SQL/relational algebra, XPath, and RDF-focused languages) for querying provenance. They suggest a semi-structured logical data model for storing both data and provenance in graph form, and discuss augmenting an existing graph query language, Lorel [14], in order to express provenance queries. These extensions adapt Lorel to traversing provenance graphs, but do not address orthogonality of provenance and user data. PASS extensions to Lorel are focused on filesystem provenance, though they may be applicable to relational data as well.

Cluet discusses querying objects and their relationships, including transitive (path-based) relationships [15] in object-based data models. Since provenance is a relationship between objects (database components, in our case), the inter-object path queries discussed by Cluet are related to our work. However, Cluet does not discuss properties of relationships, e.g. operations or timestamps of provenance links, in path queries.

SchemaSQL [16] provides extensions to SQL that enable uniform manipulation of data and schema, including querying of schema. Our predicate language for the projection operator is similar to, but much simpler than SchemaSQL's ability to query schema, though our language is restricted to querying only the provenance of schema. SchemaSQL does not support provenance.

We examined several graph query languages before selecting GraphQL [5] as our feasibility target. While none of these models address provenance directly, all provide some query capability over graph structures similar to those we use for provenance.

UnQL [17] uses a procedural query language to query graphs, but has no notion of evaluating predicates over vertices in paths. StruQL [18] and GOOD [19] take input graphs and generate new output graphs, with transformation rules provided in the form of queries. StruQL queries are declarative, while GOOD transformations are imperative. Neither provides the requisite capability for applying predicates to vertices and edges in paths as part of the transformation process. GRAM [20] includes a graph query language for *hyperwalk expressions* that describe path patterns to be matched in a graph database. However, restrictions on node labeling in GRAM prevent representing database components as vertices.

6. Conclusions and future work

We have defined a conceptual model for relational data and its provenance that supports orthogonal access to data and provenance. Our model enables queries posed in relational algebra, without recursion or complex syntax, to select rows or columns of data based on provenance. Our model does not handle certain classes of queries, for example queries that extract provenance directly. We will provide an open-ended query capability to do this in a logical model we are developing. We

also intend to evaluate performance and storage trade-offs in our logical model. Our conceptual model addresses how data at all granularities were derived, by whom, when, and in what order. We conjecture that our predicate language may be simplified to match less informative provenance data. In this paper, we have shown applicability of our language to an important class of queries. We have also shown its implementation feasibility at the conceptual level.

Acknowledgments

This work was supported by NSF grant 0534762 and 0840668, and by DARPA.

References

[1] Buneman, P., Khanna, S., Tan, W. "Why and where: a characterization of data provenance," In *Proc. of the 8th Int'l. Conf. on Database Theory*, 2001.

[2] Green, T., Karvounarakis, G., Tannen, V. "Provenance semirings," In *Proc. of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2007.

[3] Bhagwat, D., Chiticariu, L., Tan, W., Vijayvargiya, G. "An annotation management system for relational databases," In *Proc. of the 30th Int'l. Conf. on Very Large Data Bases*, 2004.

[4] Holland, D., Braun, U., Maclean, D., Muniswamy-Reddy, K., Seltzer, M. "Choosing a data model and query language for provenance," In *Proc. of the 2nd Int'l. Provenance and Annotation Workshop*, 2008.

[5] He, H. and Singh, A. K. 2008. "Graphs-at-a-time: query language and access methods for graph databases," In *Proc. of the 2008 ACM SIGMOD Int'l. Conf. on Mgmt of Data*, 2008.

[6] Widom, J. "Trio: A system for integrated management of data, accuracy, and lineage," In *Proc. of CIDR 2005,* 2005.

[7] Archer, D., Delcambre, L., Maier, D. "A framework for fine-grained data integration and curation, with provenance, in a dataspace." In *Proc. of the 1st Workshop on the Theory and Practice of Provenance,* USENIX, 2009.

[8] Buneman, P., Chapman, A., Cheney, J., and Vansummeren, S. "A provenance model for manually curated data," In *Proc. of the Int'l. Provenance and Annotation Workshop*, 2006.

[9] Green, T., Karvounarakis, G., Taylor, N., Biton, O., Ives, Z., Tannen, V. "ORCHESTRA: facilitating collaborative data sharing," In *Proc. of the 2007 ACM SIGMOD Int'l. Conf. on Mgmt of Data*, 2007.

[10] Sahoo, S., Barga, R., Goldstein, J., Sheth, A. "Provenance algebra and materialized view-based provenance management", *Technical Report MSR-TR-2008-170*, Microsoft Corporation, 2008.

[11] Buneman, P., Chapman, A., and Cheney, J. "Provenance management in curated databases," In *Proc. of the 2006 ACM SIGMOD Int'l. Conf. on Mgmt of Data* (SIGMOD '06). ACM, 2006.

[12] TriQL: The Trio Query Language. Available from http://infolab.stanford.edu/trio.

[13] Muniswamy-Reddy, K., Holland, D. A., Braun, U., and Seltzer, M. "Provenance-aware storage systems," In *Proc. of the Annual Conf. on USENIX '06,* 2006.

[14] Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J. "The Lorel query language for semistructured data," In *Int'l. Journal on Digital Libraries 1(1),* 1997.

[15] Cluet, S. 1998. "Designing OQL: allowing objects to be queried," *Information Systems* 23, 5, 1998.

[16] Lakshmanan, L., Sadri, F., and Subramanian, S. "SchemaSQL: an extension to SQL for multidatabase interoperability," *ACM Transactions on Database Systems* 26, 4, 2001.

[17] Buneman, P., Fernandez, M., and Suciu, D. 2000. "UnQL: a query language and algebra for semistructured data based on structural recursion," *The VLDB Journal* 9, 1, 2000.

[18] Fernandez, M., Florescu, D., Levy, A., and Suciu, D. "A query language for a web-site management system," *SIGMOD Record* 26, 3, 1997.

[19] Gemis, M. Paradaens, J., Thyssens, I., et al. "GOOD: A graph oriented object database system," In *Proc. of the 1993 ACM SIGMOD Int'l. Conf. on Mgmt of Data* (SIGMOD '93), 1993.

[20] Amann, B., Scholl, M. "Gram: a graph data model and query language," In *Proc. of the 4th ACM Conf. on Hypertext and Hypermedia*, 1992.

[21] Buneman, P., Cheney, J., Vansummeren, S. "On the expressiveness of implicit provenance in query and update languages," *ACM Trans. Database Syst.* 33, 4, 2008.