Dissertations and Theses

Dissertations and Theses

1-1-2011

Conceptual Modeling of Data with Provenance

David William Archer
Portland State University

Let us know how access to this document benefits you.

Follow this and additional works at: http://pdxscholar.library.pdx.edu/open_access_etds

Recommended Citation

Archer, David William, "Conceptual Modeling of Data with Provenance" (2011). *Dissertations and Theses.* Paper 133.

10.15760/etd.133

Conceptual Modeling of Data with Provenance

by

David William Archer

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Lois M. L. Delcambre, Chair
David Maier
Leonard Shapiro
Mark Jones
Charles Weber

Portland State University
ⓒ2011

ABSTRACT

Traditional database systems manage data, but often do not address its provenance. In the past, users were often implicitly familiar with data they used, how it was created (and hence how it might be appropriately used), and from which sources it came. Today, users may be physically and organizationally remote from the data they use, so this information may not be easily accessible to them. In recent years, several models have been proposed for recording provenance of data. Our work is motivated by opportunities to make provenance easy to manage and query. For example, current approaches model provenance as expressions that may be easily stored alongside data, but are difficult to parse and reconstruct for querying, and are difficult to query with available languages. We contribute a conceptual model for data and provenance, and evaluate how well it addresses these opportunities. We compare the expressive power of our model's language to that of other models. We also define a benchmark suite with which to study performance of our model, and use this suite to study key model aspects implemented on existing software platforms. We discover some salient performance bottlenecks in these implementations, and suggest future work to explore improvements. Finally, we show that our implementations can comprise a logical model that faithfully supports our conceptual model.

DEDICATION

To Cynthia

Acknowledgements

This research was a team effort, and was successful because of the contributions of everyone involved. First and foremost among contributors was my advisor, Professor Lois M. L. Delcambre, who consistently pushed me to think about the big picture, always brought to the table new ideas for us to consider, and was a patient and thorough reviewer of this work. The members of my thesis committee also contributed new perspectives, ideas that I had missed, and excellent constructive critique. For these I thank Professor David Maier, Professor Leonard Shapiro, Professor Mark Jones, and Professor Charles Weber. I also appreciate the ideas, critiques, and inputs of other faculty and students in the PSU DataLab research group: Kristin Tufte, Rafael J. Fernandez-Moctezuma, Jeremy Steinhauer, Nick Rayner, Scott Brittell, and James Terwilliger.

This work was supported in part by the National Science Foundation, grant IIS-0534762, and by DARPA. Support for this work came from a number of others, including the PSU Computer Science Department staff: Beth Holmes, Kathi Lee, and Rene Remillard.

I recognize and deeply appreciate the contributions, patience, and encouragement of my wife, Cynthia L. Archer, PhD. Without her, this achievement would have been impossible.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditional database management systems focus on providing users with efficient ways to insert, update, delete, and query data. However, the *provenance* of data is not addressed by these systems. The term provenance derives from the Latin roots *pro-*, meaning "before" or "in front of", and *veni*, meaning "to come", and appears for example in the French *provenant*, meaning "to come forth from". With regard to fine art, provenance is the record of who owned a work of art during what time period, from its creation to the present. An authentic provenance record is considered *prima facie* evidence that a work is genuine. In livestock or pet breeding, provenance is typically called pedigree, and is widely used as evidence of quality of a particular specimen. With regard to data, provenance is the record of which pre-existing data gave rise to the data, by what operations, under what conditions, when, and under the control of what agent[1].

Many application domains can benefit from data models that include provenance. For example, in accounting, there may be legal obligations to identify the provenance of ledger entries. In eScience, protein databases like Swiss-Prot [2] are populated by a diverse group of researchers using results of numerous experiments, and then con-

---

[1]This is the definition of provenance used in this work. Other database researchers use somewhat narrower definitions of provenance that may include only what pre-existing data gave rise to the data, or may include information about pre-existing data as well as some information about how that data was manipulated.

tinuously revised and improved as new data becomes available. In order to know whether selected data can be used in the context of a specific study or experiment, it may be necessary to know where the data came from, and how it was manipulated. Below, we describe three settings in the domains of medical research data management, corporate budget forecasting, and battlefield information management. In each of these, users benefit from provenance, yet current practice and current literature fall short of making provenance easy to record, query, and manage.

Current tools do not make it easy to interrogate provenance in order to inform decisions about data. In the literature, several logical models for data and provenance have been defined and demonstrated [1, 4, 5, 8, 12, 21, 24]. These models typically focus on ways to represent provenance, mechanisms for computing and storing provenance, and approaches to constructing provenance-related queries using existing database languages. In this work we contribute and evaluate a conceptual model for data and its provenance. We focus on making provenance-related queries easy for users to write, making it easy for users to describe the provenance characteristics of data they wish to select, bringing the semantics of provenance into full view of users, and managing provenance as relationships *among* data instead of as attributes *of* data. We also contribute a logical model that faithfully supports our conceptual model, and a provenance benchmark that allows for studying performance trade-offs for implementations of our logical model.

This chapter introduces several settings that motivate our work, identifies opportunities for contributions in our conceptual model, and defines our research objectives. In Section 1.1 we introduce settings for provenance and data. In Section 1.2 we briefly introduce each provenance model from the literature and discuss where these models fall short against the opportunities identified in Section 1.1. In Section 1.3 we define the research objectives for this work.

1.1 Example Settings for Provenance

The settings described below highlight five significant opportunities for contributions in provenance modeling, and enumerate other features useful in a conceptual model for provenance and data.

1.1.1 Development of Targeted Cancer Therapies

Prioritizing individualized therapies for cancer patients is an iterative process where data is subject to a mix of manipulations and queries[2]. The process begins with insertion of patient background, family history, and tumor or lesion evidence into a relational database from external data source such as patient medical records. Patient samples are obtained and analyzed, and the resulting gene sequence data are also inserted into the database. Next, queries match this patient sample data against reference databases of cancer-causing gene sequences. This analysis results in a ranked distribution of likely causative gene mutations, which is also recorded in the database. This materialized ranking information is in some cases subject to data manipulation (DML) operations by clinicians, as they may rule out or re-prioritize causes based on their expert knowledge. Next, additional queries match the list of likely causative genes with data obtained from medical literature on known inhibitor drug – gene expression interactions. The result is a ranking of likely effective drugs, which is also stored in the database. After review of medical case history literature (to find outcome data for trials of candidate drugs on other patients with similar histories and phenotypes to those of the patient), a therapy is selected. The selected therapy is recorded in the database, and therapy begins. Patient response is documented in the database. Depending on the patient's trajectory in therapy, the therapy prioritization

---

[2]Our description of the setting described here is the result of informal collaboration during December, January, and February 2011 with faculty at the Knight Cancer Center at the Oregon Health and Sciences University. Work by Druker and others [13, 14] documents how this approach to cancer treatment results in positive outcomes for chonic myeloid leukemia.

cycle may begin again.

Provenance can be useful in several ways in this setting. For example, clinicians often consult with each other on therapy selections. During such consultations, clinicians typically review data and resulting decisions at each step in the process. If the therapy process for a patient has run through multiple iterations, data and its manipulation history from all iterations may be examined. Thus the full history of a therapy selection (which database data or external data sources it derived from, what manipulations it was subject to, and what queries it resulted from) must be easily available. We note that in this setting, users freely mix definition, manipulation and queries as part of typical work. We also note that users in this setting are not data management experts. When these users query provenance, they need to be able to do so without significant query-writing expertise or time investment.

As another example of provenance in this setting, input data to the therapy decision process such as the reference databases described above are subject to frequent updates. Changes to these data may in turn affect "downstream" data such as therapy choices. Provenance provides the means for detecting when downstream data is no longer valid. A typical mechanism for performing this detection is for the software system or a user to write a query that asks, "What data in the database was derived from the source that was updated?" This is one example of queries in this setting that identify data by describing some of its provenance characteristics without knowing its full history.

In this setting, provenance of the therapy selection process is part of the medical record. This record is expected by users to be immutable. We also note that users prefer to keep provenance "out of sight" during normal data manipulation and query, only exposing it when needed for provenance-related queries.

1.1.2 Corporate Budget Planning

Corporate budget forecasting is an iterative process in which a manager inserts budget requests from subordinate managers or employees into a spreadsheet or other tabular data management tool; manipulates request data to reflect personal judgement; generates reports based on the data in formats that financial analysts and senior managers request; and iterates during several rounds of budget negotiations[3]. This process may take weeks or months to complete. Initial collection of budget requests is typically done in a spreadsheet application, using insert, update, and delete operations. Each request is typically recorded and annotated with information about where the request came from and which project it pertains to. The next step typically involves decreasing, increasing, or deleting certain requests in order to meet a budget target and a set of deliverables for the budget period. During this step, a manager may make several rounds of adjustments, resulting in a history that may later need to be reviewed. In the next step, the manager may combine data from several spreadsheets, each of which represents the budget proposal for an individual project or department for which the manager is responsible (this aggregation corresponds to query operations with materialized results). The result of this aggregation is one or more budget summary tables. In a relational database, this step would correspond to one or more query operations. Next, the manager may again manipulate data in order to balance the budget across all these projects or departments. This corresponds to further data manipulation of the summary tables. Several "give-and-take" iterations may follow, as managers negotiate with senior managers and each other. This process is typically repeated at several levels within an organization.

Corporate managers often comment that they need a "paper trail" in order to remember the many changes they typically make during a budgeting process. While

---

[3]The budget forecast setting is representative of my personal experience and that of others over a period of 12 years of direct participation as a manager responsible for forecasting budgets for medium to large engineering organizations.

some managers keep detailed records, these records are typically not stored with budget data and require considerable extra effort to maintain. Because the amount of data and number of iterations encountered in the budget process is typically beyond a user's ability to visualize or recall, users in this setting would benefit from tools to automatically record and to query the provenance of budget data. Provenance is beneficial in this setting, because managers may need to review where a request came from, when it was added, or who submitted it, in order to ensure that entries in the forecast table are complete; review justification of budget requests, or judge the impact of not including items in the budget; and use information on how requests were arrived at in order to justify proposed expenses to upper management.

There are several similarities between the therapy prioritization setting discussed above and the corporate budgeting setting. As noted above, the budget forecast setting freely mixes definition, manipulation and queries over data as part of typical work. Users in the budget forecast setting need to query provenance using available tools, without having significant query-writing expertise. Because budget forecast data undergoes typically many manipulations and queries in each iteration of the budgeting cycle, and because there may be many such cycles during a single budget process, users query multiple generations of provenance in order to understand how data came to be. Typical provenance-related queries must identify data by describing partial characteristics of data provenance, for example the external sources where data came from, or specific dates at which data was modified, without knowing data's complete history. Another similarity of these settings is that users in both expect provenance to be protected from change.

One difference between the budget forecasting process and the therapy prioritization process is that in the budget process, users may delete data (for example, if a decision is made not to fund a budget request) and then encounter the same data again (for example, if the same request is made redundantly). When this happens,

the user wants to know that the newly encountered data has been seen (and deleted) before. This suggests that provenance tools should retain deleted data and its provenance so that users can distinguish re-discovery of already-deleted items from new discovery of hitherto unseen data.

1.1.3 Battlefield Information Management

The task of gathering information, assessing its accuracy, and using it to produce reports is commonplace in theaters of military operations. A battlefield information officer routinely gathers and organizes information into a database from a variety of external data sources to help her commanders make decisions. A typical task might be to assemble a table of casualty information due to explosive device incidents during the prior week in a given patrol area [4]. Data sources for assembling this report may include military incident databases or reports from friendly forces in the area, e-mail ex-changes with local police, patrol logs for the week, and medical-team records. Personal knowledge of the operations area, recent events, and reliability of sources also play a role in assembling such reports. A task of this type might begin by writing a query against an existing database, to select known incidents during the time period of interest. Next, the information officer might select data from external sources and insert it into rows and columns in the evolving data table, adding columns to represent new data as needed. The officer may merge rows from the table that represent the same incident, and combine information from multiple columns that are found to be redundant, by writing queries. Thus the task of assembling a table of information is iterative, involving data definition, manipulation, and query operations.

Provenance could be useful in this setting. For example, a commanding officer reviewing the summary report might ask, "Where did we get the date for incident

---

[4]The battlefield information setting described here was the topic of an extended discussion with DARPA representatives at a workshop in March, 2008.

105?" – a question that requires knowledge of the provenance of the item labeled 105 to answer. As another example, the reliability of information sources in a theater of operations is subject to change. Each time this happens, the reliability of "downstream" data (data derived from the source, and possibly manipulated and queried in the interim) may be affected. As in the medical therapy selection setting, provenance provides the means for detecting when downstream data may no longer be viable.

We note that the battlefield information setting has similar characteristics to the two settings described above. Sequences of mixed data definition, manipulation, and query are common; users need to query provenance easily, without significant data management expertise; provenance queries typically involve examining several generations of operations; users may need to identify data by describing part of its provenance, without knowing its full history; and users expect provenance to be protected against unexpected change.

1.1.4 Opportunities to Enhance Provenance Models

The settings described above suggests several capabilities related to provenance that do not typically exist in current provenance models:

A provenance model should allow for intermixing manipulation and query operations. Each setting described above typically involves multiple iterations of manipulating data, querying it to produce new data, and then further manipulating the new data. One shortcoming of current provenance models is that they do not typically represent provenance due to such sequences of queries and manipulations. Some current models address provenance due to only manipulation operations. In these models, there is no provision for introducing new relations, e.g., as the result of queries. Other current models address provenance only for results of individual queries (single-generation provenance), without addressing materialization and subsequent manipulation of query results for input to later manipulations.

Provenance representations should be parse-able using available languages. Users in these settings need to express queries that select data by provenance characteristics, for example to distinguish data by its source or by when and how it was manipulated. Provenance relationships among data are naturally represented in graphs, where nodes represent data items and edges represent parent-child relationships between data items. To fit provenance into the schema of relational data, current models express provenance using symbolic expressions that fit available data types. Use of algebraic expressions stored as character strings is a common approach. When a user issues a query that selects data by its provenance characteristics, each provenance expression (representing a parent-child relationship) must first be parsed and interpreted so that it can be compared to the desired provenance pattern described by the user. Because current query languages do not provide operators for this parsing and interpretation, queries that select data based on immediate parent-child provenance cannot easily be constructed. As a result, provenance interpretation in current models is typically done manually.

Multi-generation provenance should be easy to access. Data manipulation and query in the settings we describe are iterative processes. In each case, data typically are subject to, and result from, multiple generations of manipulations and queries. In order to understand how data came to be, users may need to understand more than one generation of these operations. However, current provenance models store with each data item only the single provenance expression that ties that item to its immediate parents. Thus the whole history of a data item, which may consist of many such parent-child generations, is distributed across the database. Unfortunately, current provenance models provide no language to easily re-assemble these generations so that the entire history of a data item may be queried. As a result, the burden is on users to reconstruct multi-generation provenance before querying it. Typically reconstruction is done, if at all, by writing recursive queries or programs that traverse

all relevant parent-child (single-generation) relationships.

A query language should be available to make expressing queries over provenance easy. Users in our settings may be familiar with using relational databases, but are not typically experts in writing complex queries. Current provenance models support provenance queries using languages designed for querying data: Datalog, for some models, and SQL or SQL-like languages for others. Queries that retrieve provenance of data for manual inspection by the user, for example, "What are the ancestors of this data?", are relatively simple to write in these languages, and have been addressed in previous work. However, users in our settings use provenance characteristics as a means to select data for further processing. In order to express such queries, a user must describe characteristics of provenance that describe the data of interest.

Describing characteristics of provenance is typically difficult in query languages used in relational databases. Unlike traditional atomic data types used in relational databases, provenance is a chain of relationships. The length of the relationship chain (that is, the number of ancestors and derivation actions) is a function of the derivation process that gave rise to the data. Thus interrogation of provenance data may require examination of multiple ancestors, the number of which may not be known by the user at the time a query is written. In addition, the provenance of each ancestor element may include multiple properties, such as the relation in which the element resides, its value, and so on. However, query languages such as SQL are designed to interrogate a fixed set of elements, where attribute values defined by a relation schema and the number of tables involved in the query must be known in advance. Thus writing queries that select data by describing patterns present in its provenance is typically difficult in these traditional query languages.

Provenance should be treated differently from data with regard to protection and management. Users in our settings typically expect that although data may change,

the history of data is immutable once created. However, current provenance models use manipulation and query operators that do not distinguish schema attributes. Thus any query or manipulation may affect any data (including provenance information) present in the database. This behavior is incompatible with a "write-once" approach to recording provenance.

The five opportunities described above motivate our conceptual model. In the next section, we examine how well current provenance models in the literature address these opportunities.

1.2 Where Current Provenance Models Fall Short for Our Settings

Cui and Widom's *Lineage* model [12] annotates each relational tuple from a query result with an expression representing the set of tuples from input relations that cause the result tuple to appear. These expressions are computed *lazily*, after query execution, if a user demands provenance information for selected data. The lack of insertion operators (Lineage only addresses provenance induced by queries, not DML operations) precludes representation of external sources, data manipulations, and multiple insertions of data. Though Lineage identifies ancestor tuples in its provenance expressions, it does not provide information about derivation actions or agents. Lineage does nothing to prevent direct user manipulation of provenance information. In fact, it requires user-initiated actions to create provenance information. Lineage includes no language for querying multi-generation provenance, and does not retain or track provenance of deleted data.

In Buneman's Copy-Paste Database (CPDB) [5], provenance is recorded in an auxiliary relation. In contrast to Lineage, CPDB addresses data manipulation operations, but does not support queries (relational algebra operators). Even though insertion operators are supported, CPDB does not address multiple insertions of identical data (nor tracking of multiple histories, because these do not occur in the

CPDB model). Like Lineage, CPDB does nothing to prevent direct user manipulation of provenance information. CPDB also includes no language for querying multi-generation provenance, nor does it retain or track provenance of deleted data. In work subsequent to CPDB [7], Buneman developed a framework for recording provenance due to queries as well as data manipulations in a single model. This model retained the other characteristics and shortcomings of CPDB that we discuss in this comparison.

Trio, developed at Stanford University, supports both data uncertainty and provenance [1]. We restrict our consideration to data operations without uncertainty. Like Lineage and CPDB, Trio supports relational data, and stores provenance in the form of annotations to tuples. Like Lineage, this provenance includes where data came from, but not which manipulations were done, nor who performed them. Trio's language supports queries as well as data manipulation, but Trio cannot represent provenance due to a mix of these operations. Trio is the only current model that retains deleted data. Trio is the only current model that provides a provenance-specific built-in function, *Lineage()*, to help users in writing provenance-related queries. We examine the utility of this function in Chapter 4 when we address syntactic complexity of provenance queries.

Orchestra [21] is a collaborative data-sharing system, motivated by the need to share scientific databases between research groups. The goal of Orchestra is to provide *update-exchange* of data, where sites publish updates to their data at intervals of their choosing, and adopt published updates from others at intervals of their choosing. Rules (views) established at each site integrate incoming updates to produce potential revisions to a local database. Other rules enforce *trust policies* that allow local administrators to select which revisions are integrated. To enable trust policies, update tuples are annotated with their provenance in a way similar to that used in Trio and Lineage.

Orchestra's provenance representation differs from these other models in its ability to express more fully *how* data was derived from ancestor data. Figure 1.1 shows an example of a simple query and how the provenance of its result tuples is represented in each of these models. The query self-joins relation R on attribute A, and unions this result with the result of a self-join of R on attribute C, and then retains only A and C in the result relation. The first tuple in the query result exists because input tuple a combined with itself twice in the execution of the query, giving rise to the first result tuple each time. The first tuple in the result also exists because tuples a and c in the input relation combined to give rise to it. The Orchestra provenance model shows this provenance as a polynomial. In it, the multiplication operator indicates that the combined presence of input tuples gives rise to an output, and the addition operator indicates that each of its input tuples gives rise independently to a result. In contrast, the Trio representation makes it impossible to distinguish how many instances of an input were present to give rise to an output. The CPDB model makes it impossible to distinguish how many independent, identical terms give rise to a result. The Lineage model is the least expressive, indicating only which inputs had some bearing on a result. Green [20] has formally shown that the Orchestra model is the most expressive of these provenance models.

Like Trio and Lineage, Orchestra does not record the users or derivation processes involved. Though more expressive about how ancestors combine to yield resulting data than Lineage or Trio, this expressiveness is limited to logical expressions using "and" and "or", rather than details of which operations were performed. In Orchestra, there is no concept of derivations that include multiple operations applied over time. Thus multiple insertions of identical data are not part of the Orchestra model, and there is no notion of multi-generation provenance in Orchestra. In addition, Orchestra represents only provenance due to queries, not manipulation operations. Because Orchestra is the most expressive of the models discussed here, we

Input relation R

A	B	C
1	5	8
3	2	9
1	6	9

$$\pi_{AC}(R \bowtie_A \rho_{B \to D, C \to E}(R)) \cup (R \bowtie_C \rho_{A \to D, B \to E}(R))$$

Query Result

Provenance Representations

A	C	Lineage	CPDB	Trio	Orchestra
1	8	{a,c}	{{a},{a,c}}	$2a + ac$	$2a^2 + ac$
1	9	{a,b,c}	{{c},{a,c},{b,c}}	$2c + ac + bc$	$2c^2 + ac + bc$
3	9	{b,c}	{{b},{b,c}}	$2b + bc$	$2b^2 + bc$

Figure 1.1: Examples of Provenance Representations in Current Models

use its representation as the standard for comparing the expressiveness of provenance representations we develop in this work.

Figure 1.2 summarizes how current provenance models in the literature address the opportunities identified in our settings. Note that all models we consider provide automatic provenance collection. However, only CPDB and Trio can represent external sources as part of provenance and track operations done and users that perform them. Only CPDB addresses intermixing of queries and manipulations in multi-generation provenance. No models in the literature address the other four opportunities we describe.

We decided to make this set of opportunities the focus for our conceptual model. We also chose to address the other needs identified in the figure. All but one of these opportunities and needs are fundamental to the notion of provenance. The remaining characteristic, allowing for multiple insertion (or copy-and-paste) of identical data, is included because it offers an important opportunity. In current provenance models,

Provenance Model	Represents external sources	Automatic Provenance Collection	Multiple insertion of same data	Records operations and users	Opportunity 1: Provenance due to mix of query, definition and manipulation operators	Opportunity 2: manual parsing of provenance representations	Opportunity 3: manual assembly of multi-generation provenance	Opportunity 4: Simple language for selecting data by its multi-generation provenance	Opportunity 5: managing provenance orthogonally to data	Retaining deleted data and its provenance
Lineage		Yes								
CPDB	Yes	Yes		Yes						
CPDB Extended	Yes	Yes		Yes	Query + DML					
Trio	Yes	Yes								Yes
Orchestra		Yes								

Figure 1.2: Evaluating Current Provenance Models. Blanks indicate that a model does not address an identified issue.

provenance of data introduced by definition language operations or data manipulation language operations is limited to a single origin. For example, a tuple inserted into a database in a single operation comes from just one place, and cannot be later introduced from a different place. However, provenance of data introduced by queries may consist of several distinct origins. For example, a tuple in the result relation of a union operation may have two distinct origins. By including multiple insertion of data in our model, we allow for all operations to specify multiple origins for result data provenance.

1.3 Research Goals

In this work we:

- Define a set of capabilities desirable in a provenance model, based on opportunities to contribute to the provenance literature and needs identified in our

settings.

- Define a conceptual model for data and provenance, along with a language for manipulating and querying data represented in the model

- Formalize our conceptual model in order to define it clearly and prove propositions about its properties.

- Evaluate our conceptual model in terms of how well it addresses the opportunities we defined, how expressive its provenance query language is, and how the complexity of queries in our language compares to those of others in the literature.

- Define a performance benchmark suite that includes data and provenance, and define a workload to measure performance of important classes of operations from these settings.

- Study the performance of our model when implemented on existing software platforms, using our benchmark suite.

- Define a logical model that faithfully supports our conceptual model.

We set the scope for our work as follows. We focus on relational data in order to make our contributions comparable to existing literature and because many users are used to dealing with data in tabular form. (Using our provenance model with other kinds of data is also possible. We suggest this area for future work.) We address typical operations found in relational database languages, as well as selected operations found in the settings we defined above. These setting are examples of *data curation* settings. Data curation settings are characterized by continuous integration, maintenance, and update of datasets by domain experts over relatively long periods of time. Data curation settings represent a popular emerging discipline in data management. We choose to address selected operations characteristic of data curation

settings because of this growing popularity, because inclusion of them in our model does not unduly complicate our model structure or query language, and because these selected operations are representative of the broader set of operations in data curation settings. In this work, we distinguish individual derivation steps comprised of single operations (which we call the *single-generation* provenance of the result data) from the composition of these steps (which we call the *multi-generation* provenance of the end result). We note that, while single-generation provenance is informative, it is analogous to looking at the parents of a person in a family tree document, but ignoring their grandparents and more distant ancestors. Often, users may need to see the entire family tree in order to decide trustworthiness of data or understand its applicability. We note that features of our proposed model that are defined in order to support specific use models may be omitted without compromising the core contributions of our work.

The remainder of this dissertation is organized as follows. In Chapter 2, we informally define our conceptual model for data and provenance, and discuss the motivation for each aspect of the model. In Chapter 3, we formally define our model. In Chapter 4, we evaluate our model. Chapter 5 summarizes the benchmarks we defined, the implementations we studied, and the results of our performance studies. In Chapter 6, we show how implementations we studied can comprise a logical model that supports our conceptual model. In Chapter 7, we survey related work reported in the literature. In Chapter 8, we offer conclusions and suggest future extensions to this work.

Chapter 2

Conceptual Model Overview

In this chapter, we introduce a conceptual model for data and provenance. We call our model the Multi-granularity, Multi-provenance (MMP) model. We first outline goals for MMP, based on opportunities identified in Chapter 1. We then informally define MMP.

The fundamental structure of MMP is motivated by the differences between provenance and data. First, provenance and data differ in that provenance is created as a side effect of operations applied to data, while data is directly manipulatable by those operations. Second, once created, provenance is invariant, while data is not. Third, provenance is a temporal relationship between entities, while data in many common data models exists only "now". The first two of these differences echo Gap 5 outlined in Chapter 1, and suggest that, in MMP, 1) data and its provenance should be represented using distinct models, and 2) access to data and provenance should be controlled independently. The third of these differences suggests that 3) provenance should be modeled as a network interconnecting data from various instants in time, while 4) data and its relationships should be modeled at single instants. A naive approach might suggest that data and provenance could be modeled completely independently. However, users need to query data and its provenance in combination, using the same language, and users apply operations to data that affect provenance as well. This need suggests that in MMP, 5) although data and provenance are largely

independent, a model supporting both must allow them to be queried simultaneously, and 6) the language for MMP must define both how operations define, manipulate, and query data, and how they induce provenance.

Other goals for MMP are motivated by gaps in the literature that make current provenance models difficult to use, as discussed in Chapter 1. Gap 2 suggests that in MMP 7) users should not need to interpret and parse provenance representations in order to reconstruct provenance relationships. Gap 3 suggests that 8) users should not need to write queries to reconstruct, nor otherwise manually reconstruct, multi-generational provenance from single-generation provenance relationships. Gap 4 suggests that 9) users should be able to phrase queries that select data by its provenance (where it came from, how and when it was manipulated or queried, and who performed the manipulations) using a simple query language, something not possible in current models. In Figure 1.2, the user need "Records operations and users" suggests the part of this goal regarding what selection criteria should be available.

Although the fundamental structure of MMP that we describe in the next section is data model agnostic, the remainder of our work focuses on the use of MMP with the relational model for data because relational data management tools are in common use. This specialization suggests that MMP for the relational model should 10) support multiple relations simultaneously. Gap 1 from Chapter 1 suggests that, along with multiple relations in a database, MMP should 11) support a query, data management (DML), and data definition language (DDL) with features common to relational databases. The inclusion of data definition operators, which can affect entire relations or schema attributes, along with the presence of data management operators, which can affect tuples or individual attribute values, implies another goal: 12) MMP needs to address provenance of all granularities of relational data, including relation schema.

In current literature, Buneman defines the only other provenance model we know

of that supports more than one data granularity [8]. Buneman's model, CPDB (which stands for Copy-Paste Database), models data as tree-structured rather than relational, and makes the assumption that all data affected by an operation has the same provenance. For example, if a sub-tree of data is inserted into a CPDB instance, all data elements in that subtree gain a provenance record indicating they were inserted from the same source. This assumption is valid for data manipulation operators supported in CPDB (insert, copy-paste, update, and delete). However, our model supports query operators, which do not induce the same provenance for data at all affected granularities. For example, a tuple resulting from a Cartesian product operation has a parent tuple in each input relation to the product, yet each attribute value in such a tuple has provenance of only one input attribute value from one of the tuples. Thus 13) MMP must allow for distinct provenance of data at different granularities of a query result, as well as distinct provenance for different query result data at the same level of granularity.

Evaluation of conceptual models is typically difficult because there is no straightforward way to measure correctness or completeness. One way we choose to evaluate MMP is by determining whether the opportunities and user needs identified in Chapter 1 are met by MMP. We define the following additional goals to support specific use models. 14) MMP should be able to represent as part of data provenance that data may be inserted into an MMP instance from specified external sources as well as be manipulated within the model instance. This requirement follows from the "Represents external sources" need shown in Figure 1.2. 15) MMP should enable users to insert the same data values multiple times, for example because the data may be encountered from multiple external sources, or from the same external source more than once. This requirement follows from the "Multiple insertion of same data" need shown in Figure 1.2. If data has been deleted, curators may wish to know that the data was once present. This need is identified in Figure 1.2 as "Retaining deleted

data and its provenance". For example, if a tuple was inserted, then deleted because it was thought erroneous, a curator finding the same source of data at a later time may find it useful to know that the data was previously encountered and then deleted. Because of this use model, 16) MMP should retain deleted data without changing the semantics of the relational languages it supports.

Finally, recall that in Chapter 1, we identified one other need, which we repeat here as a goal: 17) In addition to the ability to query provenance, MMP should provide the capability to visualize the multi-generation provenance of selected data as a means of browsing the history of data.

The goals for defined above for MMP, and two others defined in the next section, are listed in Table 2.1.

2.1   Model Fundamentals

The fundamental notion of MMP is that its data and provenance models are *orthogonal*. Data (entities and relationships) are modeled at instants in time, while the provenance relationships between data at all modeled granularities interconnect these instants. Because part of data's provenance is the prior data that gave rise to it, all such instants are retained[1]: whenever an operation is applied to data, a new database is created, instead of the existing database being modified. We call each snapshot a data *face*. Each face in an MMP instance instantiates the same data model, because the set of faces represent the same database recorded at different instants in time. The data model may be any appropriate for the application at hand: a relational database, an entity-relationship model, a graph database, or an RDF representation, for example (though in this work we constrain MMP to the relational model for data). Note that the schema may change from one face to the next, for example if a new attribute

---

[1]In an implemented system, for practical reasons, a system administrator or user would likely be able to delete instants that were deemed no longer useful. We do not consider such a capability in our definition of MMP here.

Table 2.1: Goals for the MMP Conceptual Model

1. Data and its provenance should be represented using distinct models
2. Access to data and provenance should be controlled independently
3. Provenance should be modeled as a network interconnecting data from various instants in time
4. Data and its relationships should be modeled at single instants
5. MMP must allow data and its provenance to be queried simultaneously
6. The language of MMP must define both how operations define, manipulate, and query data, and how they induce provenance
7. Users should not need to interpret or parse provenance representations in order to reconstruct and query provenance relationships
8. Users should not need to reconstruct multi-generation provenance in order to query it
9. Users should be able to phrase queries that select data by its provenance using a simple query language
10. MMP should support multiple relations simultaneously
11. MMP should support query, DML, and DDL languages with typical DBMS features
12. MMP should support provenance for all granularities of relational data
13. MMP must allow for distinct provenance of data at different granularities of a query result, as well as distinct provenance for different query result data at the same level of granularity
14. MMP should be able to represent as part of data provenance that data may be inserted into an MMP instance from specified external sources as well as be manipulated within the model instance
15. MMP should enable users to insert the same data multiple times
16. MMP should retain deleted data without changing the semantics of the relational languages it supports
17. MMP should provide visualization capability for multi-generation provenance
18. MMP should avoid representing provenance redundantly
19. MMP should rely on implicit, rather than explicit, representation of provenance where possible

is added to a relation. Each face is labeled with the time of its creation, the operation that induced it, and the agent that applied the operation, in order to distinguish the event that induced it.

Faces in an MMP instance form a set totally ordered by creation time. Because the semantics of data models we consider define results of operations to be derived from the current state of the database when the operation is applied, or from specific sources of data external to the database, data in one face is derived only from data in the immediately preceding face or from these external sources, using the operation labeling that face. Provenance of each data element in a face is modeled as directed edges that originate at the element and terminate at all elements in the immediately preceding face, or at referents to external sources, that gave rise to it. These edges are induced by the operation that created the new face. Once created, these edges are retained and are not manipulatable, because they represent the effects of actions that are part of data history. We call these edges *provenance links*. Figure 2.1 shows an example of data faces and provenance links for an MMP instance. Provenance links are shown as directed edges from data elements resulting from operations to the input data that gave rise to those elements. Data elements are shown as circles on rectangular data faces. Note that data appears on all successive faces once it has been created (even if it has been deleted), while new data is added when created by applied operations. In MMP, there are implicit provenance links from a data element in one face to the same data element in the immediately preceding face. Figure 1 does not show the implicit links. Provenance links encode the complete provenance of data elements. Because data is represented in faces at all modeled granularities, provenance links can originate and terminate at data elements of any granularity. The pattern of provenance links is determined by a provenance model that is in turn defined by the language used to manipulate and query data.

Provenance links may be inspected (either visually, or using a query language)
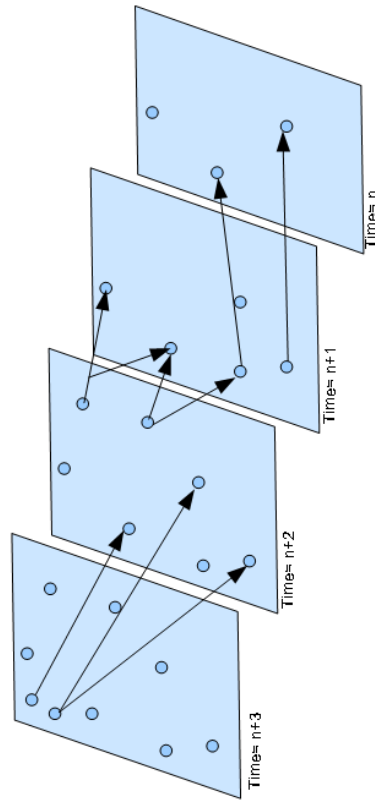
Figure 2.1: Faces and Provenance Links in an MMP instance

in order to understand the provenance of a data element. Some applications using MMP may inspect only *single-generation* provenance, (i.e., connections between one face and its immediate predecessor). Other applications may inspect the graph of provenance links stretching back to distant faces, (i.e., *multi-generation* provenance).

Multi-granularity, multi-generation provenance in MMP can result in a large and rapidly growing population of provenance links that may confuse users and cause inefficient implementations. This issue motivates two additional goals for MMP: 18) MMP should avoid representing provenance redundantly, and 19) should rely on implicit representation of provenance where possible. We expect that the explicit representation of all faces in an MMP instance is also unrealistic for any implementation. However, we believe that the use of complete faces at each step in MMP makes

the model easy to understand and visualize. Chapter 5 is devoted to the investigation of ways to implement MMP at the logical level, including removing the requirement to store complete faces at each step.

This basic structure of MMP is motivated by Goal 1 (distinct models for data and provenance, suited to the application). Goal 2 motivates the use of distinct, independent models for data and provenance. This distinction facilitates separate control of data and provenance content. Goal 3 motivates MMP provenance links and their use in interconnecting components on different faces. Goal 4 motivates the face structure of MMP.

2.2 Structure of a Relational MMP Data Face

For the rest of the work in this dissertation, we specialize MMP to use the relational data model, a decision motivated by Goal 10 described in Section 2.1. For each face, we refer to the relations, their attributes, tuples, and attribute values as the *components* of the face. Figure 2.2 shows an example of an MMP relational face. The two relations, four attributes, three tuples, and six attribute values in the figure are the components of the face shown. Figure 2.3 shows a set of faces, each labeled by the operation that induced it.

In support of Goal 16, MMP provides a function that, when applied to a data element, indicates whether that element is still active, or has been deleted. Goal 16 also motivates the decision that deleted data in MMP is not manipulatable, nor does it take part in queries.

2.2.1 External Sources of Data

Data that is directly inserted into an MMP instance from outside the database has provenance in the form of the name of the external information source from which the data came. In MMP we model these external sources with elements called *external*
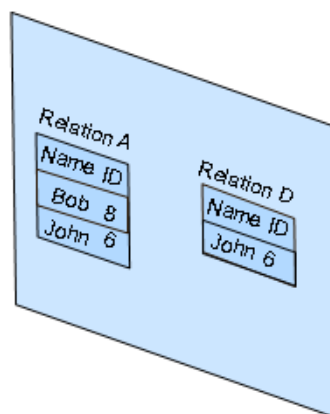
Figure 2.2: Relational Data Face in MMP

source referents. These elements exist separately from data faces. External source referents must be explicitly created in an MMP instance before use as a source of data. Upon creation, each external source referent consists of the identifier of the external source that it represents. Once created, external source referents may not be deleted. The inclusion in MMP of external source referents is motivated by Goal 14.

2.3 Structure of an Example MMP Provenance Model

As defined in Section 2.2, provenance links show the derivation relationships between data in one face and its immediate ancestors. When data in a face, or external source referents, are used as inputs to an operation, we refer to them as *parents*. We refer to the result data of the operation as *children*. Provenance links originate at a child component and terminate at its parents. Figure 2.4 extends Figure 2.3 by adding a set of external source referents and the provenance links induced by the operations that label each face in the figure.

Provenance recording is automatic in MMP. Users do not record provenance as each operation is performed, nor can they manipulate provenance. Provenance captured in MMP is determined by the semantics of operators that manipulate data on
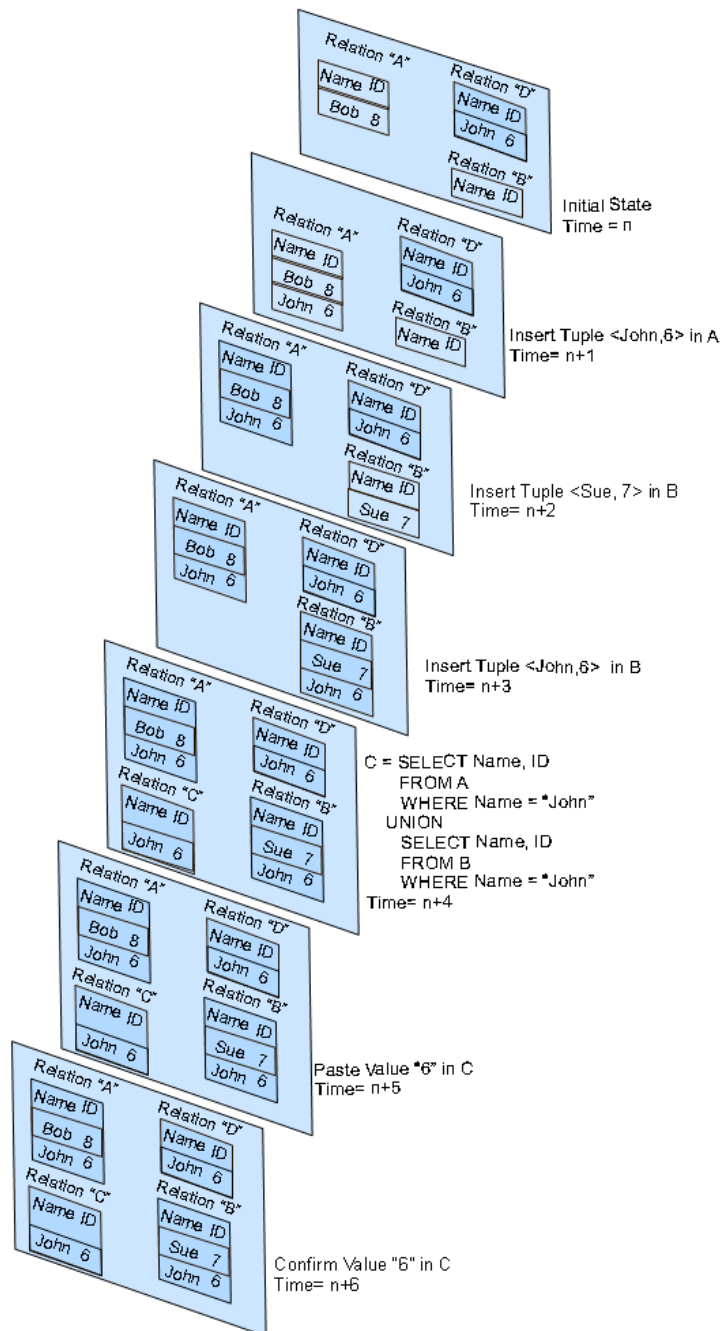
Figure 2.3: Example of Multiple Relational Faces in an MMP instance
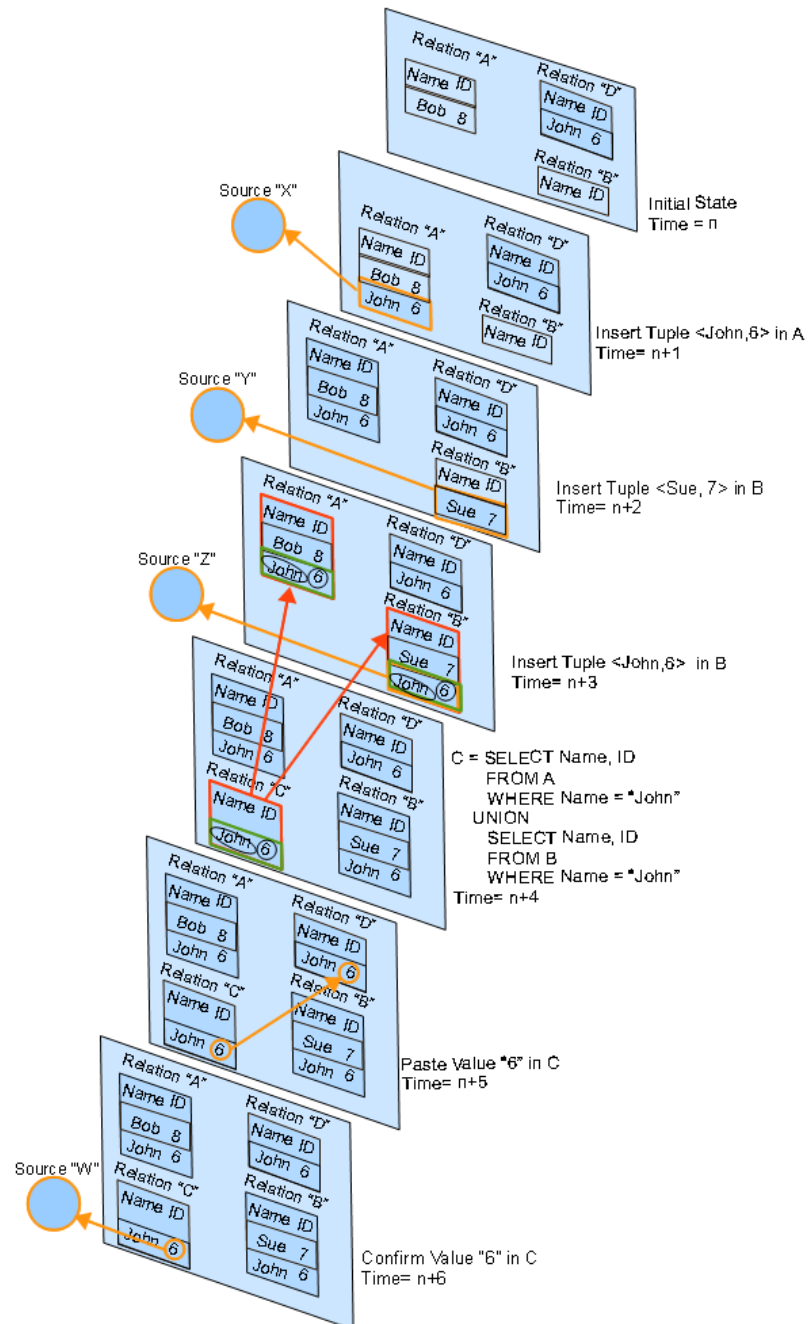
Figure 2.4: MMP Instance showing provenance links between components

the front (i.e., most recently created) face in the MMP language. The effect of operator input data on result data is operator-specific, so the provenance links induced by each operator are also operator-specific. In MMP, a result component may originate zero or more provenance links. Provenance links are hyper-edges that always have a single originating component, but may have more than one terminal component. Constants introduced by queries originate zero provenance links because there are no parents from which they derive in the preceding face. An external source referent originates no links, because operations do not affect external sources referents. An external source referent may terminate any number of links, because it may be a parent in any number of operations. A component may also terminate multiple provenance links, because (in query operations in the relational model) a given parent component may contribute to multiple child components. For example, a tuple from one input relation to a Join operation may combine with several distinct tuples in another input relation, giving rise to multiple result tuples. Each terminal of a provenance link is either of the same component type (relation, tuple, attribute, or attribute value) as its origin, or is an external source referent. Provenance links may represent the provenance of components at all granularities (relations, tuples, attributes, and attribute values), in support of Goal 12, depending on the operation.

Because some queries may produce the same child component in multiple independent ways, more than one provenance link may originate from a child component. Figure 2.5 shows an example of three input relations, (A, B, and C) at an initial point in time n. At time $n + 1$, a query $R = (\pi_{alpha}(A \bowtie_{beta} B) \cup C)$ is applied, resulting in the addition of relation R to the database in face $n+1$. Provenance links due to the query are shown at all granularities: provenance of result relation R is shown with solid lines; provenance of result attribute $alpha$ is shown dashed, in red; provenance of result tuples are shown in yellow; and provenance of result attribute values are shown in green. This example query joins relations A and B using $beta$ as the sole

join attribute, and forms the union of the join result and relation C. We read the provenance of each result component as follows:

- relation R exists because both input relations A and B exist, and exists independently because input relation C exists

- attribute $alpha$ in result relation R exists because attribute $alpha$ in input relation A exists, and exists independently because attribute $alpha$ in input relation C exists

- tuple $\langle a \rangle$ in the result relation R exists for three independent reasons: because tuple $\langle a, b \rangle$ in relation A exists and tuple $\langle b, c \rangle$ in relation B exists; because tuple $\langle a, s \rangle$ in relation A exists and tuple $\langle s, u \rangle$ in relation B exists; and because tuple $\langle a \rangle$ in relation C exists

- tuple $\langle o \rangle$ in R exists because tuple $\langle o \rangle$ in relation C exists

- attribute value a in column $alpha$ of tuple $\langle a \rangle$ in R exists independently for three reasons: because the attribute value a in column $alpha$ of tuple $\langle a, b \rangle$ in relation A exists; because the attribute value a in column $alpha$ of tuple $\langle a, s \rangle$ in relation A exists; and because the attribute value a in column $alpha$ of tuple $\langle a \rangle$ in relation C exists

- attribute value o in column $alpha$ of tuple $\langle o \rangle$ in R exists because the attribute value o in column $alpha$ of tuple $\langle o \rangle$ in relation C exists.

2.3.1 Continuity of Existing Data

All data in a face except new data introduced by queries, newly inserted from external sources, or copied and pasted from elsewhere, is present because it existed in the previous face. In MMP, there is an implicit provenance relationship between such
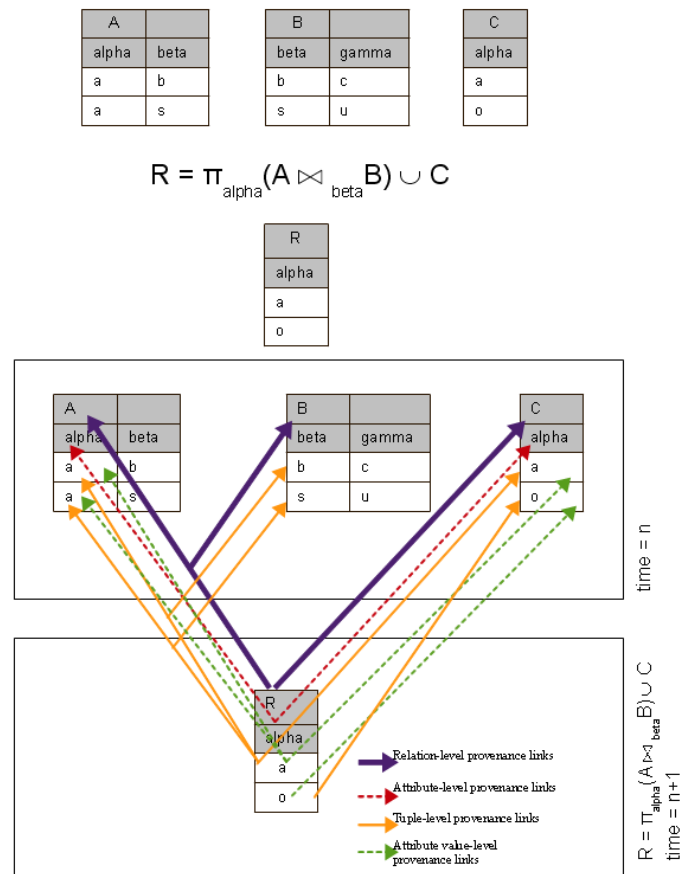
Figure 2.5: Example model instance showing provenance links

data in one face and the same data in the prior face. Because these relationships are easily discoverable by observation, Goal 19 motivates us not to include explicit provenance links for them. For example, in Figure 2.4, note that there are no explicit provenance links from face $n + 5$ to face $n + 4$, even though every data item in face $n + 5$ clearly exists because its predecessor exists in $n + 4$.

2.3.2 Granularity and Inheritance of Provenance

As shown in Figure 2.5, MMP operations may induce provenance at different component granularities. This functionality is motivated by Goal 13 from Section 2.1.
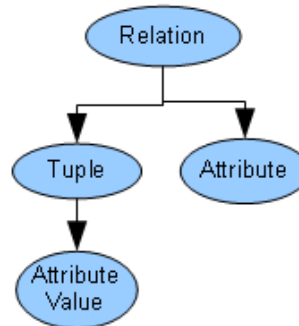
Figure 2.6: Granularity Hierarchy in Relational Data

However, explicit inclusion of provenance links at all granularities would be redundant because, for certain operations, some or all of the provenance links for lower-level components are derivable from those recorded for higher-level components. For example in Figure 2.5, MMP records only the provenance links shown as solid lines. When provenance at other granularities is needed for querying or browsing, it is reconstructed using the recorded provenance and a set of *inheritance rules* that we define in Chapter 3. We say that operations where inclusion of provenance links at lower levels is not explicit because they would be redundant to those at upper levels have *inherited* provenance. Motivated by Goal 19, we record only the provenance link at the highest possible level of granularity, using the hierarchy shown in Figure 2.6. We believe this approach will be easy for end users to understand, with provenance links explicit only where necessary.

To further clarify the mechanisms for inherited provenance, Figure 2.7 shows two independent examples of operations with inherited provenance. On the left, a tuple in relation A was inserted at time $n + 1$ from external source X via an Insert Tuple operation. The provenance link from the result tuple to the external source is explicitly recorded in the model. The provenance links for the attribute values in the tuple, shown in dotted lines, are not recorded. Instead, they are inherited from
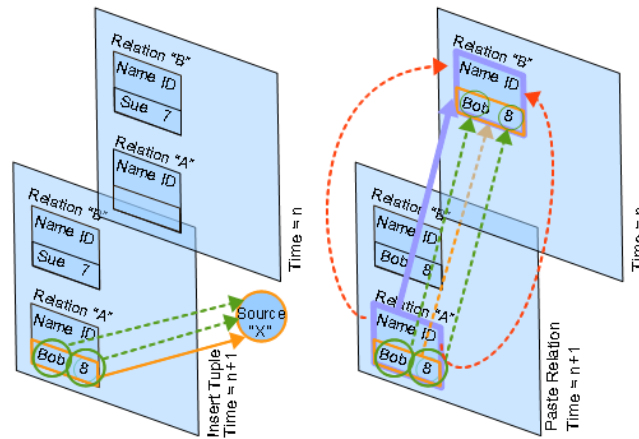
Figure 2.7: Examples of Inherited Provenance

the tuple provenance link. On the right in the figure, relation A is pasted into the database at time $n + 1$ as a copy of relation B. The provenance link (shown solid) from the relation A to relation B is explicitly recorded. The provenance links (shown dotted) for attributes, tuples, and attribute values, are not recorded but are inherited in MMP.

2.4 Interacting with and Visualizing MMP

MMP provides two ways to interact with data and provenance. The MMP language provides operations to define and manipulate data, and to query provenance and data. MMP also defines graphical visualizations of provenance relationships between data that we call *provenance graphs*.

2.4.1 The MMP Language

Goals 6 and 11 motivate us to provide a language for interaction with the most recent face in an MMP instance that includes a *data definition language* (DDL), a *data manipulation language* (DML), and a *query language*. These languages include the usual DDL, DML, and query operations offered in a typical relational DBMS.

DDL operations create relational structure; DML operations insert and delete data[2]. Queries materialize new relations as query results from existing data. We omit some language features of relational query languages, for example aggregate functions, to limit the scope of this work. We also add new operations, such as copy-and-paste[3], when the need for them arises frequently in our motivating scenarios.

MMP also provides additional functionality for some DML operations. For example, MMP allows *multiple insertion* of data. That is, if some component already exists in the database and a duplicate component is inserted or pasted, the operation succeeds, whereas in a traditional database instance it would not. This feature is motivated by Goal 15 in Section 2.1, and enables documenting *plurality of support* for data: a measure of how many independent sources give rise to a data element. Another extension MMP offers beyond a traditional relational database is a broader variety of *granularities* for DML operations: insertion and paste of whole relations in addition to individual attribute values and tuples.

Queries in MMP may be composed as unions of Select-Project-Join terms. Unions of conjunctive queries, or SPJU queries, are definable by existential positive first-order formulae. They correspond to unions of select-from-where queries in SQL such that the where clause is composed only of conjunctions of atomic value comparisons. SPJU queries are widely considered to be the most frequently expressed queries in relational databases. MMP extends the usual predicates available in the SELECT and PROJECT operators of relational algebra with predicates that select data based on its provenance.

MMP also provides operators that allow users to express confidence or doubt in data without changing the data. These operators create a new face, but do not change any data brought forward from the previous face, and do not introduce any new data.

---

[2]We model modifications to data as deletions of existing data followed by insertions of updated data.

[3]Copy-and paste operations are used for copying data from one place and pasting it into another within a database.

These operators only create provenance artifacts, which may be queried or browsed when users inspect data provenance.

Table 1.1 shows the MMP language. In our syntax, a *value* v in the column for an attribute a of a tuple t in a relation r is addressed by a 4-tuple (r,t,a,v); an *attribute* a in relation r is addressed by a 2-tuple (r,a); a *tuple* t in r is addressed by a 2-tuple (r,t); and the *relation* r is addressed (r). In this notation, r is the name of a relation, t is a referenced tuple, a is the name of an attribute, and v is an attribute value. A parameter s denotes an external source referent. In insert operations, source referents indicate the external source from which data is inserted. In drop, confirm, and doubt operations, source referents indicate the external source that justifies the operation. Square brackets indicate optional parameters. For paste operations, x_t indicates that component x is a paste *target*, or destination, while x_s indicates that it is a paste *source*. Note that this notation is intended for use in this definition only; it may not be the syntax that the user would use.

2.4.2 Data Semantics of the MMP Language

This section describes how MMP operations affect data in a face. Many of these operations have analogs in traditional database operations (although of course these analogs do not have the side effects seen in our model, for example creation of new database faces and provenance links). We discuss how MMP operations affect provenance links in Section 2.5.

2.4.2.1 Data Definition, Manipulation, and Confidence Operations

Create Relation(r) is the analog of the SQL operation CREATE TABLE *tableName*. Create Relation succeeds if there is no undeleted relation with the specified name r already in the most recent database face. If successful, it induces a new face consisting of a copy of the most recent face, along with a new, empty relation in the new

Table 2.2: MMP Operators

Data Definition Language Operators
Create Relation(r)
Create Source($name$)
Create Attribute(r,a)
Drop Relation(r)
Drop Attribute(r,a)
Data Manipulation Language Operators
Insert Value(r,t,a,v,s)
Drop Value(r,t,a,s)
Insert Tuple(r,(a,v[,a,v...],s))
Drop Tuple(r,t,s)
Paste Value(r_t,t_t,a_t,r_s,t_s,a_s)
Paste Tuple(r_t,r_s,t_s))
Paste Relation(r_t,r_s)
Confidence Expression Operators
Confirm Value(r,t,a,v,s)
Doubt Value(r,t,a,v,s)
Query operators
Unions of Project-Select-Join subqueries

face, with the specified name. The new relation has no attributes.

Create Source($name$) succeeds if there is no external source referent with the specified name already in the model instance. If successful, it creates a new external source referent with the specified name. No analog of this operator appears in SQL.

Create Attribute(r,a) is the analog of the SQL operation ALTER TABLE *table-Name* ADD *columnName dataType*. Create Attribute succeeds if an undeleted relation with the specified name r exists in the most recent face, and if no attribute with the specified name a already exists in the schema of that relation. If successful, it induces a new face consisting of a copy of the most recent face, with the addition of a new attribute with the specified name a in the schema of the specified relation r. Upon creation, the attribute values in the new column for each existing tuple in r is set to NULL.

Drop Relation(r) is the analog of the SQL operation DROP TABLE *tableName*. Drop Relation succeeds if an undeleted relation with the specified name r exists in the most recent face. If successful, it induces a new face that consists of a copy of the most recent face, with relation r and all the components it contains marked as deleted. Note that, unlike most current relational databases, our model retains deleted data. However, deleted data does not take part in successive operations, and is normally not visible to users.

Drop Attribute(r,a) is the analog of the SQL operation ALTER TABLE *tableName* DROP COLUMN *columnName*. Drop Attribute succeeds if an undeleted relation with the specified name r exists in the most recent face, and if the relation schema contains an undeleted attribute with the specified name a. If successful, it induces a new face that consists of a copy of the most recent face, with the specified attribute and all attribute values in its column marked as deleted.

Insert Value(r,t,a,v,s) is the analog of the SQL operation INSERT INTO *tableName* (*columnName*) VALUES (*value*) WHERE (⟨match condition⟩). The SQL statement is more general than the MMP Insert Value statement in that the MMP version must specify the identity of the particular tuple into which the value is to be inserted, using the ordinal position of the tuple in a display of the relation (though specification could instead be done using values of a candidate key for the relation). Insert Value succeeds if an undeleted relation with the specified name r exists in the most recent face and contains the specified, undeleted attribute a and tuple t, if the current value of the specified attribute value is NULL or identical to the one being inserted, and if the specified external source referent s exists in the model instance. If successful, it induces a new face that consists of a copy of the most recent face, with the addition of the specified attribute value in the specified attribute of the specified tuple in the specified relation. If that value already exists, the new face's relational contents are identical to the previous face.

Drop Value(r,t,a,s) succeeds if the specified attribute value (r,t,a) exists undeleted in the most recent face, and if the specified external source referent s exists in the model instance. If successful, it induces a new face that consists of a copy of the most recent face, with the specified attribute value marked as deleted. Referent s is the external source referent that the user cites as responsible for her initiating the Drop Value action. There is no SQL analog for this granularity of data, however SQL does provide a DELETE operation for entire tuples.

Insert Tuple(r,(a,v[,a,v...],s)) is the analog for the SQL operation INSERT INTO *tableName* VALUES (*value*, *value*, ...). The SQL statement allows multiple tuples to be inserted at one time. We restrict MMP to inserting just one tuple at a time with this operation. Insert Tuple succeeds if an undeleted relation with the specified name r exists in the most recent face, if the specified external source referent s exists in the model instance, and if all schema attribute specifiers named already exist in the specified relation. If successful, it induces a new face that consists of a copy of the most recent face, with the addition of a new tuple in the specified relation. The new tuple contains the specified value for each specified attribute listed in the operation. If a value is omitted for any attribute in the target relation's schema, then that value is set to NULL in the inserted tuple.

Drop Tuple(r,t,s) is the analog of the SQL operation DELETE FROM *tableName* WHERE (⟨matching condition⟩), though we limit this operation to specification of a single tuple for convenience in defining this operator. Drop Tuple succeeds if an undeleted relation with the specified name r exists in the most recent face, and if an undeleted tuple matching the tuple described in the operation exists in the relation, and if the specified external source referent s exists in the model instance. If successful, it induces a new database face that consists of a copy of the most recent face, with the specified tuple and all attribute values it contains marked as deleted. The s parameter specifies the external source referent that the user cites as responsible for

justifying the Drop Tuple action.

Paste Value(r_t,t_t,a_t,r_s,t_s,a_s) is the analog of the SQL operation INSERT INTO *tableName* WHERE .... Paste Value succeeds if the target relation r_t and the source relation r_s both exist undeleted in the most recent face, if they contain undeleted target tuple t_t and source tuple t_s, respectively, and if their schemas contain undeleted target attribute a_t and source attribute a_s, respectively. If successful, it induces a new database face that consists of a copy of the most recent face, with the addition of a copy of the specified source attribute value (from the source relation, tuple, and attribute) in the specified target column (attribute) of the specified target tuple in the target relation. If the pasted value already exists, the data contents of the new face are identical to those of the previous face. As with Insert Value, this operation fails if a conflicting attribute value exists in the specified attribute and tuple.

Paste Tuple(r_t,r_s,t_s) succeeds if the target relation r_t and the source relation r_s both exist undeleted in the most recent face, if the source relation contains undeleted source tuple t_s, if there is no tuple in r_t identical to t_s, and if the target relation has a union-compatible schema of undeleted attributes to the schema of undeleted attributes in the source relation. If successful, it induces a new database face that consists of a copy of the most recent face, with the addition of a copy of the specified source tuple in the specified target relation. There is no SQL analog of this operation.

Paste Relation(r_t,r_s) succeeds if the undeleted source relation r_s exists in the most recent face, and if there is no undeleted relation with the same name as the target relation in the most recent face. If successful, it induces a new face that consists of a copy of the most recent face, with the addition of a new relation, named r_t, copied in its entirety from r_s. There is no SQL analog of this operation.

Confirm Value(r,t,a,s) succeeds if the specified value (r,t,a) exists undeleted in the most recent face, and if the specified external source referent exists in the model instance. Confirm Value induces a new face that is a copy of the most recent face. It

records as part of the attribute value's provenance an indication of user belief about the value. It induces new content in the provenance model, discussed below. There is no SQL analog of this operation.

Doubt Value(r,t,a,s) succeeds if the specified value (r,t,a) exists undeleted in the most recent face, and if the specified external source referent exists in the model instance. Doubt Value induces a new face that is a copy of the most recent face. There is no SQL analog of this operation.

2.4.2.2   Query Operations

Well-formed queries succeed if the usual success conditions for relational queries are true in the most recent face. That is, all named input relations must exist undeleted, where necessary the schemas of the input relations must be union-compatible, and the named output relation must not already exist. A successful query induces a new face that consists of a copy of the most recent face, with the addition of a new relation containing the query result. Standard relational, set-oriented semantics are used. Note that unlike typical relational databases, every query result relation is named and materialized permanently in our model. We materialize these results because these relations may serve as inputs to future operations, and thus become part of the provenance of other data in the future. Details of the query language are presented in Chapter 3.

2.4.3   Confidence Language

The Confidence Language in MMP is, to our knowledge, unique among provenance models in the literature. Confirm Value and Doubt Value record an expression of confidence in an existing single attribute value in the model instance. Although these expressions do not affect the data values directly, they are recorded as part of the value's provenance, so each introduces a new face that is a copy of the prior face. A

user might use a confidence operator to express confidence or doubt in a data value at any point in its evolution. The confidence language is motivated by a previously unstated goal: they provide a means for users to weigh in on the trustworthiness of data without manipulating it. Another reason we include confidence operations is that they provide examples of operators that affect the contents of the provenance model without affecting the contents of the data model. Although we define confidence operators only at the attribute-value granularity, they could also be defined at other granularities.

2.4.4 Predicate Language for Selection and Projection Operators

As discussed in Chapter 1, many questions asked by domain experts involve selecting data by characteristics of its multi-generation provenance rather than by its single-generation provenance. That is, users want to select data by characteristics of ancestor components (as well as actions deriving them, or combinations of the two) anywhere in the graph recursively formed by edges originating at a component of interest and the components they connect to. In the discussion that follows, we call these structures *provenance graphs*. Moreover, we found that a significant portion of these provenance queries is describable in terms of characteristics of one or more *provenance paths* in a provenance graph, rather than characteristics of the entire graph. A provenance path is a non-branching path of finite length in a provenance graph. Our language for selecting data based on its provenance allows users to describe the provenance characteristics of a component of interest in terms of a pattern, or motif, that can be used to match these paths. Inherent in path motifs is the notion that the complete 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[4] (links) with particular properties. For example, a user might

---

[4]In MMP, the operation, the identity of the user that applied it, and the time at which it was applied appear as labels on the face induced by the operation. When we construct a provenance graph for a

be interested in data from a particular source, without knowing the full intervening history of its derivation.

We define a predicate language for use in the projection and selection operators of relational algebra used in MMP. These predicates describe characteristics of provenance paths. Components in the current face that have paths that match these predicates are projected or selected, respectively. Table 2.3 shows the grammar for the predicate language 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 for use in the selection operator. The selectionPredicate structure offers three options. A user may select tuples by their tuple provenance, or by the provenance of any attribute value belonging to the tuple. The projectionPredicate offers similar options: a user may select attributes by their provenance, or by the provenance of any attribute value belonging to the attribute.

The following examples show typical provenance questions phrased in natural language and in our selection predicate language. For selection predicates, the resulting query is of the form $DO = \sigma_{predicate}(DI)$, with input relation DI and output relation DO. For projection predicates, the resulting query is of the form $DO = \pi_{predicate}(DI)$.

- The question, "Which tuples were derived from source X?" can be expressed with the selectionPredicate *"tuple has a path with (a source with name = X)"*

- The question, "Which tuples have at least one data value derived from relation A or relation B?" can be expressed with the selectionPredicate *"some data value in tuple has a path with (a value in relation = A) or a path with (a value in relation = B)"*

---

component in an MMP instance, these labels are copied onto provenance links that originate from the induced face.

- The question, "Which tuples contain data derived from data in relation A that later appeared in relation C?" can be expressed with the selectionPredicate *"some data value in tuple has a path with (a value in relation = A before a value in relation = C)"*

- The question, "Which tuples are derived from tuples that were inserted at least once between dates D1 and D2?" can be expressed with the selectionPredicate *"tuple has a path with (an operation with action = INSERT and where time >= D1 and where time < D2)"*

- The question, "Which tuples had values derived from data inserted between dates D1 and D2 by user Y, and later deleted?", can be expressed with predicate *"some data value in tuple has a path with (an operation with (action = INSERT and where time > D1 and where time < D2 and by user = Y) before a value that is expired)"*

Provenance predicates may be combined with the usual predicate language available for the selection and projection operators, using the logical constructors AND, OR, and NOT. For example, assume that relation DI has an attribute named *shoe-size*. Then we could ask the question, "Which tuples in DI where shoe-size equals 9 contain data derived from data in relation A that later appeared in relation C?" The corresponding query can be expressed using the selection operator with the predicate *"shoe-size = 9 and some data value in tuple has a path with (a value in relation = A before a value in relation = C)"*. Note that none of the query examples above require the user to interact directly with representations of provenance, either for parsing or multi-generation traversals. This approach was motivated by Goals 7 and 8 from Section 2.1.

Table 2.3: Syntax of MMP Provenance Predicate Language

selectionPredicate ::= $componentSpecifier\ predicateQualifier$
componentSpecifier ::= TUPLE HAS
| SOME DATA VALUE IN TUPLE HAS

projectionPredicate ::= $prComponentSpecifier\ predicateQualifier$
prComponentSpecifier ::= ATTRIBUTE HAS
| SOME DATA VALUE IN ATTRIBUTE HAS

predicateQualifier ::= A PATH WITH (⟨pathQualifier⟩)
| A PATH WITH (⟨pathQualifier⟩) [AND|OR] ⟨predicateQualifier⟩

pathQualifier ::=
A ⟨component⟩ (⟨cQualSet⟩)
|AN OPERATION (⟨aQualSet⟩)
|A SOURCE (⟨sQualSet⟩)
| ⟨pathQualifier⟩ [BEFORE|AND] ⟨pathQualifier⟩

aQualSet ::= ⟨aQual⟩ | ⟨aQual⟩ [AND|OR] ⟨aQualSet⟩

cQualSet ::= ⟨cQual⟩ | ⟨cQual⟩ [AND|OR] ⟨cQualSet⟩

sQualSet ::= ⟨sQual⟩

aQual ::= WITH ACTION = ⟨constant⟩
| WITH ACTION = ANY QUERY
| BY USER = ⟨username⟩
| WHERE TIME ⟨cCmp⟩ ⟨timestamp⟩

cQual ::= IN RELATION ⟨relname⟩
| WITH A VALUE ⟨cCmp⟩ ⟨compval⟩
| WHERE EXPIRED = ⟨TRUE|FALSE⟩

sQual ::= WITH NAME = ⟨constant⟩

component ::= tuple | attribute | value

cCmp ::= = | > | < | ≥ | ≤ | ≠

2.5 Provenance Creation Semantics of the MMP Language

Here we briefly describe provenance links induced by MMP language operators. In our model, operators that target attribute values (Insert Value, Drop Value, Paste Value, Confirm Value, and Doubt Value) do not have inherited provenance. All other operators have inherited provenance.

Insert Value induces a single provenance link originating at the newly inserted attribute value in the result face and terminating at the external source referent from which the data was obtained. Similarly, Confirm Value, Doubt Value, and Drop Value each induce a single provenance link from the existing, affected attribute value and terminating at the external source referent cited in the operation.

Insert Tuple and Drop Tuple induce a single provenance link originating at the newly inserted or deleted tuple in the result face and terminating at the external source referent cited in the operation.

Paste Value induces a single provenance link originating at the newly created attribute value in the result face, and terminating at the cited attribute value in the predecessor face. Similarly, Paste Tuple and Paste Relation induce single provenance links from newly created destination structures to their cited source structures. The operators Create Relation, Create Source, and Create Attribute induce no provenance links. It would be possible to define these operators to induce a single provenance link originating at the relation, source, or attribute they create, and terminating at an external source referent. However, it seems to us that the semantics of these operators is that the user is creating a structure for data, instead of data itself, so they induce no data provenance to be recorded. For similar reasons, Drop Relation and Drop Attribute induce no provenance links. SPJU queries induce provenance links for the result relation and tuples of the query.

Informally, the rules for deriving inherited provenance links from the explicit provenance links described above are as follows:

1. If a tuple has a containing relation with an explicit provenance link to another relation then that tuple inherits a link

2. If a tuple has a containing relation with an explicit provenance link to an external source referent then that tuple inherits a link

3. If an attribute has a containing relation with an explicit provenance link to another relation then that attribute inherits that link

4. If an attribute has a containing relation with an explicit provenance link to an external source referent then that attribute inherits a link

5. If an attribute value has a containing tuple that has no explicit provenance, but has a containing relation with a provenance link to another relation, then that attribute value inherits that link

6. If an attribute value has a containing tuple that has no explicit provenance, but has a containing relation with a provenance link to an external source referent then that attribute value inherits that link

7. If an attribute value has a containing tuple that has an explicit provenance link to another tuples then that attribute value inherits that link

8. If an attribute value has a containing tuple that has an explicit provenance link to an external source referent then that attribute value inherits that link

These rules are defined formally in Section 3.7.1.1.

2.6 Provenance Graphs as Visualization Tools

Data faces and provenance links may be non-trivial for users to understand. In addition, the appearance of provenance links at multiple granularities may be confusing. The view of provenance displayed to users is typically a *provenance graph* derived

from these model structures. A provenance graph displays multi-generation provenance (showing inherited links as well as instantiated links) of a selected component in terms of other components at the same level of hierarchy, plus external source referents. A provenance graph may display all generations of provenance, or may display provenance only tracing back to a set of ancestor components specified by the user.

Figure 2.8 shows the provenance graph for attribute value 6 from attribute ID in tuple 1 in relation C at time $n + 6$ from the example in Figure 2.4. The dot notation used for vertex labels in the graph indicates relation first, then tuple number, then attribute, then attribute value. The timestamps associated with links are shown on the right of the figure. Edges in the graph labeled "continuity" indicate that the origin of the link is a component that exists at the specified timestamp because the component at the terminal of the link existed at the previous timestamp. At time $n + 6$, we see that the existing value $C.1.ID.6$ was the object of a Confirm Value operation that references external source W. At time $n + 5$, we see that the existing value $C.1.ID.6$ is re-inserted by a paste operation from $D.1.ID.6$. At time $n + 4$, the initial creation of $C.1.ID.6$ results from the query SELECT Name, ID FROM A WHERE Name = "John" UNION SELECT Name, ID FROM B WHERE Name = "John". In particular, we see that $C.1.ID.6$ has parents $A.2.ID.6$ and $B.2.ID.6$. We can also trace $A.2.ID.6$ to its original creation as part of an Insert Tuple operation that inserted tuple $\langle$John, $6\rangle$ into relation A from external source X. Similarly, we can trace $B.2.ID.6$ to its original creation as part of an Insert Tuple operation from external source W. Inherited provenance links, generated by MMP inheritance rules, are shown in the figure with dashed lines.
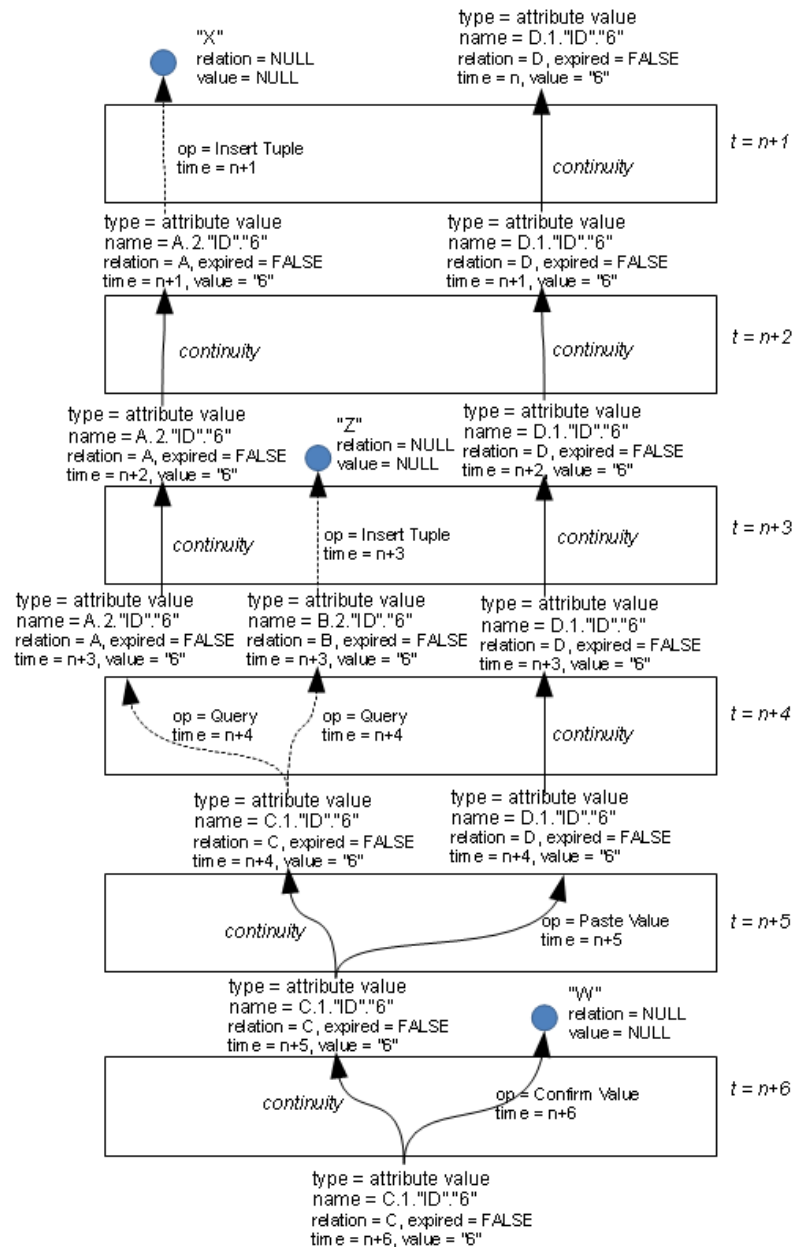
Figure 2.8: Example Provenance Graph. Inherited provenance links are shown using dotted lines

2.7 Chapter Summary

In this chapter, we defined goals for our conceptual model for data and its provenance; defined the MMP model to meet those goals; and introduced two user views of MMP: a relational view of data, and a graph view of data provenance. The relational view, or face, can be manipulated with a language familiar to relational database users, yet incorporating new features that address needs of data curators. The provenance graph view allows visual inspection of the provenance of selected data, and can easily be extended to enable graph algorithms to further query data provenance.

Chapter 3

Formalizing the Conceptual Model

In this chapter, we formally define MMP in order to be precise about its structure and operations, and in order to better characterize it in terms of expressive power and semantics. As part of this definition, we specify a semantics for provenance in SPJU queries at all granularities of relational data, something we have not seen elsewhere in the literature. We also define two mechanisms (in addition to the predicate language we defined in Chapter 2) to access provenance information in our model: a graphical representation of provenance intended to allow users to browse provenance of a selected data item; and an algebraic representation of provenance that extends representations in the current literature by representing multi-generation provenance and by including the history of operations applied to data and the identity of the agents that applied them.

We begin by defining the data portion of MMP and the way in which external data sources outside MMP are represented. Next, we define the provenance portion of our model. We follow this with definition of operations over model data and information sources. With these basic mechanisms defined, we then define how operations, data, and provenance interact. Finally, we define mechanisms to access provenance information in our model.

3.1 Modeling Evolving Data: Faces

An MMP instance M includes a finite ordered tuple of databases $D = (d_1, d_2, \ldots, d_n)$, where n is the current number of faces. This formalizes the idea that a model has a succession of faces, ordered by time as discussed in Chapter 2. An MMP instance also includes a set of labels $l_D = TS \times OpD \times U$, where each $ts \in TS$ is a timestamp, each $op \in OpD$ describes an operation from the MMP language, and each $u \in U$ identifies a user of M. An MMP instance also includes a labeling function $\lambda_D : D \to l_D$, which labels a face $d \in D$ with its time of creation, the operation that induced it from its immediate predecessor face, and the identity of the user who applied that operation. D is ordered by increasing timestamps of the label associated with each $d \in D$. These labels provide the "when", "who", and "how" portions of provenance for all provenance links that originate at the new face and terminate at the preceding face or the appropriate external source referents.

Because we support relational data, each face $d \in D$ is a relational database. Consistent with the definition of a relational database, each $d_i \in D$ consists of a finite set of relations R_i; and each $r_j \in R_i$ consists of a finite set of tuples $T_{i,j}$ sharing a common schema consisting of a finite set of attributes $A_{i,j}$. In the discussion below, we refer to individual attributes as $a_{i,j,l}, 1 \leq l \leq |A_{i,j}|$. Each tuple $t_{i,j,k} \in T_{i,j}$ includes a set $V_{i,j,k}$ of attribute values where each $v_{i,j,k,l} \in V_{i,j,k}$ is taken from the domain of $a_{i,j,l}$. We refer to $C_i = R_i \cup T_{i,j} \cup A_{i,j} \cup V_{i,j,k}, 1 \leq j \leq |R_i|, 1 \leq k \leq |T_{i,j}|$ as the *components* in d_i. We refer to $\bigcup_i C_i, 1 \leq i \leq n$ as the components in D. Figure 3.1 shows an example face $d_i \in D$ from an MMP instance. The figure shows two relations A and B comprising R_i. A has attributes Name and ID comprising $A_{i,A}$. B also has attributes Name and ID comprising $A_{i,B}$. A has two tuples, $\langle$Bob,8$\rangle$ and $\langle$John,6$\rangle$ comprising $T_{i,A}$. Attribute values Bob and 8 thus comprise $V_{i,A,\langle Bob,8\rangle}$.
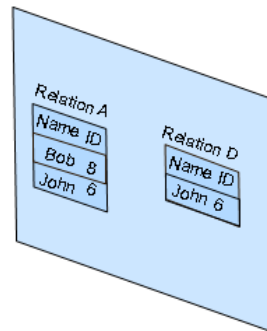
Figure 3.1: Example Face in an MMP Instance

3.2 Modeling The Outside World: External Source Referents

In addition to faces, an MMP instance M includes a finite set S of *external source referents* that represent sources from which data may be inserted into database faces. S is initially empty and acquires a new element when a Create Source command is processed. When data is excerpted from or inspired by an external source and inserted into some $d_i \in D$, the induced provenance link terminates at the corresponding $s \in S$.

3.3 Modeling Data Derivation: Provenance Links

The provenance of each component in an MMP face is modeled as a set of relationships between that component and its ancestor components. Because each operator (or composition of query operators forming a single query) in the MMP language takes the current database face d_n as input and produces a new database face d_{n+1}, a provenance relationship links a component in one face to those in the immediately preceding face, or to external source referents. An MMP instance M includes a finite set of *provenance links* L, where each $l_p \in L$ is a relationship (a hyper-edge) from a component $c_{n+1} \in d_{n+1}$ to one or more components $c_n \in d_n$ or to an external source referent $s \in S$. Figure 3.2 shows provenance links connecting components on one
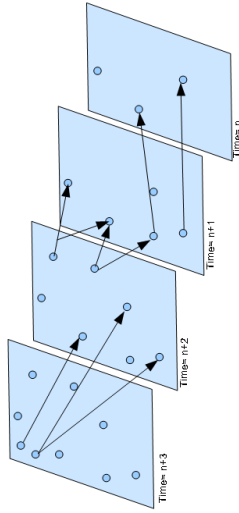
Figure 3.2: Faces and Provenance Links in an MMP instance

face to parents in the preceding face. Each edge shown in the figure is a link $l_p \in L$. Each origin of an edge is a component $c \in C_i$, where C_i is the set of components in face d_i.

MMP models two forms of provenance relationships. *Operation-induced* provenance links, which we refer to as *provenance links*, model derivation of components from other components by operations applied to the database. *Continuity* provenance links, which we refer to as *continuity links*, represent the continuing presence of components in a face due to the presence of identical predecessors in the prior face. Continuity links arise by default for all components in a new face that are not newly created by the operation that induces the face.

3.3.1 Operation-induced Provenance Links

A component created by or affected by a DDL or DML operation that induces face d_{n+1} originates a single provenance link that terminates at an external source referent (for insertions or deletions) or a component in face d_n (for copy-and-paste). Similarly, a component affected by a data confidence language (DCL) operation that

induces face d_{n+1} originates a single provenance link that terminates at an external source referent. A component created in face d_{n+1} as part of a query result may originate multiple provenance links, each of which might have multiple terminals at components in face d_n. Let C_n be the set of all components in d_n from which a particular c_p in d_{n+1} is derived. Let B_n be a subset of C_n. Then we define a provenance link $l_p(c_p, B_n)$ to be a 1-to-many pairing

$$c_p : B_n$$

indicating that the components in B_n give rise to c_p as a result of the operation that induced d_{n+1}. We define a provenance link $l_p(c_p, s)$ to be a binary pair

$$c_p : s \in S$$

indicating that external source s gave rise to c_p as a result of an operation that induced d_{n+1}. In each case, c_p is called the *origin* of l_p, and each $b \in B_n$ or s is called a *terminal* of l_p. A link $l_p(c_p, B_n)$ or $l_p(c_p, s)$ indicates that c_p was affected (created, deleted, or annotated with an expression of confidence or doubt) by the operation that induced d_{n+1} by acting on B_n or s. In Chapter 2, we explained that provenance links always terminate at the same component type from which they originate (or at external source referents). For example, result attribute values always have other attribute values, or external source referents, as immediate ancestors. Similarly, result tuples always have other tuples, or external source referents, as ancestors. Thus each $b \in B_n$ in a provenance link is of the same component type as c_p.

A component c_p may originate zero or more provenance links. Attribute values arising from query constants (for example, $\pi_{1,Name}(A)$ introduces a constant 1 in each output tuple) originate no provenance links, because they do not arise from data in d_n. An external source referent $s_i \in S$ also originates no provenance links, as explained in Chapter 2. A component in a face d_n may terminate any number of links originating at components in face d_{n+1}, because it may be an input that affects many

result components in the operation that induced d_{n+1}. An external source referent may terminate any number of links, because it may be the source used in multiple distinct operations.

Operations that induce provenance define the provenance of a single component of the type named in the operation (which we call the *target* component), and may also define the provenance of components that are contained in that component. (Note that a query implicitly defines the provenance of its result relation.) For example, the Insert Tuple operation defined in Chapter 2 affects the provenance of a single tuple in the result face, but also affects the provenance of attribute values contained in that tuple. As another example, a Paste Relation operation affects the provenance of a single relation, but also affects the provenance of the tuples, attributes, and attribute values contained in that relation. For all operations in the MMP language, MMP explicitly records the provenance induced on the target component of the operation. Components contained in the target component have provenance explicitly recorded if the applied operation does not have inherited provenance semantics. Otherwise, contained components do not have explicit provenance recorded due to the operation. Their provenance due to an operation is inherited from the provenance of the containing operation target component.

3.3.2 Continuity Provenance Links

In MMP, we say that data has *continuity*: each component in one face that is not newly created by the operation that induced the face has a corresponding component in the preceding face. That is, when a component $c_p \in d_{n+1}$ is not newly created by the operation that induces d_{n+1}, then c_p has an identical *predecessor* $c_p \in d_n$ of the same component type[1]. For example, suppose a relation A exists in face d_{n+1}, and that d_{n+1} was induced by an operation that inserted a new tuple in some other

---

[1]We expect a modification of a current component to be a deletion followed by an insertion for all appropriate components

relation B. Then a prior version of A existed in d_n. We define that prior version to be the *predecessor* of A.

We define continuity links for connecting components with their predecessors. Components newly created by the operation that induced d_{n+1} originate no continuity links, because they have no predecessors in d_n. (Recall that they originate provenance links to appropriate ancestors, as explained above.) In a verbose variation of MMP, each component in d_{n+1} other not newly created by the operation that induced d_{n+1} originates a single continuity link to its predecessor. However, in the more concise version we describe here, continuity links need not be explicitly recorded in an MMP instance because they can easily be inferred by inspection of two consecutive database faces. We specify continuity link inference rules when we discuss provenance graphs later in this chapter.

3.4 Modeling Operations Applied to Data: Revisions

We now introduce operations that are applied to data, and define how data and provenance are affected or created by these operations. We define operations on external source referents in the next section.

Recall that an MMP instance M defines an ordered tuple of faces D, a set of labels for those faces l_D, a function λ_D that maps faces to their labels, a set of external source referents S, and a set of provenance links L. Upon creation of M, $D = \emptyset, S = \emptyset$, and $L = \emptyset$, because an empty database has no data. Since no operations have yet been applied to it, no faces exist, no external sources have been defined, and there can be no provenance relationships. OpD is the set of operations in the MMP language; U is initialized to the set of users who will issue operations against data in M, and TS is a totally ordered set of timestamps at which users may apply operations. Subsequent *revisions* populate D and L. A revision $\Re$ is a mapping

$$\Re : (D \times S \times L \times OpD \times U \times TS) \rightarrow (D' \times L')$$

Informally, we say that $\Re$ applies operator $op \in OpD$ by user $u_{op} \in U$ at time $t_{op} \in TS$ to the most recent database face $d_n \in D$ in order to add a new database face d_{n+1} to D and a new set of provenance links to L. That is, each revision induces an entirely new database, and adds the appropriate provenance links to the MMP instance to represent derivation of the new database from the prior database and external source referents.

Face d_{n+1} is labeled with $\langle t_{op}, op, u_{op} \rangle$ using $\lambda_{D'}$. The timestamp of each successor face is relative rather than absolute, and strictly greater than the timestamp of its predecessor face. We refer to database faces d_i by their subscript i. Labeling the new database with the operation, user, and timestamp in the definition allows the user to see this information when provenance is examined.

Operations never delete data from an MMP instance because users may later wish to inspect deleted data as part of provenance relationships. Although MMP retains deleted data for inspection, deleted data does not participate in operations in OpD. If a component is deleted and an identical component later re-introduced, these components are considered distinct in MMP. In order to distinguish deleted data in our model, we define a set of functions, one for each component type, that return TRUE if a component has been deleted, and FALSE otherwise. These functions are $Expired_r()$ over the domain of all relations in an MMP instance M, $Expired_a()$ over attributes in M, $Expired_t()$ over tuples in M, and $Expired_v()$ over attribute values in M.

$$Expired_r(i, j) =$$

$$\begin{cases} True, & \text{if the } j^{th} \text{ relation in face i has been deleted} \\ False, & \text{otherwise} \end{cases}$$

$$Expired_a(i, j, l) =$$

$$\begin{cases} True, & \text{if the } l^{th} \text{ attribute in relation j in face i has been deleted} \\ False, & \text{otherwise} \end{cases}$$

$$Expired_t(i, j, k) =$$

$$\begin{cases} True, & \text{if the } k^{th} \text{ tuple in relation j in face i has been deleted} \\ False, & \text{otherwise} \end{cases}$$

$$Expired_v(i, j, k, l) =$$

$$\begin{cases} True, & \text{if the value of the } l^{th} \text{ attribute in } t_{i,j,k} \\ & \text{has been deleted} \\ False, & \text{otherwise} \end{cases}$$

Upon creation, each component c has Expired(c) set to False. Upon deletion of a component c, Expired(c) is set to True. Once a component c is deleted, Expired(c) cannot be changed to False thereafter. Components c with Expired(c) = True are not available for use by operators, but such components may appear in the provenance of other components.

3.5   Modeling Creation of External Source Referents

Revisions formally define the impact of operations on data and provenance. We also define the impact of operations on external source referents. The Create Source operator creates external source referents s which are added to S to represent a source from which data may be inserted into database faces by subsequent revisions. A *source creation* SC is a mapping

$$SC : S \times \text{Create Source} \times U \times TS \rightarrow S'$$

Informally, we say that SC applies the Create Source operator by user $u_{op} \in U$ at time $t_{op} \in TS$ to introduce a new external source referent $s \in S$. Note that source creations do not affect faces or their contents, nor do they induce provenance links. Upon creation, each $s \in S$ has the name of the external source it represents. Note that sources are never dropped or deleted.

3.6 Single-revision and Source-Creation Impact on Data and Provenance

Each revision applied to an MMP instance induces provenance links from components in the resulting new face to their *immediate ancestors*, which may be either parent components in the preceding face, or an external source referent. The induced links describe the provenance of components in the new face due to this revision. We use the term *single-revision provenance* of $\Re$, $Prov^{S(\Re)}$, to denote the links added to L by a single application of $\Re$.

In Chapter 2, we informally defined the operators in the MMP language, along with the effects they have on data when applied to an MMP instance. Also in Chapter 2, we gave an overview of how those operators induce provenance links. In this section, we formally define revisions implementing these operations, and their effects on data and provenance in MMP.

For each revision we define below, let the initial state of an MMP instance M, subject to the revision, be as follows. D is the set of faces in M. The current (most recent) face in D is d_i. d_i is a relational database with relation set R_i. Each relation $r_j \in R_i$ has attribute set $A_{i,j} = \{a_{i,j,l}\}, 1 \leq l \leq m$, where m is the number of attributes in r_j. Each $r_j \in R_i$ also has a set of tuples $T_{i,j} = \{t_{i,j,n}\}, 1 \leq n \leq k$, where k is the number of attributes in r_j. Each tuple in r_j has a set of attribute values $V_{i,j,k} = \{v_{i,j,l,n}\}$. L is the set of provenance links in M. S is the set of external source referents in M. Also, let $u \in U$ describe a user of M, and $t \in TS$ be the time at which u applies an operation to M, resulting in revision $\Re$. We use similar notation for the relations, attributes, tuples, and values for the other faces in $D, d_x, 1 \leq x \leq i - 1$.

In the definitions below, we frequently state that a new face, d_{i+1}, is a copy of a prior face, d_i, by saying $d_{i+1} = d_i$. By this, we mean that all of the relations in R_i also exist in R_{i+1}, with the same names; that each relation $r_{i+1,j}$ has an attribute set

$A_{i+1,j}$ identical to the attribute set $A_{i,j}$ of its corresponding $r_{i,j}$; and that each relation $r_{i+1,j}$ has the same instance $T_{i+1,j}$ as the instance $T_{i,j}$ of the corresponding $r_{i,j}$.

3.6.1 DDL Revisions and Source Creations

Revisions implementing DDL operations and source creation are defined in this section. As described in Chapter 2, these operations do not induce provenance links, though two of these operations, Drop Attribute and Drop Relation create continuity links.

3.6.1.1 Create Relation

$\Re(D, S, L, \text{Create Relation}(r_j), u, time) =$

if $r_j \notin R_i$ then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i, R_{i+1} = R_{i+1} \cup r_j.$$

$$A_{i+1,j} = \emptyset,$$

$$T_{i+1,j} = \emptyset,$$

$$L' = L.$$

$$S' = S.$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Create Relation", u} \rangle.$$

$$Expired_r(i+1, j) = \text{False}.$$

else Create Relation fails.

Note that the timestamp $time$ is a natural number, and is guaranteed to be monotonically increasing.

3.6.1.2 Create Source

$SC(S, \text{Create Source}(s_{new})) =$

if $s_{new} \notin S$ then $S' = S \cup s_{new}$. The remainder of M is unchanged.

else Create Source fails.

3.6.1.3 Create Attribute

$\Re(D, S, L, \text{Create Attribute}(r_j, a_l), u, time) =$

if $r_j \in R_i \wedge Expired_r(i, j) = \text{False} \wedge a_l \notin A_{i,j}$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$A_{i+1,j} = A_{i+1,j} \cup a_l.$$

$$\forall t_{i+1,j,k} \text{ in } T_{i+1,j}, 1 \leq k \leq |T_{i+1,j}|, v_{i+1,j,k,l} = NULL.$$

$$L' = L.$$

$$S' = S.$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Create Attribute", u} \rangle.$$

$$Expired_a(i + 1, j, l) = \text{False}.$$

else Create Attribute fails.

3.6.1.4 Drop Relation

$\Re(D, S, L, \text{Drop Relation}(r_j), u, time) =$

if $r_j \in R_i \wedge Expired_r(i, j) = \text{False}$ then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$Expired_r(i + 1, j) = \text{True}.$$

$$\forall l \text{ such that } a_{i+1,j,l} \in A_{i+1,j}, Expired_a(i + 1, j, l) = \text{True},$$

$$\forall k \text{ such that } t_{i+1,j,k} \in T_{i+1,j}, Expired_t(i+1,j,k) = \text{True},$$

$$\forall k \text{ such that } t_{i+1,j,k} \in T_{i+1,j}, \forall l \text{ such that } v_{i+1,j,k,l} \in V_{i+1,j,k},$$

$$L' = L.$$

$$S' = S.$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Drop Relation", u} \rangle.$$

$$Expired_v(i+1,j,k,l) = \text{True}.$$

else Drop Relation fails.

3.6.1.5 Drop Attribute

$\Re(D, S, L, \text{Drop Attribute}(r_j, a_l), u, time) =$
if $r_j \in R_i \wedge Expired_r(i,j) = \text{False} \wedge a_{i,j,l} \in A_{i,j} \wedge Expired_a(a_{i,j,l}) = \text{False}$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$Expired_a(i+1,j,l) = \text{True}.$$

$$\forall k \text{ such that } t_{i+1,j,k} \in T_{i+1,j}, Expired_v(i+1,j,k,l) = \text{True}.$$

$$L' = L.$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Drop Attribute", u} \rangle.$$

else Drop Attribute fails.

3.6.2 DML and DCL Revisions

This section defines revisions that implement data manipulation and confidence operations. In addition to affecting MMP faces, each of these operations creates new provenance links.

3.6.2.1 Insert Value

Let $t = \langle\{v_h\}\rangle, 1 \leq h \leq |A_{i,j}|$, such that v_h is the attribute value in t
corresponding to $a_{i,j,h} \in A_{i,j}$.

$\Re(D, S, L, \text{Insert Value}(r_j, t, a_l, v_{new}, s), u, time) =$

if $r_j \in R_i \wedge Expired_r(i,j) = \text{False} \wedge a_l \in A_{i,j} \wedge Expired_a(i,j,l) = \text{False} \wedge \exists t_{i,j,k} \in$
$T_{i,j}$ such that $t = t_{i,j,k} \wedge Expired_t(i,j,k) = \text{False} \wedge (v_{i,j,k,l} = NULL \vee v_{i,j,k,l} = $
$v_{new}) \wedge s \in S$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$v_{i+1,j,k,l} = v_{new}.$$

$$L' = L \cup l_p(v_{i+1,j,k,l}, s).$$

$$\lambda_D(d_{i+1}) = \langle time, \text{``Insert Value''}, u\rangle.$$

$$Expired_v(i+1, j, k, l) = \text{False}.$$

else Insert Value fails.

3.6.2.2 Drop Value

Let $t = \langle\{v_h\}\rangle, 1 \leq h \leq |A_{i,j}|$, such that v_h is the attribute value in t
corresponding to $a_{i,j,h} \in A_{i,j}$.

$\Re(D, S, L, \text{Drop Value}(r_j, t, a_l, s), u, time) =$

if $r_j \in R_i \wedge Expired_r(i,j) = \text{False} \wedge a_l \in A_{i,j} \wedge Expired_a(i,j,l) = \text{False} \wedge$
$\exists t_{i,j,k} \in T_{i,j}$ such that $t = t_{i,j,k} \wedge Expired_t(i,j,k) = \text{False} \wedge v_{i,j,k,l} \in V_{i,j} \wedge$
$Expired_v(i,j,k,l) = \text{False} \wedge s \in S$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$L' = L \cup l_p(v_{i+1,j,k,l}, s).$$

$$\lambda_D(d_{i+1}) = \langle time, \text{``Drop Value''}, u\rangle.$$

$$Expired_v(i+1, j, k, l) = \text{True}.$$

else Drop Value fails.

3.6.2.3 Insert Tuple

Let $t = \langle\{v_h\}\rangle, 1 \le h \le |A_{i,j}|$, such that v_h is the attribute value in t
corresponding to undeleted $a_{i,j,h} \in A_{i,j}$.
$\Re(D, S, L, \text{Insert Tuple}(r_j, t, s), u, time) =$
if $r_j \in R_i \land Expired_r(i, j) = \text{False} \land s \in S$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i,$$

$$T_{i+1,j} = T_{i,j} \cup t,$$

$$L' = L \cup l_p(t, s).$$

$$\lambda_D(d_{i+1}) = \langle\text{time, "Insert Tuple", u}\rangle.$$

$$Expired_t(t) = \text{False}.$$

$$\forall v \in t, Expired_v(v) = \text{False}.$$

else Insert Tuple fails.

3.6.2.4 Drop Tuple

Let $t = \langle\{v_h\}\rangle, 1 \le h \le |A_{i,j}|$, such that v_h is the attribute value in t
corresponding to $a_{i,j,h} \in A_{i,j}$.
$\Re(D, S, L, \text{Drop Tuple}(r_j, t, s), u, time) =$
if $r_j \in R_i \land Expired_r(i, j) = \text{False} \land \exists t_{i,j,k} \in T_{i,j}$ such that t is identical to $t_{i,j,k} \land$
$Expired_t(i, j, k) = \text{False} \land s \in S$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$L' = L \cup l_p(t_{i+1,j,k}, s).$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Drop Tuple", u} \rangle.$$

$$Expired_t(i+1, j, k) = \text{True}.$$

$$\forall v_{i+1,j,k,l} \in V(i+1, j, k), Expired_v(i+1, j, k, l) = \text{True}.$$

else Drop Tuple fails. Here and in later uses, by t is identical to $t_{i,j,k}$, we mean that each undeleted attribute value in t has a corresponding undeleted attribute value in $t_{i,j,k}$, and all such (undeleted) corresponding pairs of attribute values are equal.

3.6.2.5 Paste Value

Let $t = \langle \{v_h\} \rangle, 1 \le h \le |A_{i,j}|$, such that v_h is the attribute value in t
 corresponding to $a_{i,j,h} \in A_{i,j}$.
Let $t_s = \langle \{v_f\} \rangle, 1 \le f \le |A_{i,js}|$, such that v_f is the attribute value in t_s
 corresponding to $a_{i,js,f} \in A_{i,js}$.
$\Re(D, S, L, \text{Paste Value}(r_j, t, a_l, r_{js}, t_s, a_{ls}), u, time) =$
If $r_j \in R_i \wedge Expired_r(i, j) = \text{False} \wedge a_l \in A_{i,j} \wedge Expired_a(i, j, l) = \text{False} \wedge$
$\exists t_{i,j,k} \in T_{i,j}$ such that t is identical to $t_{i,j,k} \wedge Expired_t(i, j, k) = \text{False} \wedge r_{js} \in$
$R_i \wedge Expired_r(i, js) = \text{False} \wedge a_{ls} \in A_{i,js} \wedge Expired_a(i, j, ls) = \text{False} \wedge \exists t_q \in$
$T_{i,js}$ such that t_q is identical to $t_s \wedge Expired_t(i, js, q) = \text{False} \wedge (v_{i,j,k,l} = NULL \vee$
$v_{i,j,k,l} = v_{new})$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$v_{i+1,j,k,l} = v_{i,js,q,ls}.$$

$$L' = L \cup l_p(v_{i+1,j,k,l}, v_{i,js,q,ls}).$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Paste Value", u} \rangle.$$

$$Expired_v(i+1, j, k, l) = \text{False}.$$

else Paste Value fails.

3.6.2.6 Paste Tuple

Let $t_s = \langle \{v_f\} \rangle, 1 \leq f \leq |A_{i,j}|$, such that v_f is the attribute value in t_s
corresponding to $a_{i,j,f} \in A_{i,j}$.

$\Re(D, S, L, \text{Paste Tuple}(r_j, r_{js}, t_s), u, time) =$

If $r_j \in R_i \wedge Expired_r(i,j) = \text{False} \wedge r_{js} \in R_i \wedge Expired_r(i, js) = \text{False} \wedge$
the undeleted members of $A_{i,j} = $ the undeleted members of $A_{i,js} \wedge \exists t_{i,js,ks} \in T_{i,js}$
such that $t_{i,js,ks}$ is identical to $t_s \wedge Expired_t(i, js, ks) = \text{False}$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$T_{i+1,j} = T_{i+1,j} \cup t_{i,js,ks}$ and we refer to the new tuple as $t_{i+1,j,new}$.

$$L' = L \cup l_p(t_{i+1,j,new}, t_{i,js,ks}).$$

$$\lambda_D(d_{i+1}) = \langle time, \text{"Paste Tuple"}, u \rangle.$$

$$Expired_t(i+1, j, new) = \text{False}.$$

$\forall v_{i+1,j,new,l} \in V(i+1, j, new), 1 \leq l \leq |A_{i+1,j}|, Expired_v(i+1, j, new, l) = \text{False}.$

else Paste Tuple fails.

3.6.2.7 Paste Relation

$\Re(D, S, L, \text{Paste Relation}(r_t, r_s)), u, time) =$

If $r_s \in R_i \wedge Expired_r(i, s) = \text{False} \wedge r_t \notin R_i$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$R_{i+1} = R_i \cup r_t.$$

$$T_{i+1,t} = T_{i,s}.$$

$$A_{i+1,t} = A_{i,s}.$$

$$\forall k \text{ such that } t_{i,s,k} \in T_{i,s}, V_{i+1,t,k} = V_{i,s,k}.$$

$$L' = L \cup l_p(r_{i+1,t}, r_{i,s}).$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Paste Relation", u} \rangle.$$

$$Expired_r(i+1, t) = \text{False}.$$

$$\forall a_{i+1,t,l} \in A_{i+1,t}, Expired_a(i+1, t, l) = \text{False}.$$

$$\forall t_{i+1,t,k} \in T_{i+1,t}, Expired_t(i+1, t, k) = \text{False}.$$

$$\forall v_{i+1,t,k,l} \in V_{i+1,t,k}, Expired_v(i+1, t, k, l) = \text{False}.$$

else Paste Relation fails.

3.6.2.8 Confirm Value and Doubt Value

Semantics for these two operations are identical except for the operation that appears in the label for the result face, so only one definition is provided here.

Let $t = \langle \{v_h\} \rangle, 1 \le h \le |A_{i,j}|$, such that v_h is the attribute value in t corresponding to $a_{i,j,h} \in A_{i,j}$.

$\Re(D, S, L, \text{Confirm Value}(r_j, t, a_l, s), u, time) =$

if $r_j \in R_i \wedge Expired_r(i, j) = \text{False} \wedge a_l \in A_{i,j} \wedge Expired_a(i, j, l) = \text{False} \wedge \exists t_{i,j,k} \in T_{i,j}$ such that $t_{i,j,k}$ is identical to $t \wedge Expired_t(i, j, k) = \text{False} \wedge v_{i,j,k,l} \in V_{i,j,k} \wedge Expired_v(i, j, k, l) = \text{False} \wedge s \in S$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

$$L' = L \cup l_p(v_{i+1,j,k,l}, s).$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Confirm Value", u} \rangle.$$

else Confirm Value fails.

3.6.3 Query Revisions

In this section, we describe the effects on provenance due to revisions that implement query operations. Because the semantics of queries in the SPJU fragment of relational algebra are well understood with regard to data, we do not repeat these here. However, we note that as a consequence of our goal of supporting standard relational algebra semantics, we define that queries in MMP ignore components c with $Expired(c)$ = True. That is, if there are input relations r with $Expired_r(r)$ = True, queries are considered not well formed and are ignored. Attributes a in input relations with $Expired_a(a)$ = True are ignored. Tuples t in input relations with $Expired_t(t)$ = True are ignored. Attribute values v of non-deleted attributes in non-deleted tuples in input relations, with $Expired_v(v)$ = True, are considered to be NULL.

We introduce query-induced provenance by first defining the provenance induced by the individual relational algebra operations supported by MMP. We then define provenance induced by general SPJU queries.

Recall that an SPJU query can be equivalently expressed as a union of SPJ query terms. We re-write joins to be the composition of a selection operator and a Cartesian product operator. Using the commutativity property of selection and Cartesian product operators, we then commute all selection operators to the left of all Cartesian products in query terms. As a result, we obtain for each SPJ term an equivalent SPX term: a projection composed with a selection composed with zero or more Cartesian product operators. We also re-write query terms by renaming attributes in the result of each term so that all terms in a query have consistent names for corresponding attributes.

We considered a range of possibilities regarding inheritance of provenance in query results. At one extreme, provenance for queries might be *flat*, in that components of all types have explicit provenance links induced by the query that created them. This has the disadvantage that a substantial number of provenance links might

be induced for a query. Contrast this with the other extreme where no provenance links need be recorded at all. With this approach, construction of provenance links may be done on demand, after the query is run, by inspection of the text of the query and the input data. However, Cong, Fan, and Geerts have shown [11] that inferring provenance after the fact for tuples subjected to projection operations in a query requires that we retain in the query result at least one candidate key for each query input relation. This has the disadvantage of limiting the expressive power of the query language that MMP supports. A middle ground induces explicit provenance links for relations and tuples, allowing inheritance rules to generate provenance links on demand for attributes and attribute values. This approach has the advantages that the number of explicit provenance links is reduced from the flat approach, while the expressive power of the query language is not limited by the need to retain candidate keys. We choose this middle ground in MMP. This represents a balance between goals 19 and goal 11 from Chapter 2.

3.6.3.1 Selection Operator Provenance

$\Re(D, S, L, (r_{out} = \sigma_{condition}(r_{in}), u, time)) =$

if $condition$ is well-formed $\wedge\, r_{out} \notin R_i \wedge r_{in} \in R_i$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

r_{out} is the query result as defined by the semantics of relation queries.

Let r_{out} be called $r_{i+1,j}$ and r_{in} be called $r_{i,n}$. Add $r_{i+1,j} to R_{i+1}$.

$$L' = L \cup l_p(r_{i+1,j}, r_{i,n}).$$

$\forall t_{i+1,j,k} \in T_{i+1,j}, 1 \le k \le |T_{i+1,j}|$

 and the single $t_{i,n,u} \in T_{i,n}$ where $t_{i+1,j,k}$ is identical to $t_{i,n,u}$

$$L' = L' \cup l_p(t_{i+1,j,k}, t_{i,n,u}).$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "Select operation (condition)",u} \rangle.$$

else the query fails and no provenance links are created.

Figure 3.3(a) shows the provenance link induced for the result relation of a selection as a solid blue line, the provenance link induced for each tuple as a solid yellow line, and the links inherited by attributes and attribute values as red and green dotted lines, respectively. The rules for inheritance are define in the section on provenance graphs, below.

3.6.3.2 Projection Operator Provenance

$\Re(D, S, L, r_{out} = \pi_{columnList}(r_{in}), u, time) =$

if $columnList$ is well-formed $\wedge \, r_{out} \notin R_i \wedge r_{in} \in R_i$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

r_{out} is the query result as defined by the semantics of relation queries.

Let r_{out} be called $r_{i+1,j}$ and r_{in} be called $r_{i,n}$. Add $r_{i+1,j} to R_{i+1}$.

$$L' = L \cup l_p(r_{i+1,j}, r_{i,n}).$$

$\forall t_{i+1,j,k} \in T_{i+1,j}, 1 \le k \le |T_{i+1,j}|,$

$$L' = L' \cup \bigcup_{X=1}^{U} l_p(t_{i+1,j,k}, t_{i,n,X})$$

such that each $t_{i,n,X} \in T_{i,n}$ is one of the U tuples from which $t_{i+1,j,k}$ was derived by the projection.

$$\lambda_D(d_{i+1}) = \langle \text{time, "Projection operation (column-list)", u} \rangle.$$

else the query fails and no provenance links are created.

Figure 3.3(b) shows the provenance links induced for the result relation and tuples of a projection along with the provenance links inherited by attributes and attribute values, using the same color and line scheme as Figure 3.3(a).
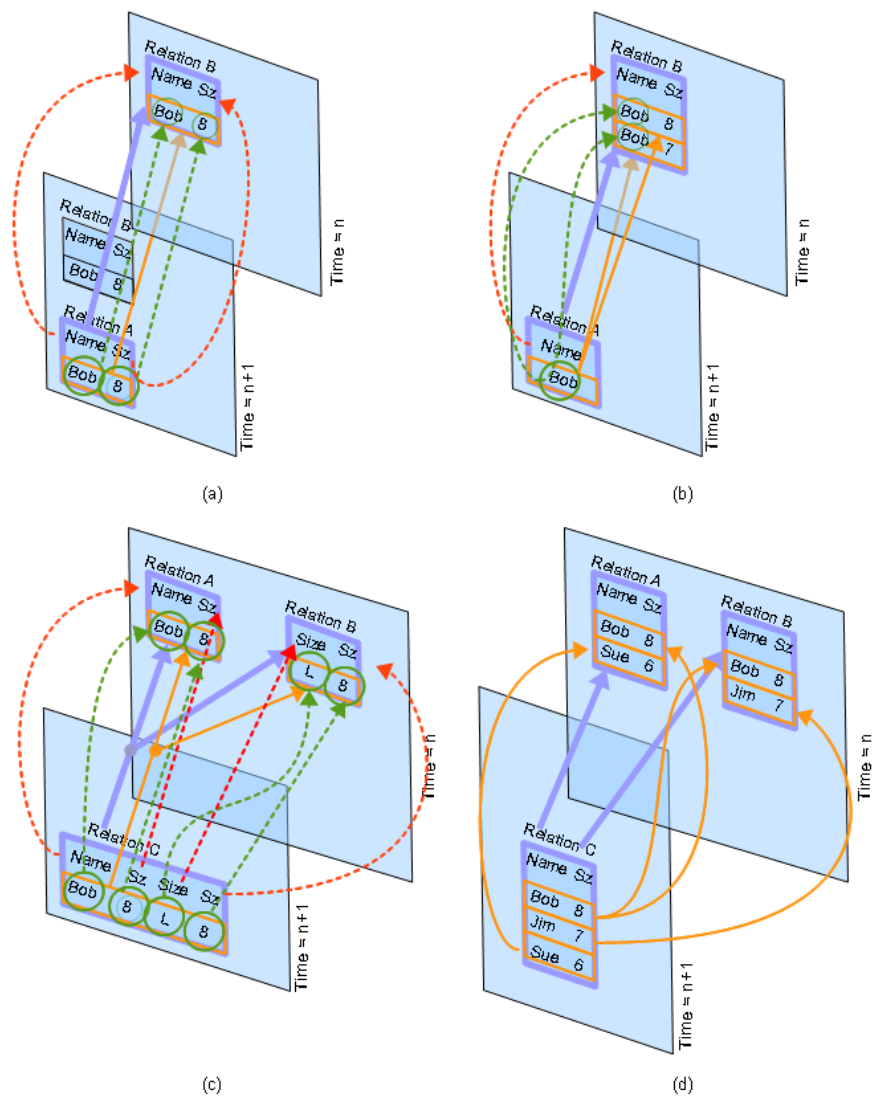
Figure 3.3: Single-Revision Provenance Resulting from Select (a), Project (b), Cartesian product (c), and Union (d) Operations.

3.6.3.3 Cartesian Product Operator Provenance

We define provenance for the polyadic Cartesian product.

$\Re(D, S, L, (r_{out} = r_{in1} \times \ldots \times r_{inM}), u, time) =$

if $r_{out} \notin R_i \wedge r_{in1}, \ldots r_{inM} \in R_i$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

r_{out} is the query result as defined by the semantics of relation queries.

Let r_{out} be called $r_{i+1,j}$ and each r_{inX} be called $r_{i,X}, 1 \leq X \leq M$. Add $r_{i+1,j} to R_{i+1}$.

$$L' = L \cup l_p(r_{i+1,j}, \bigcup_{X=1}^{M} r_{i,X}).$$

$\forall t_{i+1,j,k} \in T_{i+1,j}, 1 \leq k \leq |T_{i+1,j}|,$

$$L' = L' \cup l_p(t_{i+1,j,k}, \bigcup_{X=1}^{M} t_{i,X,m(X)}$$

where $m(X)$ identifies the tuple in $r_{i,X}$ from which $t_{i+1,j,k}$ was derived.

$$\lambda_D(d_{i+1}) = \langle time, \text{``Cartesian product''}, u \rangle.$$

else the query fails and no provenance links are created.

Figure 3.3(c) shows the provenance link induced for the result relation and tuples of a Cartesian product.

3.6.3.4 Union Operator Provenance

We define provenance for the polyadic union operator.

$\Re(r_{out} = (r_{in1} \bigcup \ldots r_{inM}), u, time) =$

if the union is well-formed and $r_{out} \notin R_i \wedge r_{in1}, \ldots r_{inM} \in R_i$, then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

r_{out} is the query result as defined by the semantics of relation queries.

Let r_{out} be called $r_{i+1,j}$ and r_{inM} be called $r_{i,M}$. Add $r_{i+1,j} to R_{i+1}$.

$$L' = L \cup \bigcup_{X=1}^{M} l_p(r_{i+1,j}, r_{i,X}).$$

$\forall t_{i+1,j,k} \in T_{i+1,j}, 1 \le k \le |T_{i+1,j}|,$

$$L' = L' \cup \bigcup_{X=1}^{M} \{l_p(t_{i+1,j,k}, t_{i,X,m})\} | t_{i+1,j,k} \text{ is identical to } t_{i,X,m}.$$

$$\lambda_D(d_{i+1}) = \langle time, \text{``Union''}, u \rangle.$$

else the query fails and no provenance links are created.

Figure 3.3(d) shows the provenance links induced for the result relation and tuples of a Union.

3.6.4 Provenance for Results of General MMP Queries

$\Re(r_{out} = (\pi\sigma(r_{1,1}^{in} \times \dots r_{1,n1}^{in}) \cup \dots \cup \pi\sigma(r_{M,1}^{in} \times \dots \times r_{M,nM}^{in})), u, time) =$

if the query is well-formed and $r_{out} \notin R_i \wedge \forall r_{X,Y}^{in}, 1 \le X \le M, 1 \le Y \le nM, r_{X,Y}^{in} \in R_i,$

then

$$D' = D \cup d_{i+1}, \text{ where } d_{i+1} = d_i.$$

r_{out} is the query result as defined by the semantics of relation queries.

Let r_{out} be called $r_{i+1,j}$. Add $r_{i+1,j} to R_{i+1}$.

$$L' = L \cup \bigcup_{X=1}^{M} l_p(r_{i+1,j}, \{\bigcup_{Y=1}^{nM} r_{X,Y}^{in}\}).$$

$\forall t_{i+1,j,k} \in T_{i+1,j}, 1 \le k \le |T_{i+1,j}|,$

$$L' = L' \cup \bigcup_{X=1}^{M} l_p(t_{i+1,j,k}, \{\bigcup_{Y=1}^{nM} t_{i,jY,m(Y)}\})$$

where $T_{i,jY}$ are the tuples of $r^{in}_{X,Y}$ and

$$t_{i+1,j,k} \text{ was derived by the query from } \bigcup_{Y=1}^{nM} t_{i,jY,m(Y)}.$$

$$\lambda_D(d_{i+1}) = \langle \text{time, "query statement",u} \rangle$$

where *query statement* is the text of the query, else the query fails and no provenance links are created.

Figure 3.4 shows an example of provenance due to a query. The input relations, A, B, and C, are shown at the top, followed by the query. At the bottom of the figure the query result relation is shown along with the explicit provenance of the result relation and the inherited provenance of all its components. Figure 3.5 shows the graphical representation of the provenance links shown textually in Figure 3.4. The explicit provenance of the result relation is shown in solid lines, while the inherited provenance of tuples, attributes, and attribute values are shown in dashed lines. We read the provenance of the result components of Q as follows:

- the result relation, R, exists because both input relations A and B exist, and exists independently because input relation C exists

- the inheritance rules (defined in Section 3.7, below) define that attribute $alpha$ in result relation R exists because attribute $alpha$ in input relation A exists, and exists independently because attribute $alpha$ in input relation C exists

- tuple $\langle a \rangle$ in the result relation R exists for three independent reasons: because tuple $\langle a, b \rangle$ in relation A exists and tuple $\langle b, c \rangle$ in relation B exists; because tuple $\langle a, s \rangle$ in relation A exists and tuple $\langle s, u \rangle$ in relation B exists; and because tuple $\langle a \rangle$ in relation C exists

- tuple $\langle o \rangle$ in R exists because tuple $\langle o \rangle$ in relation C exists

- the inheritance rules define that attribute value a in column $alpha$ of tuple $\langle a \rangle$ in R exists independently for three reasons: because the attribute value a in column $alpha$ of tuple $\langle a, b \rangle$ in relation A exists; because the attribute value a in column $alpha$ of tuple $\langle a, s \rangle$ in relation A exists; and because the attribute value a in column $alpha$ of tuple $\langle a \rangle$ in relation C exists

- the inheritance rules define that attribute value o in column $alpha$ of tuple $\langle o \rangle$ in R exists because the attribute value o in column $alpha$ of tuple $\langle o \rangle$ in relation C exists.

3.7 Accessing Provenance Information

If the user wishes to see complete provenance for a component, then in addition to provenance links originating at the component of interest, inherited links must be visualized, as must implicit continuity links. Since provenance links for all component types are included in the MMP instance, it may be difficult for users to distinguish provenance connectivity at different levels of granularity. In addition, an MMP instance is 3-dimensional: faces define two dimensions, and the succession of faces (and the provenance links that connect them) defines the third. In order to make provenance more accessible to users, we provide three mechanisms to present or interrogate provenance: provenance graphs, which provide a 2-dimensional graph representation of provenance for a selected component; the provenance query predicates in the MMP language, which allow users to select data by specifying provenance characteristics; and an algebraic representation of component provenance comparable to provenance representations in current literature.

A	
alpha	beta
a	b
a	s

B	
beta	gamma
b	c
s	u

C
alpha
a
o

$$R = \pi_{alpha}\,\sigma_{A.beta=B.beta}(A \times B) \cup C$$

R
alpha
a
o

Explicit provenance induced by query:
Provenance of relation R = $\{I_p(R,\{A,B\}), I_p(R,C)\}$

Provenance of tuple (R,<a>) =
$$\{I_p(t_{R,<a>},\{t_{A,<a,b>},t_{B,<b,c>}\}), I_p(t_{R,<a>},\{t_{A,<a,s>},t_{B,<s,u>}\}), I_p(t_{R,<a>},t_{C,<a>})\}$$

Provenance of tuple (R,<o>)= $I_p(t_{R,<o>},t_{C,<o>})$

Inherited provenance:

Provenance of attribute (R,alpha) = $\{I_p(a_{R,alpha},a_{A,alpha}), I_p(a_{R,alpha},a_{C,alpha})\}$

Provenance of attribute value (R,<a>,alpha)=
$$\{I_p(v_{R,<a>,alpha},v_{A,<a,b>,alpha}), I_p(v_{R,<a>,alpha},v_{A,<a,s>,alpha}), I_p(v_{R,<a>,alpha},v_{C,<a>,alpha})\}$$

Provenance of attribute value (R,<o>,alpha)= $I_p(v_{R,<o>,alpha},v_{C,<o>,alpha})$

Figure 3.4: Example Single-Revision Provenance Resulting from a Query
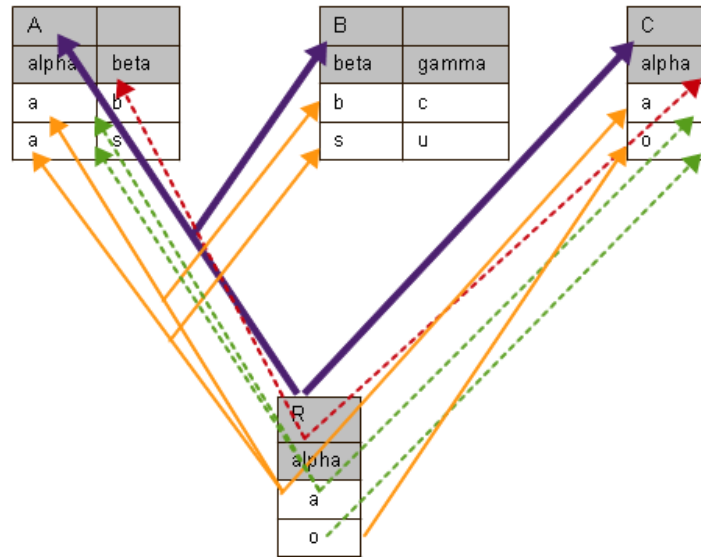
Figure 3.5: Graphical Representation of Single-Revision Provenance

3.7.1 Provenance Graphs

A provenance graph is a directed graph that represents the provenance of an individual component (a relation, a tuple, an attribute, or an attribute value). A provenance graph vertex represents a component or an external source referent. A provenance graph edge represents an explicit or inherited provenance link or an implicit continuity link. Each provenance graph edge is labeled with the name, user, and time of the operation that induced the corresponding provenance link, or is labeled *continuity* if it corresponds to an implicit continuity link. For brevity, we sometimes show examples with provenance graphs where continuity links and predecessor nodes are omitted if they are not pertinent to the example. Construction of a provenance graph $G_p(c_0)$ for a component c_0 in an MMP instance M requires that we find all components and external source referents in M that are ancestors (based on provenance links) or predecessors of c_0 (based on continuity links) and their ancestors and predecessors. We must also find all provenance links that record the actions used in

deriving these components. We find these by beginning at c_0 in the MMP instance and tracing backwards through all explicit provenance links, all inferred continuity links, and all inherited provenance links connected to c_0.

3.7.1.1 Preliminaries: Tracing Continuity and Inheritance

The rules for inferring continuity links from a component in one face to its predecessor component in the immediately preceding face in an MMP instance are as follows. Note that continuity links may only be inferred for components not newly induced by the revision that created a database face.

- The predecessor of a relation in database face d_{n+1} is the relation with the same name in face d_n

- The predecessor of an attribute in database face d_{n+1} is the attribute with the same name in the prior version of its relation

- The predecessor of a tuple in database face d_{n+1} is the unique tuple in the prior version of its relation with identical values for all corresponding attributes

- The predecessor of an attribute value in database face d_{n+1} is the attribute value in the prior version of its tuple for the same attribute

We now define the rules for determining the provenance links inherited by a component from its containing components. As explained in Section 3.7.1.2, the rules are applied to a component c when c originates no provenance links, and the face d containing c is labeled with an operation that has inherited provenance semantics[2], and a

---

[2]The label on d indicates the applied operation and thus tells us whether the operation has inherited provenance semantics.

component that contains c in d originates provenance links[3]. At most one rule from the list below applies to a component c

Recall that we assume that the attributes of each SPX term result in our queries are renamed to a common set of names prior to applying the union operation. Let the set of inherited provenance links for component c in face d_n be called $L_{inherited}(c)$.

1. If a tuple has a containing relation with an explicit provenance link to another relation, then that tuple inherits a link:

   If $c = t_{n,p,k} \in T_{n,p} \wedge \exists l_p(r_{n,p}, r_{n-1,q})$ then

   $L_{inherited}(c) = l_p(t_{n,p,k}, t_{n-1,q,m})$ where $t_{n,p,k}$ is identical to $t_{n-1,q,m}$.

2. If a tuple has a containing relation with an explicit provenance link to an external source referent, then that tuple inherits a link:

   If $c = t_{n,p,k} \in T_{n,p} \wedge \exists l_p(r_{n,p}, s)$ then

   $L_{inherited}(c) = \{l_p(t_{n,p,k}, s)\}$

3. If an attribute has a containing relation with an explicit provenance link to another relation, then that attribute inherits one or more links. That is, each attribute exists independently because of its parent attribute in each ancestor relation:

4. If $c = a_{n,p,l} \in A_{n,p} \wedge \exists l_p(r_{n,p}, r_{n-1,q})$ then

   $L_{inherited}(c) = \emptyset$.

   $\forall l_p(r_{n,p}, \{r_{n-1,q}\}), L_{inherited}(c) = L_{inherited}(c) \cup$

   $\{l_p(a_{n,p,l}, a_{n-1,q,r}) | \exists r$ where $a_{n,p,l}$ and $a_{n-1,q,r}$ have the same name $\}$

---

[3] Components in d that originate provenance links are those directly affected by the operation that created d. These can be efficiently identified by examining the text of the operation in the face label: the operation name indicates its target component type; the text of the operation argument list identifies the target component; the semantics of the operation indicates which components contained by the target component, if any, have explicit provenance links induced by the operation. We sometimes omit the operation argument list in our diagrams.

5. If an attribute has a containing relation with an explicit provenance link to an external source referent, then that attribute inherits a link:

    If $c = a_{n,p,l} \in A_{n,p} \wedge \exists l_p(r_{n,p}, s)$ then

    $L_{inherited}(c) = l_p(a_{n,p,l}, s)$

6. If an attribute value has a containing tuple that has no explicit provenance, but has a containing relation with provenance links to other relations, then that attribute value inherits one or more links:

    If $c = v_{n,p,k,l} \in V_{n,p,k} \wedge \neg\exists l_p$ originating at $t_{n,p,k} \wedge \exists l_p(r_{n,p}, r_{n-1,q})$ then

    $L_{inherited}(c) = \emptyset$.

    $\forall l_p(r_{n,p}, r_{n-1,q})$,

    $L_{inherited}(c) = L_{inherited}(c) \cup \{l_p(v_{n,p,k,l}, v_{n-1,q,m,r})|\exists r$

    where $a_{n,p,l}$ and $a_{n-1,q,r}$ have the same name $\}$

    $\wedge t_{n,p,k}$ is identical to $t_{n-1,q}$

7. If an attribute value has a containing tuple that has no explicit provenance, but has a containing relation with a provenance link to an external source referent, then that attribute value inherits a link:

    If $c = v_{n,p,k,l} \in V_{n,p,k} \wedge \exists l_p(r_{n,p}, s) \wedge \neg\exists l_p$ originating at $t_{n,p,k}$ then

    $L_{inherited}(c) = l_p(v_{n,p,k,l}, s)$

8. If an attribute value has a containing tuple that has explicit provenance links to other tuples, then that attribute value inherits one or more links:

    If $c = v_{n,p,k,l} \in V_{n,p,k} \wedge \exists l_p(t_{n,p,k}, \{t_{n-1,q,m}, \text{ where } 1 \leq m \leq |T_{n+1,q}|\})$ then

    $L_{inherited}(c) = \{l_p(v_{n,p,k,l}, v_{n-1,q,m,r})|$

    $\exists r$ such that $a_{n,p,l}$ and $a_{n-1,q,r}$ have the same name $\}$

9. If an attribute value has a containing tuple that has an explicit provenance link to an external source referent, then that attribute value inherits a link:

    If $c = v_{n,p,k,l} \in V_{n,p,k} \wedge \exists l_p(t_{n,p,k}, s)$ then $L_{inherited}(c) = l_p(v_{n,p,k,l}, s)$
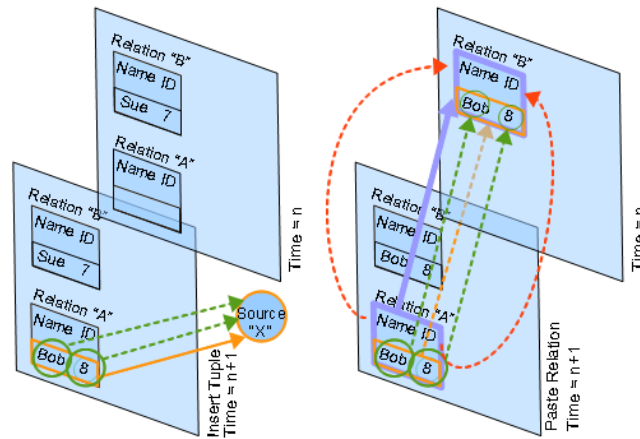
Figure 3.6: Examples of Inherited Provenance

Figure 3.6 shows, using dotted lines, examples of inherited provenance. On the left, a tuple in relation A was inserted at time $n + 1$ from external source X via an Insert Tuple operation. The provenance link from the result tuple to the external source is explicitly recorded in the model. The provenance links inherited by the attribute values in the tuple are shown in dotted lines. On the right in the figure, relation A is pasted into the database at time $n + 1$ as a copy of relation B. The provenance link (shown solid) from the relation A to relation B is explicitly recorded. The provenance links inherited by attributes, tuples, and attribute values, are shown dotted.

3.7.1.2 Defining Provenance Graphs

With these rules for inference and inheritance defined, we now formally define provenance graphs. Let an MMP instance M have components C, external source referents S, and provenance links L. Let $c_0 \in C$ be the component (in the most recent face) for which we wish to construct a provenance graph $G_p(c_0)$ with vertices V and edges E. We define m, an injective mapping $m : V \to C \cup S$ and we define V and E for $G_p(c_0)$ as follows:

1. $V = \emptyset, E = \emptyset$.

2. distinguished component: $v_{initial}$ is created and added to V, and we set $m(v_{initial}) = c_0$.

3. explicit provenance links and connected components: $\forall v \in V, \forall l_p(m(v), B) \in L \Rightarrow \forall b \in B$, we add a new vertex v' to V, and we set $m(v') = b$, and we add to E an edge
$e(v, \bigcup_{X=1}^{|B|} m(v_X))$.

4. continuity links and connected components: $\forall v \in V, \exists$ predecessor c' for $m(v)$, then add to V a vertex v' and set $m(v') = c'$, and add to E an edge $e(v, v')$.

5. inherited provenance links and connected components: for all v in V, if $m(v)$ originates no provenance links, and $\lambda_D(d_i)$ where d_i contains $m(v)$ indicates an operation with inherited provenance semantics[4], and a component that contains $m(v)$ in d_i originates provenance links, then use the inheritance rules from Section 3.7.1.1 to find the set of provenance links inherited by $m(v)$, which we call $L_{inherited}(m(v))$. For each $l \in L_{inherited}(m(v))$: add to V a set of vertices V' consisting of one vertex v' for each terminal component c' of l; for each v', set $m(v') = c'$; and add to E a hyper-edge $e(v, V')$.

6. nothing else is in V or E

When each v is placed in V, it is labeled with the type of component of $m(v)$ (relation, attribute, tuple, attribute value, or source), a name for $m(v)$ that indicates its relation, attribute, tuple, and value, as appropriate, the relation to which $m(v)$ belongs, if appropriate, the value of $Expired(m(v))$, the timestamp portion of $\lambda_D(d)$, where d containst $m(v)$, and a value. If the type of $m(v)$ is relation, attribute, source,

---

[4]Insert Tuple, Drop Tuple, Paste Tuple, Paste Relation, and queries have inherited provenance semantics. Create Relation, Create Source, Create Attribute, Drop Relation, Drop Attribute, Insert Value, Drop Value, Paste Value, Confirm Value, and Doubt Value do not have inherited provenance semantics.

or tuple, the value is $NULL$. If $m(v)$ is an attribute value, the value is the value of $m(v)$. If the type of $m(v)$ is source, the relation label is $NULL$. We refer to these labels as $type, name, relation, expired, time,$ and $value$, respectively. In addition, when each e is placed in E, it is labeled with $\lambda_D(d)$, where d contains $m(v)$, where v is the originating vertex of e, or is labeled *continuity*, if e represents a continuity link. If e is labeled with $\lambda_D(d)$, we refer to the portions labels as $op, user,$ and $time$, respectively.

Figure 3.7 shows the provenance graph for the attribute value from attribute ID in tuple 1 in relation C at time $n + 6$ from the example in Figure 3.8. The boxes shown delineate the operations performed on the ancestors of the component, and are labeled on the right with associated timestamps. The value of *user* is omitted on all edges in the figure for brevity. As before, the dot notation used for vertex name properties in the graph indicates relation first, then a shorthand identifier for the tuple, then attribute, then attribute value. The timesteps associated with links are shown on the right of the figure. We abbreviate the description of operations attached to each graph edge. Note the use of inherited links. For example, the top link on the left of the graph represents the insertion of the parent relation A. This link is inherited from the attribute value's parent relation. We envision provenance graphs as a visualization presented to a user for the purpose of browsing provenance of a user-selected component.

3.7.2 Querying Provenance

In Chapter 2, we defined the grammar of predicates for selecting rows and columns of data in relations by characteristics of their provenance, or characteristics of the provenance of attribute values they contain. In this section, we define when a predicate qualifies a table row or column for output, and how predicate terms are evaluated. Refer to Table 3.1, reproduced here from Chapter 2, for the syntax of our
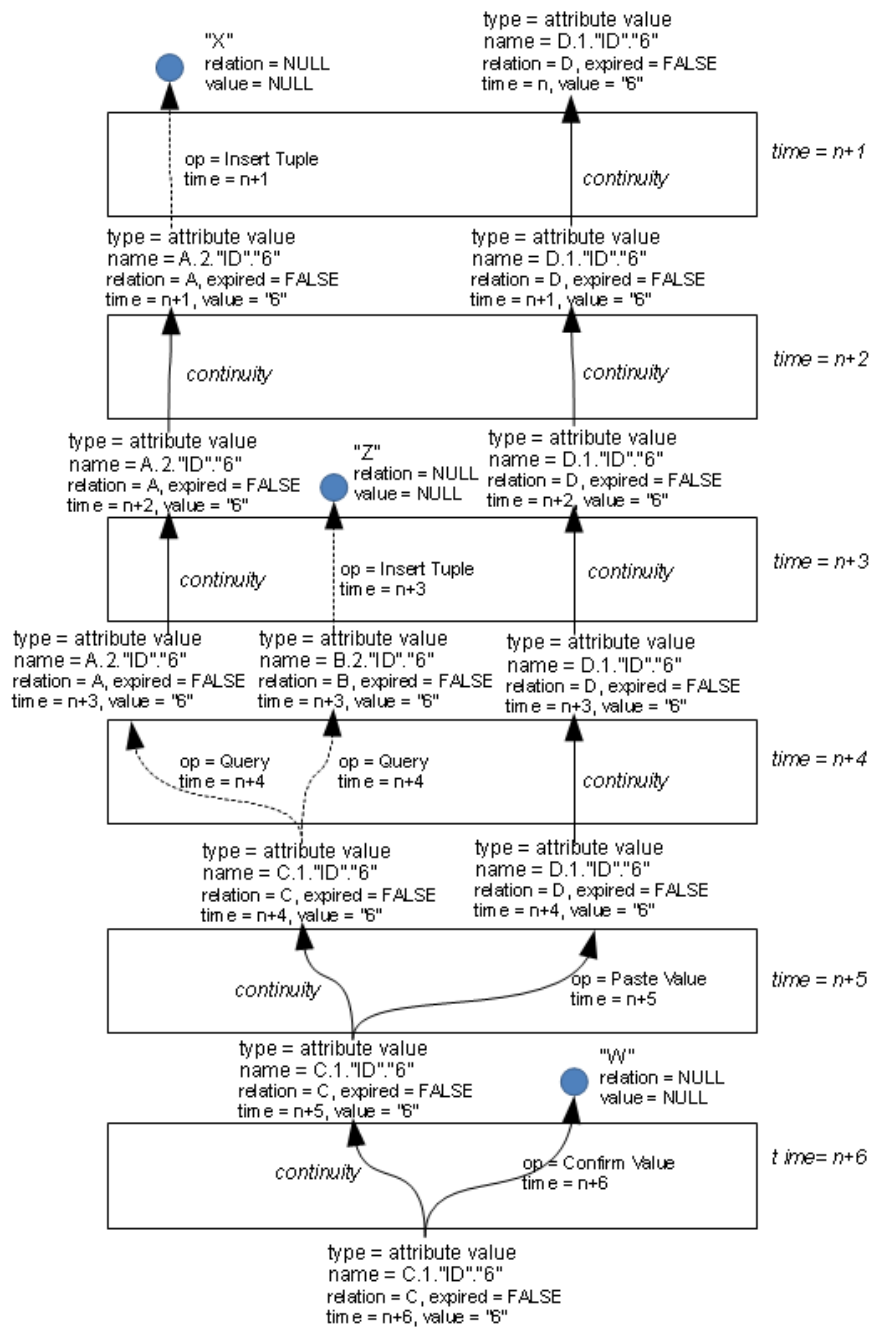
Figure 3.7: Example Provenance Graph
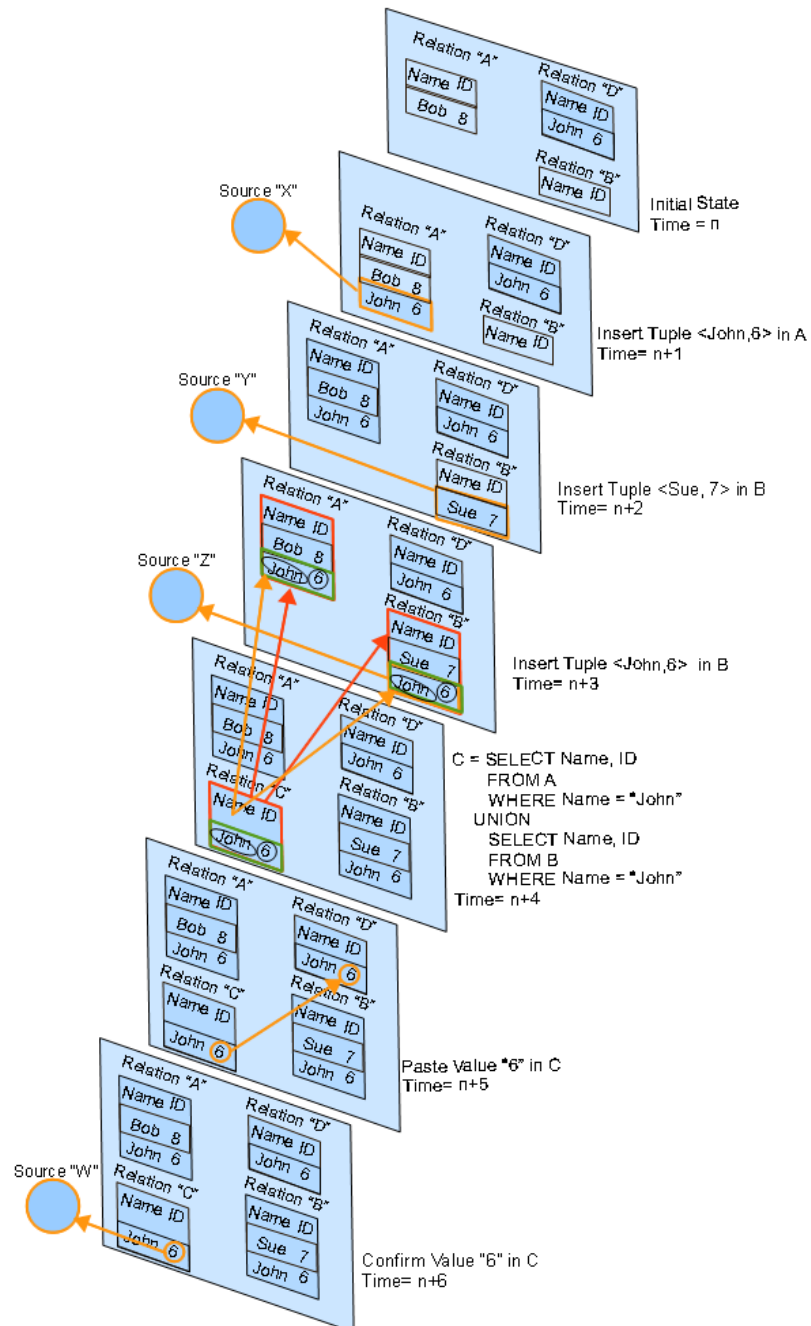
Figure 3.8: MMP Instance showing provenance links between components

predicate language. We define here the semantics of selectionPredicates. ProjectionaltionPredicates have similar semantics, the difference being that they apply to the relational projection operator in MMP, and they qualify columns for output instead of tuples.

Let M, an MMP instance with current face d_i, have relation $r_{i,j} \in R_i$, the relation set of d_i. Let $\{t_{i,j,k}\}, 1 \leq k \leq |T_{i,j}|$ be the tuples in $r_{i,j}$. Let $\Re(D, S, L, r_{out} = \sigma_P(r_{i,j}), u, t)$ be a revision applied to M, where P is a selectionPredicate in the MMP language[5]. P consists of component specifier CS, along with one or more predicateQualifiers Q connected by logical operators AND and OR. This expression of predicateQualifiers Q and their connecting logical operators we call the $Qgroup$ of P.

As shown in Table 3.1, each predicateQualifier Q in a $Qgroup$ consists of one or more pathQualifiers p_n^Q connected by logical operators AND and BEFORE. Each pathQualifier p_n^Q describes the characteristics of a provenance path[6]. In order to describe a path, p_n^Q contains vertex descriptors V_n^Q and edge descriptors E_n^Q that describe characteristics of vertices and edges in the path, and O_n^Q, a poset that describes orderings (by time value) of these vertices and edges as specified by "BEFORE" clauses in p_n^Q. Each cQualset in p_n^Q describes a vertex in the path, so each cQualset defines a vertex descriptor $v_n^Q \in V_n^Q$. Similarly, each sQualset in p_n^Q describes a vertex in the path, and hence also defines a vertex descriptor $v_n^Q \in V_n^Q$. Each aQualSet in p_n^Q describes an edge in the path, so each aQualset defines an edge descriptor $e_n^Q \in E_n^Q$. Each vertex descriptor and edge descriptor specifies one or more conditions under which it *matches* a vertex or edge, respectively, in a provenance path. These conditions are called *constraints*, and are defined as follows:

---

[5]If a predicate of σ includes a combination of MMP provenance selection predicates and the usual relational selection predicates, each selectionPredicate is evaluated for each tuple, and the resulting boolean valuation is combined in the usual way with the other predicates.

[6]A provenance path is a path in the provenance graph of a component.

selectionPredicate ::= *componentSpecifier predicateQualifier*
componentSpecifier ::= TUPLE HAS ⟨predicateQualifier⟩
|SOME DATA VALUE 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⟩ (⟨cQualSet⟩)
|AN OPERATION (⟨aQualSet⟩)
|A SOURCE (⟨sQualSet⟩)
| ⟨pathQualifier⟩ [BEFORE|AND] ⟨pathQualifier⟩

aQualSet ::= ⟨aQual⟩ | ⟨aQual⟩ [AND|OR] ⟨aQualSet⟩

cQualSet ::= ⟨cQual⟩ | ⟨cQual⟩ [AND|OR] ⟨cQualSet⟩

sQualSet ::= ⟨sQual⟩

aQual ::= WITH ACTION = ⟨constant⟩
| WITH ACTION = ANY QUERY
| BY USER = ⟨username⟩
|
WHERE TIME ⟨cCmp⟩ ⟨timestamp⟩

cQual ::= IN RELATION ⟨relname⟩
| WITH A VALUE ⟨cCmp⟩ ⟨compval⟩
| WHERE EXPIRED = ⟨TRUE|FALSE⟩

sQual ::= WITH NAME = ⟨constant⟩

component ::= tuple | attribute | value

cCmp ::= $=$ | $>$ | $<$ | $\geq$ | $\leq$ | $\neq$

Table 3.1: Syntax of MMP Provenance Predicate Language (Repeated from Table 2.3)

1. For each "IN RELATION $\langle relationName \rangle$" qualifier in a cQualset cQ_n^Q in p_n^Q, the corresponding v_n^Q has constraint $relation = \langle relationName \rangle$.

2. For each "WITH A VALUE $\langle cCmp \rangle$ $\langle comparisonValue \rangle$" qualifier in a cQualset cQ_n^Q in p_n^Q, the corresponding v_n^Q has constraint $value \langle cCmp \rangle \langle comparisonValue \rangle$.

3. For each "WHERE EXPIRED = $\langle booleanValue \rangle$" qualifier in a cQualset cQ_Q, the corresponding v_n^Q has constraint $expired = \langle booleanValue \rangle$.

4. For each "WITH NAME = $\langle sourceName \rangle$" qualifier in an sQualset sQ_n^Q, the corresponding v_n^Q has constraint $name = \langle sourceName \rangle$

5. For each "WITH ACTION = $\langle actionName \rangle$" qualifier in an aQualset aQ_n^Q, the corresponding e_n^Q has constraint $op = \langle actionName \rangle$

6. For each "BY USER = $\langle userName \rangle$" qualifier in an aQualset aQ_n^Q, the corresponding e_n^Q has constraint $user = \langle userName \rangle$.

7. For each "WHERE TIME $\langle cCmp \rangle \langle timestamp \rangle$" qualifier in an aQualset aQ_n^Q, the corresponding e_n^Q has constraint $time \langle cCmp \rangle \langle timestamp \rangle$

If a Qualset Q_{n1}^Q is followed in p_n^Q by "BEFORE"[7] and then a Qualset Q_{n2}^Q, then O_n^Q contains the v_{n1}^Q (e_{n1}^Q, if Q_{n1}^Q is an aQualset) corresponding to Q_{n1}^Q and the v_{n2}^Q (or e_{n2}^Q, if Q_{n2}^Q is an aQualset) corresponding to Q_{n2}^Q, which represents that v_{n1}^Q(or e_{n2}^Q) $\prec$ v_{n2}^Q(or e_{n2}^Q).

Each tuple $t_{i,j,k}$ evaluated by $\sigma_P(r_{i,j})$ is selected for output if the expression $Qgroup$ from P evaluates to TRUE for $t_{i,j,k}$. $Qgroup$ is TRUE for $t_{i,j,k}$ if the expression formed by its predicateQualifers Q and connecting logical operators evaluates to TRUE for $t_{i,j,k}$, and is FALSE otherwise. Each predicateQualifier $Q \in Qgroup$ is

---

[7]The BEFORE binary relationship indicates ordering in time of two Qualsets. In particular, it indicates that the first Qualset in the relationship must occur earlier in the timeline than the second.

evaluated over a set of components $C^t_{i,j,k}$ defined for $t_{i,j,k}$ as follows: If CS uses the language "TUPLE HAS", then $C^t_{i,j,k} = t_{i,j,k}$; otherwise $C^t_{i,j,k} = V_{i,j,k}$. Q evaluates to TRUE if any path in the provenance graph $G_p(c)$ of any component $c \in C^t_{i,j,k}$ satisfies the logical expression formed by its pathQualifiers p^Q_n and their connecting logical operations. p^Q_n is TRUE for path p if each vertex descriptor $v^Q_n \in V^Q_n$ matches a vertex $v_p \in V_p$ and each edge descriptor $e^Q_n \in E^Q_n$ matches an edge $e_p \in E_p$ and if the relative ordering of any pair of matching vertices or edges in p satisfies the order of their corresponding vertex and edge descriptors in O^Q_n, else p^Q_n is FALSE for p.

Vertex descriptor v^Q_n matches vertex v_p if all constraints in v^Q_n are met by v_p. A constraint of a vertex descriptor v^Q_n is met by vertex v_p if and only if the property named in the constraint exists for v_p and its value satisfies the constraint. Edge descriptor e^Q_n matches edge e_p if all constraints in e^Q_n are met by e_p. A constraint of an edge descriptor e^Q_n is met by edge e_p if and only if the property name in the constraint exists for e_p and its value satisfies the constraint.

3.7.2.1 Example of Provenance Predicate Evaluation

As an example of evaluating a selectionPredicate, consider the example relation and its associated history shown in Figure 3.8. The provenance graphs for the attribute values of the only tuple in relation C in the figure at time n+6 are shown in Figure 3.7 and Figure 3.9. We omit the $user$ property from edges in the figure. We indicate vertices with type *source* with circles. Consider the query $\sigma_P(C)$, where P is the selectionPredicate "SOME DATA VALUE IN TUPLE HAS A PATH WITH A VALUE IN RELATION D BEFORE AN OPERATION WITH ACTION = 'Paste Value' ". We evaluate P for each tuple in C; in this case, only one tuple t, $\langle$"John","6"$\rangle$ exists to be evaluated for selection.

First, form the set of paths to be considered for t. The component specifier CS in P is "SOME DATA VALUE IN TUPLE HAS ", so the set of paths includes all paths

from the provenance graphs of all attribute values v in t. In this example, there are two such attribute values, "John" and "6", so the set of paths to evaluate includes all paths from the two provenance graphs shown in Figure 3.7 and Figure 3.9. Call these g_{John} and g_6.

The $Qgroup$ of P consists of a single predicateQualifier Q that has a single pathQualifier p_Q,

"A PATH WITH A VALUE IN RELATION D BEFORE AN OPERATION WITH ACTION = 'Paste Value' "

Thus p_Q consists of the cQualset, "A VALUE IN RELATION D", the ordering term BEFORE, and the aQualset, "AN OPERATION WITH ACTION = Paste Value". Then p_Q has one vertex descriptor v, $\{relation = D\}$ and one edge descriptor e, $\{op = PasteValue\}$. Because BEFORE connects the corresponding Qualset pair, $O = \{v, e\}$, which means that $v \prec e$.

Consider Figure 3.9. No vertex in g_{John} matches the vertex descriptor $\{relation = D\}$, so g_{John} does not qualify tuple 1 from relation C for output. Now consider Figure 3.7. In g_6, all vertices named $D.1."ID"."6"$ satisfy v. In the same path as all of these (the rightmost path in the figure), the edge with timestamp $n + 5$ satisfies e. Thus both descriptors are satisfied by this path. In addition, the edge matching e has associated timestamp n+5, and at least one vertex matching v has timestamp less than n+5 (for example, the terminal vertex of the edge matching e matches d and has timestamp n+4). Thus the ordering constraint is also satisfied by this path. As a result, P is satisfied by tuple 1 from relation C, and so tuple 1 is selected for output.

3.7.3 Provenance Polynomials

While MMP represents provenance using provenance links between components, other provenance models in the literature [3], [4], [6], [12], [19] represent provenance by annotating relational tuples with semi-ring expressions representing their single-

"X"
relation = NULL
value = NULL

op = Insert Tuple
time = n+1

t = n+1

type = attribute value
name = A.2."ID"."John"
relation = A, expired = FALSE
time = n+1, value = "John"

continuity

t = n+2

type = attribute value
name = A.2."ID"."John"
relation = A, expired = FALSE
time = n+2, value = "John"

"Z"
relation = NULL
value = NULL

continuity

op = Insert Tuple
time = n+3

t = n+3

type = attribute value
name = A.2."ID"."John"
relation = A, expired = FALSE
time = n+3, value = "John"

type = attribute value
name = B.2."ID"."John"
relation = B, expired = FALSE
time = n+3, value = "John"

op = Query
time = n+4

op = Query
time = n+4

t = n+4

type = attribute value
name = C.1."ID"."John"
relation = C, expired = FALSE
time = n+4, value = "John"

continuity

t = n+5

type = attribute value
name = C.1."ID"."John"
relation = C, expired = FALSE
time = n+5, value = "John"

continuity

t = n+6

type = attribute value
name = C.1."ID"."John"
relation = C, expired = FALSE
time = n+6, value = "John"

Figure 3.9: Provenance Graphs for Attribute Values of Relation C at time t+6 in Figure 3.8

.

Input relation R

Tuple ID	A	B	C
a	1	5	8
b	3	2	9
c	1	6	9

$$\pi_{AC}(R \bowtie_A \rho_{B \to D, C \to E}(R)) \cup (R \bowtie_C \rho_{A \to D, B \to E}(R))$$

Query
Result

Provenance
Representations

A	C	Lineage	CPDB	Trio	Orchestra
1	8	{a,c}	{{a},{a,c}}	$2a + ac$	$2a^2 + ac$
1	9	{a,b,c}	{{c},{a,c},{b,c}}	$2c + ac + bc$	$2c^2 + ac + bc$
3	9	{b,c}	{{b},{b,c}}	$2b + bc$	$2b^2 + bc$

Figure 3.10: Examples of Provenance Expressions from Current Models

.

generation provenance [20]. These semi-ring expressions are either set-theoretic (expressed as sets of tuple identifiers, or sets of sets of tuple identifiers) or algebraic (expressed as polynomials where the variables are tuple identifiers). Examples are shown in Figure 3.10. In current models, these expressions are stored as text strings. Although one goal of MMP is to avoid the need for users to parse such representations in order to query provenance, we provide here a similar representation as a part of MMP in order to make our provenance system comparable to others in the literature. In this section, we define our algebraic representation for provenance, which extends those from the literature in these ways:

- We define algebraic representations of provenance at all granularities (relations, attributes, tuples, and attribute values), instead of only at the tuple level.

- Our algebraic expressions represent multi-generation provenance instead of single-generation provenance.

- We include operations performed, identity of users performing them, and time at which they were performed.

- Our expressions represent provenance due to DDL, DML, and query operations.

First, we define algebraic expressions without representing operations, users, and timestamps, and then we explain our first two extensions.

Let C be the set of all components in an MMP instance M. Let S be the set of all external source referents in M. Let $V = C \cup S$. Define a set of variables I and a bijection $componentToVar : V \to I$. We define $Prov^{SN}$ to be a semiring $(I, +, \bullet, 0, 1)$, where + is algebraic addition and $\bullet$ is algebraic multiplication. Provenance of $c \in C$ is represented by a polynomial expression in $Prov^{SN}$ where + represents that any of its arguments alone gives rise to c, and $\bullet$ represents that all of its arguments together give rise to c. For example, if we represent the provenance of c as $x_1 \bullet x_2 + x_3$, for $c \in C$ and x_1, x_2, and $x_3 \in I$, then c is present in I because both x_1 and x_2 were present as inputs to an operation that had c as output, and is independently present because x_3 was present as an input to a (possibly distinct) operation that gave rise to c.

Let K be the set of constants, if any, introduced by queries that have previously run on M. Let C_{stop} be a set of components of the same type as c in M, specified by the user as beyond which no provenance should be represented. Let c' be the predecessor, if one exists, of component c. Let c originate N provenance links, $\{l_p1(c, B_1), \ldots l_pN(c, B_N)\}$, where link $l_pX, 1 \leq X \leq N$ has a terminal at each $b_{X,Y} \in B_X, 1 \leq Y \leq |B_X|$. Then for a distinguished component c in face d_n of M,

$$Prov^{SN}(c) =$$

$$
\begin{cases}
componentToVar(c), & \text{if } c \in S \cup K \cup C_{stop} \\
\left(\sum_{X=1}^{N}(\prod_{Y=1}^{|B_X|} Prov^{SN}(b_{X,Y}))\right) + (Prov^{SN}(c')), & \text{if } c \notin S \cup K \cup C_{stop} \\
& \text{and } c' \text{exists} \\
\left(\sum_{X=1}^{N}(\prod_{Y=1}^{|B_X|} Prov^{SN}(b_{X,Y}))\right), & \text{otherwise}
\end{cases}
$$

Here, summation indicates the $+$ operation in $Prov^{SN}$, and multiplication indicates the $\times$ operation in $Prov^{SN}$. This definition recursively traces the single-generation provenance of c, then traces the provenance of those ancestors, and so on. Recursion stops when original sources, or constants induced by queries, or stopping points specified by the user are encountered. By including the option for user-specified stopping points, we can represent as many generations of a component's provenance as the user wishes to see. If C_{stop} includes all components in the face preceding the one where c first appears, then $Prov^{SN}(c)$ is the single-generation provenance of c, and so is comparable to most provenance representations from the literature. If $C_{stop} = \emptyset$, then $Prov^{SN}(c)$ is the complete multi-generation provenance of c, which traces back every provenance path to a query constant or an external source.

As an example, let x be the referent for external source X, z be the referent for external source Z, w be the referent for external source W, and d be an alias for the attribute value $(D, 1, ID)$ which we assume for this example to be a constant induced by a previous query. Let $S = \{x, y, w\}$ and $K = \{d\}$. Then for the attribute value $(C, 1, ID,'' 6'')$ at $time = n + 6$ (with provenance graph shown in Figure 3.11),

$$
Prov^{SN}((C, 1, ID)) = n1 + w
$$

$$
= (n2 + n3) + w
$$

$$
= (n4 + n5) + n6 + w
$$

$$
= n7 + z + n8 + w
$$

$$
= n9 + z + n10 + w
$$
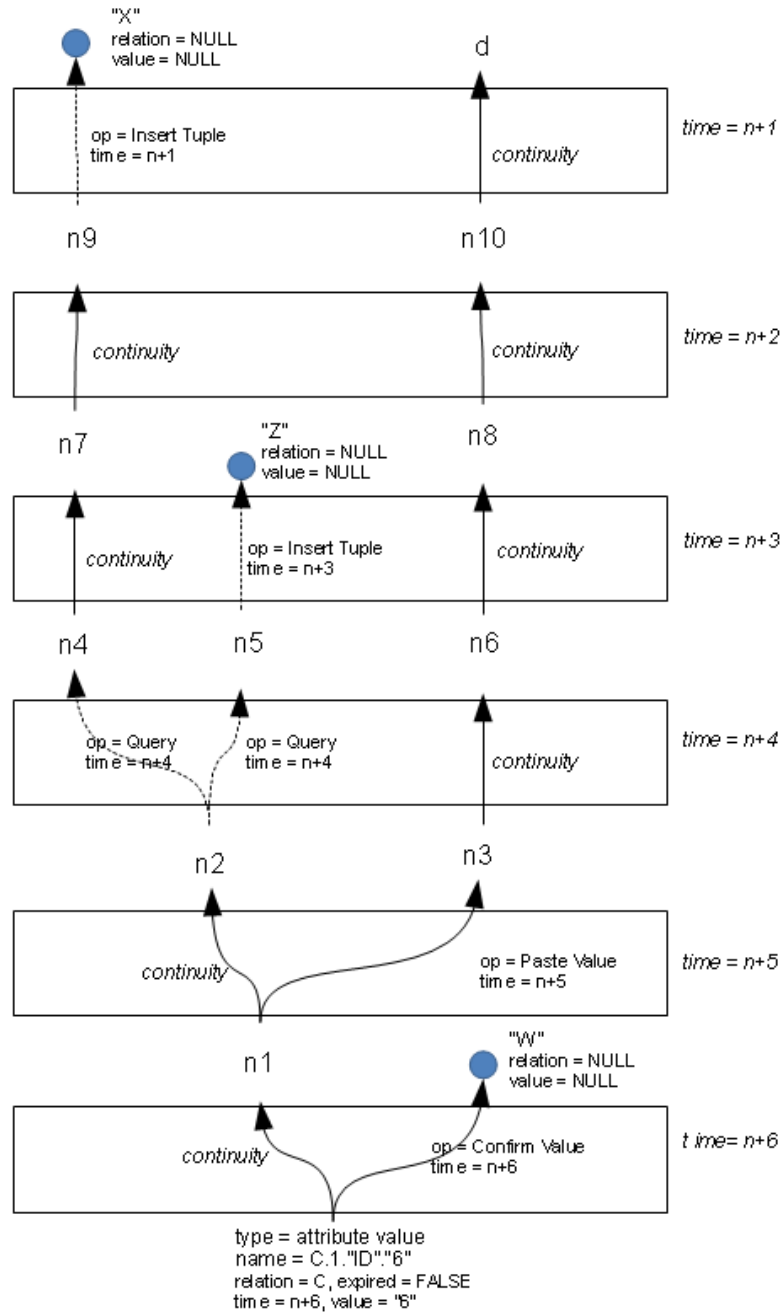
Figure 3.11: Example Provenance Graph. Repeated from Figure 3.7, with vertex descriptions replaced by representative names.

$$= x + z + d + w$$

Although the definition of $Prov^{SN}(c)$ is recursive, the recursive expansion of $Prov^{SN}$ expressions always terminates in polynomial time, because:

- causality ensures that all provenance graphs are acyclic, that is, no component can be derived from itself, so no component can have provenance that includes itself;

- traversal always follows the indicated direction of the directed edges in our graphs, (i.e., from d_{n+1} to d_n); and

- no provenance links originate from external source or query constant vertices.

Note that our definition of $Prov^{SN}$ applies to all component types in instances of the MMP model. Thus we can express as many generations of provenance as desired for a selected relation, attribute, tuple, or attribute value.

3.7.3.1 Representing Operations in Provenance Polynomials

In addition to representing ancestor components in provenance expressions, it is also useful to represent information about the operations used to derive the data, the users applying the operations, and the times when they were applied. We extend $Prov^{SN}$ to include this additional information.

First, we augment the set of identifiers in our semi-ring by including variables that range over the set of operations, users, and timestamps. Recall that $L = OpD \times U \times TS$, and that L is finite. We define a set of coefficients in $Prov^{SN}$, τ_{il}, such that there is a bijection $labelToVar : L \leftrightarrow \tau_{il}$:

$$Prov^{SN}(c) =$$

$$
\begin{cases}
componentToVar(c), & \text{if } c \in S \cup K \cup C_{stop} \\[2ex]
\big(\sum_{X=1}^{M}\big(\prod_{Y=1}^{|B_X|} Prov^{SN}(b_{X,Y})\big)labelToVar(\lambda_D(d_n)))) & \\
\qquad\qquad +(Prov^{SN}(c')), & \text{if } c \notin S \cup K \cup C_{stop} \\
& \text{and } c' \text{ exists} \\[2ex]
\sum_{X=1}^{M}\big(\prod_{Y=1}^{|B_X|} Prov^{SN}(b_{X,Y})\big)labelToVar(\lambda_D(d_n)), & \text{otherwise}
\end{cases}
$$

d_n in the above expressions is the face in D that contains the component $b_{X,Y}$. Simply stated, each step through an ancestor $b_{X,Y}$ of c induces a coefficient $labelToVar(\lambda_D(d_n))$ that represents the operation applied at that step.

As an example, let *Joe* be the user that applied all the operations in Figure 2. Let

- τ_1 be the value from τ_{il} that represents

  ((Insert Tuple(A, $\langle Name = John, ID = 6\rangle$, X), Joe, n+1)

- τ_2 represent (Insert Tuple(B, $\langle Name = John, ID = 6\rangle$,Z), Joe, n+3)

- τ_3 represent ((C = SELECT Name, ID...), Joe, n+4)

- τ_4 represent (Paste Value(D,1,ID,C,1,ID), Joe, n+5)

- τ_5 represent (Confirm Value(C,1,ID,W), Joe, n+6)

The example provenance expression shown above now becomes

$$
Prov^{SN}((C,1,ID)) = (((x\tau_1)(z\tau_2)\tau_3) + d\tau_4) + w\tau_5)
$$

where juxtaposition represents the $\bullet$ operator.

3.7.3.2 Evaluating Plurality of Support with Provenance Polynomials

The recursive substitution of component identifiers with their provenance results in polynomials that include variables that represent only external sources, query-generated constants, user-identified stopping points, and the operations performed in deriving data from these sources. By providing valuations of zero or one for each

such variable in an expression, a user obtains an integral value representing the plurality of support for the component at the root of the provenance graph. This idea was originally proposed by Green [21]. We extend it to allow valuations on the τ coefficients introduced above. Consider the example shown above. Suppose initially that we trust all operations, as well as all external sources used. This situation corresponds to a valuation of all terminal components and τ variables as one. The resulting value of the $Prov^{SN}$ expression is then three. This valuation tells us that there are three independent, trusted derivations that give rise to $(C, 1, ID)$. Now suppose we learn that external source W is not trustworthy, the attribute value at $(D, 1, ID)$ is incorrect, and Joe's work at time t_{n+1} was unreliable. Valuing w, d, and τ_1 as zero yields a $Prov^{SN}$ value of zero. This valuation tells us that given these new developments, data $(C, 1, ID)$ has no support.

3.7.4 Chapter Summary

The conceptual model defined here models both data and its provenance. Data is represented in the familiar relational structure, and provenance relationships are indicated by directed hyper-edges connecting components. Each operation on data creates a new database so that users can see the evolution of database contents over time. Each component in each new database has either continuity to its predecessor component, or provenance tracing back to ancestor components as a result of the applied operation, or both (e.g., when a tuple is re-inserted), or neither (e.g., when a new relation is created). Access to data is provided by a language analagous to both SQL and relational algebra. Additions to traditional predicates for selecting data in relational algebra provide a means to use provenance as a selection criteria. A graphical view of data and its provenance is provided in order to facilitate user browsing or provenance. An algebraic representation of provenance is provided to enable comparison of a subset of our model against other provenance models in the

literature.

Chapter 4

Conceptual Model Evaluation

In this chapter, we evaluate MMP, our conceptual model for data and provenance. We begin by evaluating MMP with two subjective comparisons. In Section 4.1, we compare the capabilities of MMP to the gaps in current literature outlined in Chapter 1, to see what, if anything, MMP contributes that helps to fill those gaps. In Section 4.2, we compare the capabilities of MMP to the needs discussed in the settings from Chapter 1, to see how useful MMP may be in practice.

Next, we provide an objective comparison of MMP against other models for data and provenance in the literature. In Section 4.3, we compare the expressive power of the algebraic provenance representation from the MMP model to that of other models, following an approach developed by Green [20] to compare his provenance polynomials to other models in the literature. In Section 4.4, we contribute a taxonomy for subclasses of an important class of provenance queries identified by He and Singh [23]. We then compare the subclasses of queries expressible in the language of MMP to the subclasses expressible in the languages of other models in the literature. For one sub-class of provenance queries expressible by MMP as well as other models in the literature, we contribute a benchmark of provenance-related queries. In Section 4.5, we compare MMP to other models with regard to expressiveness of provenance graphs. We also compare MMP to other models with regard to the complexity of exploiting provenance to detect where data is used as input to

later derivations. In Section 4.6, we use the benchmark from Section 4.4 to compare the relative complexity of queries expressed in the languages of these models. We do this using a software engineering metric to measure complexity of the semantically same queries expressed in the languages of the models we compare. We assume that a lower measure of query complexity for a semantically identical query is preferable for users.

4.1 Evaluating MMP Against Gaps in the Literature

In Chapter 1, we identified five significant gaps left unaddressed by current models for provenance and relational data in the literature: 1) current models do not model provenance resulting from a mix of DDL, DML, and query operations; 2) in current models, users must parse and interpret each provenance representation manually; 3) in current models, users must assemble multi-generation provenance manually before querying or browsing it; 4) query languages used in current models are designed for relational data, and so are not well-suited to phrase queries over provenance; and 5) current models do not distinguish provenance from data in order to provide suitable management for provenance.

With regard to Gap 1, MMP models provenance for all operations except those DDL operations, for example Create Relation, that do not have meaningful provenance semantics.

MMP provides several mechanisms to address Gaps 2, 3, and 4. Provenance graphs show an intuitive representation that requires no user parsing or reconstruction. For provenance queries, the MMP predicate language allows users to describe the characteristics of provenance that are required for their query. We then use this description to compare against provenance information stored in MMP. In neither case do users need to interpret or parse symbolic representations of provenance when using MMP. In addition, users do not need to re-assemble successive generations of

Input relation R

A	B	C
1	5	8
3	2	9
1	6	9

$$\pi_{AC}(R \bowtie_A \rho_{B \to D, C \to E}(R)) \cup (R \bowtie_C \rho_{A \to D, B \to E}(R))$$

Query Result		Provenance Representations			
A	C	Lineage	CPDB	Trio	Orchestra
1	8	{a,c}	{{a},{a,c}}	2a + ac	$2a^2$ + ac
1	9	{a,b,c}	{{c},{a,c},{b,c}}	2c + ac + bc	$2c^2$ + ac + bc
3	9	{b,c}	{{b},{b,c}}	2b + bc	$2b^2$ + bc

Figure 4.1: Example Data and Provenance In Current Provenance Models

provenance to obtain the entire provenance information for components of interest when using MMP.

We contrast our approach with that used in current models using the example in Figure 4.1. Consider the Orchestra [21] provenance representation, which Green has shown to be the most informative of the models shown in the figure. The result relation from the figure, with Orchestra provenance, which we call R_{out}, is repeated in Table 4.1.

TupleID	A	C	Orchestra Provenance
d	1	8	$2a^2 + ac$
e	1	9	$2c^2 + ac + bc$
f	3	9	$2b^2 + bc$

Table 4.1: Result Relation R_{out} from Figure 4.1 with Orchestra Provenance Annotations

We add another derivation step to our example by creating relation T from R_{out} with the query $T = \sigma_{C=9}(R_{out})$. Relation T is shown in Table 4.2.

TupleID	A	C	**Orchestra Provenance**
g	1	9	e
h	3	9	f

Table 4.2: Result Relation T with Orchestra Provenance Annotations

Suppose we wish to answer the question, "Which tuples in relation T have an ancestor in relation R?" The intuitive and correct answer is both tuples g and h. Tuple g in relation T has tuple e in relation S as an ancestor, which in turn has as ancestor tuples a, b, and c from relation R. Tuple h in relation T has tuple f in S as an ancestor, and f in turn has b and c as ancestors.

In order to answer this question using Orchestra's provenance representation, the user must first retrieve the provenance representation for each tuple in T. Next, the user must parse the provenance representations to extract the identities of tuples that are named as ancestors in the representations. In our example, tuple g has e as its immediate ancestor, and tuple h has tuple f as its immediate ancestor. Next, the user must retrieve the provenance of e and the provenance of f, and parse these provenance representations to determine the next generation of ancestor tuples to examine. For candidate result tuple g, tracing through e, the next generation includes a, b, and c. For candidate result tuple h, the next generation includes b and c. Because ancestors of both tuples g and h are found in R, both g and h are selected for output. This example demonstrates that obtaining this answer requires the user to parse provenance representations, and to re-construct and trace multi-generation provenance that is distributed across the database.

In contrast, consider the MMP query $U = \sigma_P(T)$, where the selectionPredicate P = "TUPLE HAS A PATH WITH A TUPLE IN RELATION R". This query directly returns a relation containing tuples g and h. As an alternative, the user might browse relation T and call up the provenance graph for each tuple in T, which would intu-

itively show their derivation from tuples in R. The provenance graph (for browsing) and predicate language (for querying) in MMP allow users to interact with provenance without the need to know anything about how provenance is modeled behind the scenes. We believe this feature contributes significantly to filling Gaps 2 and 3 described above. We note that browsing a large provenance graph would require a user interface to assist the user with visualization.

As shown in the example above, MMP provides a selection predicate language that allows users to select data by describing characteristics of paths in its provenance. For certain classes of queries, this language alleviates the need for users to manually review provenance returned from a query in order to select data of interest. This feature contributes to filling Gap 4 described above. Although the provenance predicate language of MMP is not comprehensive, MMP provides query language functionality for answering some questions difficult to answer using languages of other models in the literature.

The specification of MMP requires that provenance be introduced only as a side-effect of user-applied operations in the MMP language. Any implementation of MMP must follow the formal specification and create provenance in this way. In addition, no user operation manipulates or deletes provenance directly. These constraints on the accessibility of provenance by language operations contribute to filling Gap 5 mentioned above.

One additional contribution of MMP addresses an implicit gap in the literature: current provenance models are limited to a single data model (which is in all but one case, the relational data model). In this work, we define MMP for the relational model. However, data and provenance are treated separately by operators in the MMP language, and provenance and data are represented in distinct structures. We call this property *orthogonality* of provenance and data. Because of orthogonality, MMP offers the possibility that the provenance model in MMP may be preserved

if the data model implemented in MMP faces is changed. We conjecture that many data models can be defined in terms of components, just as we do with the relational model. To the extent that provenance for a data model is definable in terms of components having other components as ancestors, and to the extent that the progression of database state may be modeled as a succession of snapshots induced by operations on data, we believe that MMP can be adapted to support such data models.

4.2 Evaluating MMP Against Needs in Target Settings

Here we briefly assess how well our model meets the needs of users discussed in Chapter 1. We briefly recapitulate these needs as follows:

1. Models for data and provenance should represent external sources from which data was excerpted.

2. The history of operations performed on data should be recorded automatically and unobtrusively.

3. Because data may be encountered more than once, tools should allow for and remember multiple insertion of the same data.

4. Provenance should record what operations were performed on data, and who performed them.

5. Users should be able to visualize data and its provenance, as well as select data based on its provenance.

6. Provenance should be recorded at all manipulation granularities

7. Provenance should record all creation, manipulation, and query operations on data.

8. Models should retain deleted data and its provenance, and do so in a way that makes it available for provenance queries, yet prevents it from taking part in operations on data.

To address (1) above, MMP models external sources using external source referents. MMP's external source referents enable the model to trace data back to the external sources from which it was excerpted, allowing provenance queries to ask about specific external sources.

As defined in Chapter 3, provenanced MMP operations automatically induce provenance links as result faces are created. This automatic process does not affect the user's work model, addressing (2) above.

Data may be initially created in an MMP instance by DML or query operations. After creation, the MMP language allows data to be redundantly inserted using DML operators. Each DML operation induces provenance links, so an MMP instance addresses item (3) above by recording each operation applied to data, including those that redundantly insert or paste data.

Item (4) above is addressed because each face of MMP is annotated with the operation and user that induced it, as well as a timestamp of its creation. When provenance graphs are produced from an MMP instance, these annotations are used to label graph edges, so that users can clearly see each operation affecting the data represented by graph vertices. In addition, the predicate language of MMP allows queries to inspect these face annotations in order to perform data selection and projection.

MMP provides three mechanisms for users to interact with the provenance portion of an MMP instance, in support of item (5) above. First, the provenance graph mechanism defines a graphical view of data (as vertices) and revisions (as edges), so that users may visually browse data provenance. Second, the provenance predicate language in MMP allows users declaratively to define provenance characteristics of

interest when phrasing a query. The predicate language is supported for both the selection and projection operators of our extended relational algebra, providing the ability to select rows or columns based on their provenance or those of the attribute values they contain.

Relational DDL and DML operators address components at multiple granularities. However, traditional provenance models ignore DDL operations and support only tuple-granularity DML operations (with one notable exception, Buneman's CPDB [6]). In addition, traditional provenance models support queries, but only maintain provenance of result tuples, ignoring query-induced provenance for attribute values, schema attributes, and relations.[1] MMP defines provenance at all granularities (relation, tuple, attribute, and attribute value) for DDL, DML, and query operators that induce provenance. In addition, MMP introduces new operators that affect components at multiple granularities. Paste Value, Paste Tuple, and Paste Relation affect provenance for attribute values, tuples and attribute values, and all four granularities, respectively. Confirm Value, Doubt Value, and Drop Value affect provenance at attribute-value granularity. Drop Tuple affects provenance at tuple granularity as well as attribute value granularity. Because provenance is defined for all granularities affected by each operator in the MMP language, we can build a provenance graph for a relation, an attribute, a tuple, or an attribute value that shows its complete derivation history. This feature helps MMP to address item (6) above.

Each operator in the MMP language, except those that by our definition induce no provenance (for example, Create Relation), records provenance information for data affected by the operation. This feature addresses item (7) above.

MMP retains all data once it is inserted, including data later deleted by applied operations. By marking a data component as deleted, yet retaining it and all prove-

---

[1]Note that provenance for attribute values in query results coud be derived in these models from provenance for tuples, so long as attribute names for parent and child relations match. Provenance for relations may be also be derived, by inspection of queries, if a history of queries is retained. Provenance of attributes may also be derived from provenance of relations.

Provenance Model	Represents external sources	Automatic Provenance Collection	Multiple insertion of same data	Records operations and users	Gap1: Query, data definition, and data manipulation operators	Gap 2: manual parsing of provenance representations	Gap 3: manual assembly of multi-generation provenance	Gap 4: Simple language for selecting data by its multi-generation provenance	Gap 5: managing provenance orthogonally to data	Retaining deleted data and its provenance	Visualization of data and its multi-generation provenance
MMP	Yes	Yes	Yes	Yes	All 3	Yes	Yes	Yes	Yes	Yes	Yes
Lineage		Yes									
CPDB	Yes	Yes		Yes							
CPDB Extended	Yes	Yes		Yes	Query + DML						
Trio	Yes	Yes			Query + DML					Yes	
Orchestra		Yes			Query + DML						

Figure 4.2: Evaluating MMP and Current Provenance Models. Blank cells indicate that a model does not support a need.

nance links originating or terminating at it, MMP enables later provenance queries that specify deleted ancestor data as a characteristic of data satisfying a query. In addition, the predicate language can specify the data's deletion status as part of selection criteria. Operators in the MMP language are defined to ignore input data that has been deleted, so that query results and DML operations are consistent with the relational model. This feature addresses item (8) above.

Figure 4.2 summarizes the evaluation above, and presents a similar assessment of other models in the literature. The five gaps in the literature addressed by MMP are also shown. Empty cells in the table indicate that a model does not address a gap or requirement.

4.3 Relative Expressiveness of Algebraic Provenance Representations

Green [18] defined a lattice of expressiveness for provenance models from the literature for relational data at tuple granularity. This lattice includes Cui and Widom's Lineage model [12], which annotates tuples with the set of identifiers for tuples that contribute to their presence; Buneman's Why-provenance [8], in which these annotations are further refined into sets of input tuples that contribute independently to result tuple presence; Green's own provenance polynomials [21], which use natural number coefficients and exponents over contributing tuple IDs to express the cardinality and plurality with which input tuples combine to give rise to result tuples; and the provenance model from Trio [3], which is similar in many ways to Green's polynomial model. Examples of comparable representations for these models were shown in Figure 4.1.

Each of these models has provenance expressions representable as elements of a semi-ring. Green establishes a lattice comparing the expressiveness of these models by relating the various semirings by surjective semiring homomorphisms. One principal result of Green's work is that the most expressive model in this lattice is his own polynomial semiring, $\mathbb{N}[X]$. In order to compare expressiveness of the MMP provenance semi-ring $Prov^{SN}$ to the models considered by Green, we adopt his definition that, for naturally ordered semirings K_1 and K_2, if there exists a surjective homomorphism $m : K_1 \rightarrow K_2$, then K_1 is at least as expressive as K_2. By "at least as expressive", Green means that K_1 carries at least as much information about the provenance of data as K_2.

We first define a mapping from $Prov^{SN}$ to $\mathbb{N}[X]$, the most expressive in Green's lattice. This mapping, $m : Prov^{SN} \rightarrow \mathbb{N}[X]$ maps

1. the τ coefficients in $Prov^{SN}$ to unity in $\mathbb{N}[X]$

2. all other variables in $Prov^{SN}$ to identical variables in $\mathbb{N}[X]$

3. all integer coefficients in $Prov^{SN}$ to identical coefficients in $\mathbb{N}[X]$

4. 0 in $Prov^{SN}$ to 0 in $\mathbb{N}[X]$

5. 1 in $Prov^{SN}$ to 1 in $\mathbb{N}[X]$

6. the + and $\bullet$ operators in $Prov^{SN}$ to identical operators in $\mathbb{N}[X]$

Next, we prove that m is surjective and a homomorphism. To show that it is a homomorphism, we must prove that 1) $m(0) = 0$ and $m(1) = 1$, that is, that the additive and multiplicative identities remain after m is applied; 2) and for all distinct a and b in $Prov^{SN}$, $m(a + b) = m(a) + m(b)$ and $m(a \bullet b) = m(a) \bullet m(b)$.

The two equalities of condition (1) are satisfied by definition. In both $Prov^{SN}$ and $\mathbb{N}[X]$, 0 is the additive identity and 1 is the multiplicative identity. We now address condition (2). As defined in Section 3.7.3.1, τ factors appear as coefficients of monomials of one or more variables, or as coefficients of polynomial sums of such monomials. We restrict ourselves to proving condition (2) for expressions in these forms. For a monomial of a single variable with a τ coefficient, $\tau \bullet a_1$, $m(\tau \bullet a_1) = 1 \bullet a_1 = a_1$. Here we use juxtaposition of variables and coefficients to represent the $\bullet$ operation. We argue by induction over the number of variables $a_1, a_2, \ldots$ in a monomial A that $m(\tau \Pi_{n=1}^{N} a_n) = 1\Pi_{n=1}^{M} a_n = \Pi_{n=1}^{M} a_n$. Next, for any polynomial composed of terms $A_1, A_2, \ldots$ that are such monomials, we argue by induction over the number of terms in the polynomial that $m(\tau \Sigma_{k=1}^{K} A_k) = 1\Sigma_{k=1}^{K} A_k = \Sigma_{k=1}^{K} A_k$. Then for any two such polynomials, A and B, with coefficients τ_A and τ_B, respectively, we have both $m(\tau_A A + \tau_B B) = 1A + 1B = A + B$, and $m(\tau_A A) + m(\tau_B B) = 1A + 1B = A + B$. As a result, we have $m(A + B) = m(A) + m(B)$. Because we have that $m(\tau_A A) = 1A = A$ and $m(\tau_B B) = 1B = B$, we also have that $m(\tau_A A)m(\tau_B B) = AB$. In addition, we have that $m(\tau_A A \tau_B B) = 1A1B = AB$.

As a result, we have $m(AB) = m(A)m(B)$. Thus m is a homomorphism from $Prov^{SN}$ to $\mathbb{N}[X]$.

Finally, we must show that m is surjective. That is, we must show that all elements in the semi-ring $\mathbb{N}[X]$ are also represented by at least one element in $Prov^{SN}$. The set of elements in $\mathbb{N}[X]$ is the set of identifiers for tuples in the database. The set of elements in $Prov^{SN}$, when applied to the same database, is the same set of tuple identifiers, plus the set of τ coefficients. Furthermore, using the proposed homomorphism, when an expression in $\mathbb{N}[X]$ contains a tuple identifier, then the equivalent $Prov^{SN}$ expression contains the same tuple identifier. Thus we construct a $Prov^{SN}$ expression equivalent to a given $\mathbb{N}[X]$ expression by using the same identifiers for each mentioned tuple in the $\mathbb{N}[X]$ expression, and including the appropriate τ coefficients. Thus the proposed homomorphism is surjective.

The existence of this surjective homomorphism shows that $Prov^{SN}$ is at least as expressive as $\mathbb{N}[X]$. However, the converse is not true. There can be no surjective mapping from $\mathbb{N}[X]$ to $Prov^{SN}$: all elements in $\mathbb{N}[X]$ identify tuples, so there are none that represent members of $\{\tau_{il}\}$ in $Prov^{SN}$. Informally, we say that $Prov^{SN}$ is more expressive than $\mathbb{N}[X]$ because $Prov^{SN}$ represents the operations applied to data, but $\mathbb{N}[X]$ does not. Because Green has shown that $\mathbb{N}[X]$ is the most expressive of the provenance models included in his analysis, we can conclude that $Prov^{SN}$ is more expressive than the models compared by Green.

4.4 Relative Expressiveness of Provenance-related Queries

In this section, we compare the expressive power of the MMP query language to that of other provenance query languages in the literature. To do so, we first define a taxonomy of subclasses for an important class of provenance-related queries over relational data. Within each identified subclass, we develop sample queries. We then state each query in the language of MMP, and in the language of one or more

other provenance models from the literature, when possible. The number of queries phraseable in each language provides an indication of the expressive power of the language within the context of the defined class of queries.

4.4.1 Provenance Selection Queries

Current provenance models record provenance as additional attributes in the same schema as the (relational) data. Most then use relational operators to access both provenance and data. Typical queries posed in these systems aim to extract provenance based on characteristics of data. For example, such a query might be phrased in natural language as, "Where did tuples in this relation that contain data about 'Joe' come from?" In contrast, in informal discussions with domain experts we find that they often want just the opposite: they want to extract data based on characteristics of its provenance. Sahoo *et al.* [30] call queries like this *provenance selection queries*. We refine this taxonomy along three axes: whether a query aims to select tuples (rows) or attributes (columns) of data; whether the selection criteria mentions the location of ancestor data or historical derivation actions, or both; and whether the selection is based on a single such criterion, multiple criteria without relative timestamp-based ordering constraints, or multiple criteria with ordering constraints. Table 4.3 illustrates this taxonomy, numbering each subclass in it for convenience in our discussion. The two subclasses labeled "N/A" are not realizable. In each case, the provenance criteria (Both) conflicts with the criteria (Single): when there is but a single criteria to express in a query, it can be one of "Ancestor Location" or "Derivation", but not both.

4.4.2 Query set for Expressiveness Comparison

For each subclass shown in the table, we pose a sample query in natural language. We restrict these examples to tuple granularity, in order to fairly compare MMP (in

Component	Provenance Criteria			Criteria
	Ancestor Location	Derivation	Both	
Tuples	1	4	N/A	Single
	2	5	7	Unordered
	3	6	8	Ordered
Attributes	9	12	N/A	Single
	10	13	15	Unordered
	11	14	16	Ordered

Table 4.3: Enumeration of Subcategories of Provenance Selection Queries

which we can query about the provenance of relations, tuples, attributes, and the provenance of individual data values they contain) to models from the literature that track provenance only at tuple granularity.

The sample queries corresponding to the subclasses shown in Table 4.3 that we use in evaluating expressiveness are as follows:

1. Which tuples in relation R were derived from source X?

2. Which tuples in relation R were derived from source X and source Y?

3. Which tuples in relation R were derived from tuples in source X via tuples in relation $R2$?

4. Which tuples in relation R are derived from tuples that were inserted at least once between 04/15/09 and 04/15/10?

5. Which tuples in relation R were pasted from elsewhere and were also inserted directly?

6. Which tuples in relation R were derived from tuples inserted between 04/15/09 and 04/15/10, and later deleted?

7. Which tuples in relation R were derived from tuples in relation $R1$ that were inserted at least once?

8. Which tuples in relation R were derived from source X that were inserted at least once since they appeared there?

9. Which attributes in relation R were derived from source X?

The remaining subdivisions, 9-16, express queries for attributes (columns), and are identical in syntax to subdivisions 1 to 8, after substituting "attribute" for "tuple".

4.4.3   Comparison of Expressiveness

We compare MMP against three representative models and their attendant query languages from the literature:

- Buneman's Why-provenance model [5], with recursive Datalog

- The Trio model [3], with the TriQL query language

- Green's provenance polynomial model [20], also with TriQL

4.4.3.1   Buneman's Why-provenance model

Buneman's model relies on an extra relation, $Prov$, to store provenance information. The schema of $Prov$ includes a transaction ID for transactions that affect data, an operation attribute that indicates the operation applied in the transaction (i.e., C for copy-and-paste, I for insert, D for delete), a source attribute that identifies the root of a source subtree or external source of data for the transaction, and a destination attribute that identifies the root of a target subtree for the transaction. Buneman's model addresses only insertion, copy-and-paste, and deletion of data; no query operators are supported. Unlike other models in the literature that support only tuple granularity provenance, Buneman's model supports provenance for arbitrary subtrees in tree-structured data.

We express queries in Buneman's model in recursive Datalog, following Buneman's own work. In these queries, we make use of several pre-defined views described in Buneman's article: $Unch(t, p)$ intuitively means that subtree p was unchanged by transaction t; $Copy(t, p, q)$ intuitively means that subtree p was copied from subtree q by transaction t; and $From(t, p, q)$ intuitively means that the provenance of p due to transaction t is q. In Buneman's model, $Prov(t, C, p, q)$ is a fact from the $Prov$ relation, described above. Buneman defines $Unch$, $Copy$, and $From$ as follows:

$$Unch(t, p) \leftarrow \neg(\exists x, q.Prov(t, x, p, q)).$$

$$Copy(t, p, q) \leftarrow Prov(t, C, p, q).$$

$$From(t, p, q) \leftarrow Copy(t, p, q).$$

$$From(t, p, p) \leftarrow Unch(t, p).$$

Buneman also defines a transitive version of $From$ called $Trace$, which says that data at p at the end of a transaction t derives from data at q at the end of a transaction u:

$$Trace(p, t, p, t).$$

$$Trace(p, t, q, u) \leftarrow Trace(p, t, r, s), Trace(r, s, q, u).$$

$$Trace(p, t, q, t-1) \leftarrow From(t, p, q).$$

For purposes of comparison with Trio and Green's model, we limit Buneman's model to tuple-level provenance, though it is capable of representing tree-structured data (a generalization of relational data). We also assume the existence of an additional pre-defined view, $Member(R, p)$, which indicates whether tuple p is a member of relation R.

4.4.3.2 Trio

Trio has a built-in function in the TriQL query language to filter data based on provenance: Lineage(R1,R2) takes relations R1 and R2 as input, and outputs a relation consisting of tuples in R1 that have lineage traceable to R2. Trio is different from Buneman's model in that it tracks provenance only at the tuple level, and uses an SQL-like language for manipulations and queries. In addition, Trio retains deleted data, while Buneman's model does not.

4.4.3.3 Green's model

Green's model annotates provenance directly with a polynomial expression as described above as an additional attribute for each tuple, and uses query languages without purpose-built provenance functions. Because Green's model lacks an operator similar to Trio's Lineage(), we add to Green's model a similar operator to assist in phrasing queries. Such an operator is easy generated by a recursive query over provenance annotations.

4.4.3.4 Example query 1

Natural language: Which tuples in relation R were derived from source X?

<u>MMP</u>: $\sigma_{\text{tuple has a path with (a source with name = X)}}(R)$

<u>Buneman</u>: $Result(p) \leftarrow Member(R, p), Trace(p, t_{now}, q, \_), Member(X, q).$

<u>Trio, Green</u>: Select * from R Where Lineage(R,X)

4.4.3.5 Example query 2

Natural language: Which tuples in relation R are derived from both source X and source Y?

<u>MMP</u>: $\sigma_{predicate}(R)$

where *predicate* = "tuple has a path with (a source with name = X) and a path with (a source with name = Y)"

<u>Buneman</u>: $Result(p) \leftarrow Member(R,p), Trace(p, t_{now}, q_1, \_), Member(X, q_1),$
$Trace(p, t_{now}, q_2, \_), Member(Y, q_2).$

<u>Trio, Green</u>: SELECT * FROM R WHERE Lineage(R,X) AND Lineage(R,Y)

4.4.3.6 Example query 3

<u>Natural language</u>: Which tuples in relation R are derived from source X via tuples in relation $R2$?

<u>MMP</u>: $\sigma_{predicate}(R)$

where *predicate* = "tuple has a path with (a source with name = X before a tuple in relation = R2)"

<u>Buneman</u>: $Result(p) \leftarrow Member(R,p), Trace(p, t_{now}, q_2, t_2), Member(R2, q_2),$
$Trace(q_2, t_2, q_1, \_), Member(X, q_1), t_2 < t_{now}.$

<u>Trio, Green</u>: SELECT * FROM R WHERE EXISTS ((SELECT * FROM R2 WHERE Lineage(R,R2) INTERSECTION SELECT * FROM R2 WHERE Lineage(R2,X))

4.4.3.7 Example query 4

<u>Natural language</u>: Which tuples in relation R are derived from tuples that were inserted at least once between timestamps 04/15/09 and 04/15/10?

<u>MMP</u>: $\sigma_{predicate}(R)$

where *predicate* = "tuple has a path with (an operation with action = INSERT and where time $\geq$ 4/15/09 and where time $\leq$ 4/15/10)"

<u>Buneman</u>: $Result(p) \leftarrow Member(R,p), Ins(t,q), t \geq 04/15/09,$

$t \leq 04/15/10, Trace(p, t_{now}, q, \_)$.

Trio, Green: not expressible in closed form.[2]

4.4.3.8 Example query 5

Natural language: Which tuples in relation R were pasted from elsewhere and were also inserted directly?

MMP: $\sigma_{predicate}(R)$

where *predicate* = "tuple has a path with (an operation with action = INSERT and not an operation with action = Paste) and a path with (an operation with action = Paste)"

Buneman: inexpressible, because no such history is representable in the underlying data model)

Trio, Green: not expressible in closed form, as described in query 4

4.4.3.9 Example query 6

Natural language: Which tuples in relation R were derived from tuples inserted between 04/15/09 and 04/15/10, and later deleted?

MMP: $\sigma_{predicate}(R)$

where *predicate* = "tuple has a path with (an operation with (action = INSERT and where time $\geq$ 4/15/09 and where time $\leq$ 4/15/10) before a value that is expired)"

Buneman: $Result(p) \leftarrow Member(R, p), Ins(t, q), t \geq 04/15/09,$

$t \leq 04/15/10, Del(t_2, r), Trace(p, t_{now}, r, t_{any}), t_{any} \geq t_2,$

$Trace(q, t_{any}, q, t_{any2}), t_{any2} \geq t$

---

[2]Trio provides a function, *Lineage(r1, r2)*, that selects data in relation r1 with ancestors in r2. However, Lineage takes specific relations as arguments. Query 4 explicitly mentions the descendant relation, R, for use as argument r1 to Lineage(), but does not specify the relation for argument r2. As a result, the user must trace ancestry "step-by-step", which requires multiple queries, each of which can only be written after its predecessor has returned results.

Trio, Green: not expressible in closed form, as described in query 4

4.4.3.10 Example query 7

Natural language: Which tuples in relation R were derived from tuples in relation $R1$ that were inserted at least once?

MMP: $\sigma_{predicate}(R)$

where *predicate* = "tuple has a path with (a tuple in relation = R1) and (an operation with (action = INSERT))"

Buneman: $Result(p) \leftarrow Member(R, p),$

$Trace(p, t_{now}, q, \_), Member(R1, q), Ins(\_, q)$

Trio, Green: not expressible in closed form, as described in query 4

4.4.3.11 Example query 8

Natural language: Which tuples in relation R were derived from tuples in relation R_1 and were also inserted at least once since they appeared there?

MMP: $\sigma_{predicate}(R)$

where *predicate* = "tuple has a path with (a tuple in relation = R1) before (an operation with (action = INSERT))"

Buneman: $Result(p) \leftarrow Member(R, p), Trace(p, t_{now}, q, t),$

$Member(R1, q), Ins(t_2, q), t < t_2$

Trio, Green: inexpressible, as described in query 4

4.4.3.12   Example query 9

Natural language: Which attributes in relation R were derived from source X?

MMP: $\pi_{\text{attribute has a path with (a source with name = X)}}(R)$

Buneman, Trio, Green: inexpressible: schema is not addressed by these models

All remaining provenance selection subdivisions address projecting attributes that meet certain provenance criteria. MMP is capable of expressing queries in all of these, while comparable models cannot express such queries.

4.4.3.13   Conclusions About Expressiveness of Provenance Selection Queries

We summarize the expressiveness of provenance selection queries from models compared here in Table 4.4. In the table, models capable of expressing the example queries in a subdivision are noted by an identifying letter: M for MMP, B for Buneman's model, T for Trio, G for Green's polynomials.

We conclude from the results of this comparison that MMP can express at least some queries in subclasses where other models cannot, and that no subclasses of the class of provenance selection queries are unaddressable by MMP.

	Provenance Criteria			
Component	Ancestor Location	Derivation	Both	**Structure**
Tuples	MBTG	MB	N/A	Single
	MBTG	MB	MB	Unordered
	MBTG	MB	MB	Ordered
Attributes	M	M	N/A	Single
	M	M	M	Unordered
	M	M	M	Ordered

Table 4.4: Expressive Power of Comparable Provenance Models

4.5 Other Advantages of MMP Relative to Other Models

In Sections 4.3 , we showed that MMP has more expressive provenance polynomials than other models in the literature. In Section 4.4, we showed that the query language of MMP can express queries that other models cannot. In addition, MMP has other advantages not yet discussed. We discuss two of these here.

4.5.1 Accessing Ancestors and Operational History of Data

Recall that MMP retains all ancestral databases, and links components in those databases together based on their provenance relationships. As a result, MMP allows for browsing and querying of multi-generation provenance. In MMP, users can access the identity and values of data used to derive other data, the operations used to derive data, and the timeline corresponding to the derivation. We contrast this capability with the Orchestra model [21], shown by Green to have the most expressive provenance representation of relevant models in the literature [18]. Orchestra (and other comparable models that rely on semi-ring provenance representations) can easily be extended to construct representations of multi-generation provenance. Recursively substituting variables in Orchestra provenance polynomials with the polynomials representing their provenance results in polynomials that represent original sources of data, and how data were combined disjunctively and conjunctively to arrive at a result. This same recursive approach can also be used to generate a provenance graph. At each recursive step, a graph node can be generated to represent each component visited, and the provenance relationships implied by the provenance polynomial operators can be used to generate edges between nodes. However, the provenance graphs generated from and Orchestra database are not as descriptive as those derived from an MMP instance.

First, Orchestra does not retain information about the derivation operations used, when they occurred, or who applied them. Instead, Orchestra retains (and can thus

include in a provenance graph) only how data combined conjunctively or disjunctively to give rise to a result. In contrast, MMP includes all operations, the users who applied them, and timestamps in provenance graphs.

Second, Orchestra models only queries, not DML or confidence operations. Query result data that are further manipulated by DML operations, or that accrete confidence operations, cannot be represented in Orchestra's provenance model. In contrast, MMP allows for provenance of data subject to manipulation (and application of confidence expression operations) after a query, and also allows for provenance of data subject only to manipulation (without queries).

Third, Orchestra retains only a single database, rather than all ancestral data. Modifications made to ancestor data after its use in deriving descendant data cannot be distinguished from modifications made to ancestor data prior to its use in deriving descendant data. When constructing a provenance graph, for example, the data values found as ancestors thus may or may not be accurate. In fact, if ancestor data was subsequently deleted after its use in deriving descendant data, ancestral data values may not be found at all. Because of this limitation, Orchestra cannot reliably present ancestor data values as part of data provenance. In contrast, MMP retains all ancestral data, including deleted data, and uses these ancestral databases to determine data values used to derive descendant data.

4.5.2 Computing Forward-Looking Provenance

Because MMP retains all generations of data and provenance for all data, users (and programs) are able to trace provenance of data backward to ancestors (as done in building provenance graphs), as well as forward from ancestors to descendants (by traversing provenance links from their terminals to their origin). We call the latter tracing the *progenance* of data. Progenance is useful in detecting when descendant data is made obsolete by changes in ancestor data. For example, in a cancer therapy

prioritization workflow, as shown in Figure 4.3, data inputs to each step in the workflow are subject to frequent revision as scientific knowledge improves. In the figure, the overall workflow is composed of smaller sub-workflows that feed results into successive sub-workflows. Each sub-workflow may also have input data not derived from its preceding sub-workflow. For example, Treatment Selection depends on the results of the Therapy Prioritization sub-workflow, relying on treatment rankings and confidence measures. Treatment Selection also uses inputs from literature mining to find effectiveness studies, new information about confounding conditions in the patient, and so on. If information input to an earlier sub-workflow changes, it may invalidate information in use later in the workflow. For instance, the content of a genetic reference library used as input to the Molecular Characterization sub-workflow might change as more gene mutations are identified. If this information was used to derive gene expression signatures used as input to the Therapy Prioritization sub-workflow, and that in turn affected candidate treatment rankings, then decisions made in Treatment Selection might be affected. Progenance analysis provides a means to identify whether derived data may be affected by a change in ancestor data, alerting users to the need to redo some analysis steps.

Other models in the literature are also capable of tracing progenance. To trace progenance for one generation of descendants of a selected component in a database, the entire database must be inspected to find components with provenance representations that reference the selected component. To trace the next generation forward, each component found to be a descendant in the first step must be the subject of a similar search. Each successive generation in this recursive trace requires a search of the entire database to find the next generation of descendants. This process is functional, but is computationally expensive. If a database consists of n components, and we wish to trace the descendants of each, n^2 components must be inspected. If each of these components is found to have on average m descendants in the first gener-

ation, then tracing the second generation of progenance requires an additional mn inspections for each of the n, for a two-generation total of $n^2 + mn^2$. In general, if we assign each inspection of a component a fixed cost of 1, then the complexity of tracing k generations of descendants is

$$\sum_{x=1}^{k} m^{k-1}n^2.$$

In contrast, tracing progenance of a selected component in an MMP instance requires only that each provenance link terminating at the selected component be traversed to its origin, and that each link terminating at each of those origins be traversed, until all data components connected to the selected component are identified. This traversal never requires a complete search of the database: only descendants are visited. The resulting complexity of tracing k generations of descendants for each of n components is

$$\sum_{x=1}^{k} m^{k}n.$$

This saving in computational complexity is possible in MMP because MMP retains all historical data.

4.6 Relative Complexity of Provenance-related Queries

We do not intend to make or formally prove broad claims about the relative complexity of queries expressible in the models compared here. Instead, we seek to give a rough indication of relative complexity for a set of examples that represent some query subclasses described above. For subclasses 1-8 in our taxonomy, as shown in Table 4.4, a variety of the languages we consider can express the representative queries we propose. We use these queries, and three other similarly representative queries from each of the eight subclasses to compare the complexity from the user point of view of writing queries in these languages.
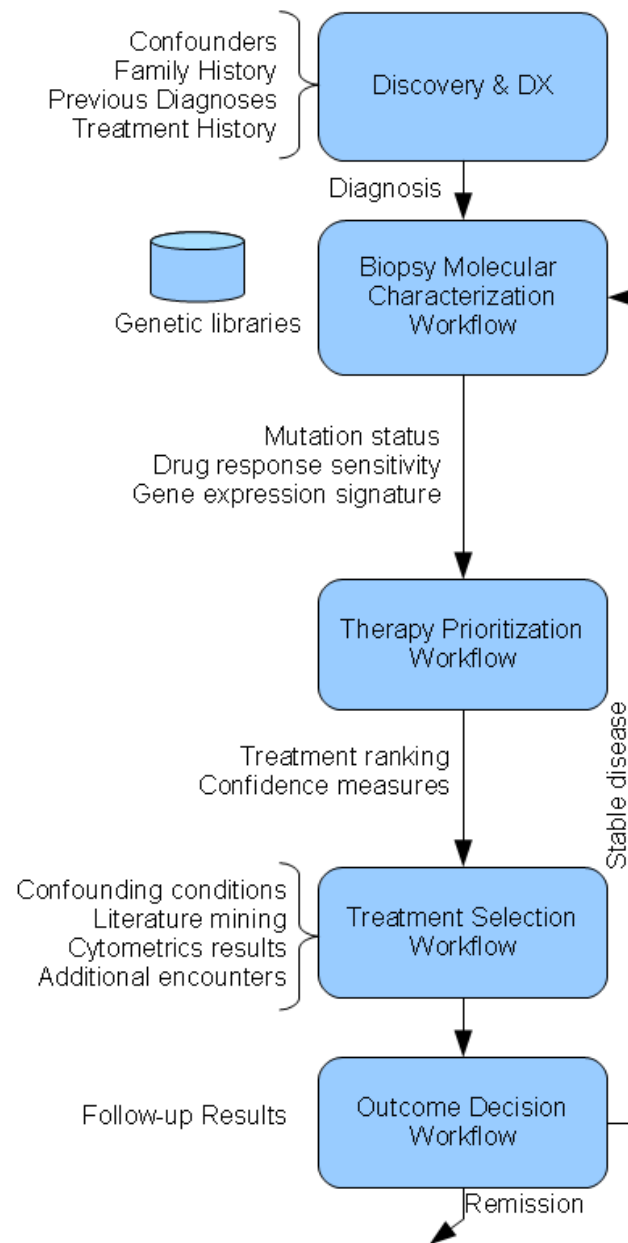
Figure 4.3: Cancer Therapy Prioritization Workflow

Because query expressions are closely related to statements in a programming language, we adopt a complexity metric typically used in programming environments. Perhaps the simplest metric would be the number of lines of (non-comment) code. However, for comparing expressions of very few lines, a simple line count is not very informative. McCabe [27] suggests a metric, now commonly used in software engineering projects, that measures code complexity as 1 + the number of IF, AND, and OR tokens in code. More recently, this metric has been termed *cyclomatic* complexity. Unfortunately, McCabe's scheme does not account for the complexity of logic in conditional statements, just the number of these tokens. In addition, in some of the languages we compare, for example Datalog, AND and OR conditions are implicit, so McCabe's metric might miss significant complexity. Halstead [22] proposes a complexity measure, *Size*, that counts the number of distinct operands and operators in code. Despite some controversies [9], Halstead's metrics have been shown to correlate well to intuitive measures of code complexity reported by users. However, Halstead's metric does not account for non-operator tokens, for example parentheses, which in query languages may account for a significant differences in expression semantics (for example, order of evaluation of operators). A similar metric, Levitin's *token count* [26], takes these tokens into account. Levitin's metric counts individual tokens in an expression, though it counts matched parentheses as a single token. As an example of Levitin's accounting, consider the Datalog expression for example query 1 above: $Result(p) \leftarrow Member(R, p), Trace(p, t_{now}, q, \_), Member(X, q)$. Levitin's metric for this expression is 25. In this work, we use Levitin's token count as a comparison metric for query complexity.

We summarize the results of complexity comparisons in Figure 4.4. It is not surprising that as criteria structure increases in complexity, query complexity also rises for all models. It is also not surprising that queries describing derivation actions are typically more complex than those that simply name ancestors. Comparing the
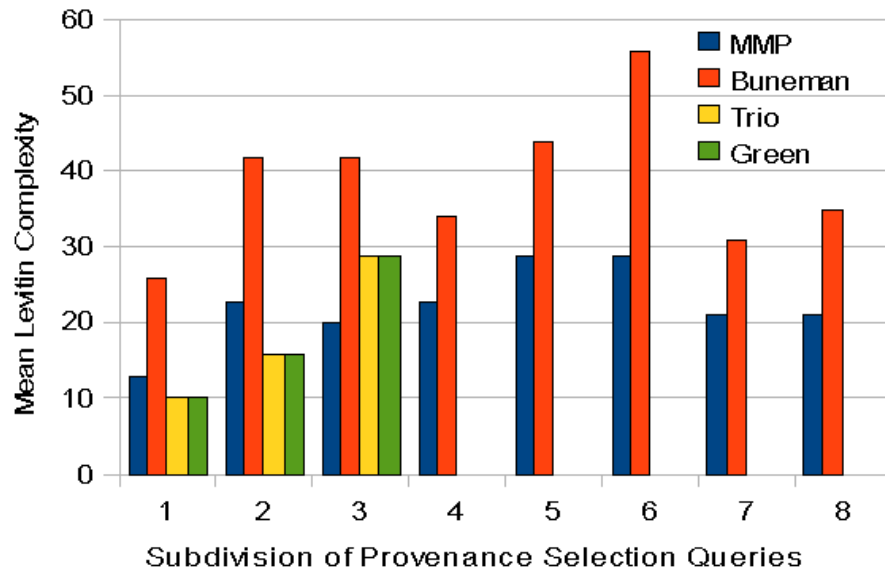
Figure 4.4: Query Complexity Comparison

models, we see that the Datalog queries for Buneman's model are uniformly more complex than the relational algebra queries for MMP. We also see that the queries for Trio's model and Green's model are comparable in complexity to the MMP queries, though applicable in fewer query subclasses. Note that the complexity of MMP is worse than Trio and Green's model for subclasses 1 and 2. In both cases, the built-in function *Lineage()* in the TriQL language is sufficient to trace the necessary provenance, making the query syntax quite compact. For queries where *Lineage()* is not sufficient, as in subclass 3 where ordering is important, MMP is more compact even though its syntax is specifically designed to be intuitive for users. We conclude that the MMP predicate language has roughly comparable or better conciseness than other provenance models in the literature, for similar subclasses of provenance selection queries.

4.7 Chapter Summary

With regard to subjective evaluation of MMP against gaps in the literature, we have shown that five explicit gaps, and an additional implicit gap, are addressed by MMP. Evaluating MMP against the needs of users in example settings, we have shown that MMP addresses a variety of user needs beyond what other models in the literature address. Although these evaluations are not easily quantifiable, we believe that MMP contributes new capabilities not found elsewhere.

With regard to objective evaluation of expressiveness, we have shown that the algebraic notation defined in MMP is more expressive than that of other models in the literature: MMP is as expressive as the most expressive model (as shown by Green) if representation of applied operations is not considered; if applied operations are considered in the comparison, MMP is more expressive than all other models addressed. With regard to the ability of MMP to answer provenance selection queries, we have shown some evidence, though not conclusive proof, that MMP addresses more subclasses of this query class than other comparable models, and does not fail to cover any subclasses addressed by other models. With regard to relative complexity of comparable queries in other models, we have shown that MMP has roughly comparable or lower complexity to other models.

However, we recognize that a direct implementation of MMP may have less practical usability than other provenance models in the literature. For example, MMP has significant redundancy in data, because each face is a copy of the entire database. We address this problem in Chapter 6 by proposing a logical model that faithfully supports MMP while removing redundant data.

Chapter 5

Characterizing Performance of Implementation Choices for MMP

MMP includes aspects of both relational and graph data models. Achieving a correct and performant implementation of MMP should be possible if the abstractions in it can be mapped onto a software infrastructure that is fast and scalable and can represent both of these data models. However, we know of no single software platform that supports both data models. To provide some high-level guidance concerning possible MMP implementations, in this chapter we study the performance of MMP implemented on two candidate platforms: a relational DBMS and a graph DBMS. We contribute definition of a performance benchmark that models data and provenance queries from data curation settings, as well as the results of our performance-comparison studies.

5.1 Benchmarks and Metrics

As defined in Chapter 2, MMP provenance predicates select data by its provenance. An implementation of MMP would evaluate a provenance predicate by comparing the properties of potentially all generations of data ancestors and derivation actions to characteristics specified in the predicate. Current literature does not offer benchmarks for multi-generation provenance queries such as this, where it may be necessary to trace provenance of data through multiple generations of materialized results. The only work we know of that addresses benchmarks for provenance is by Kar-

vounarakis, Ives, and Tannen [25]. They describe a micro-benchmark set for prove-
nance in a data-exchange setting. In that setting, provenance represents only a single
derivation step, and so involves only a single generation of ancestor data. Multiple-
generation provenance queries differ substantially from single-generation queries.
Because all provenance models in the literature store provenance of only one gen-
eration at a time, extraction of multiple generations of provenance information re-
quires an iterative (recursive) approach of visiting successive ancestors. Recursive
queries are typically more complex and require substantially more computation and
I/O resources than single-generation queries. They may also be more difficult for the
average user to write.

MMP provides a language that supports interrogation of multi-generation prove-
nance and relational data, either separately or together. To compare performance of
MMP implementations, our benchmark suite includes separate benchmarks for re-
lational data queries and multi-generation provenance queries so that performance
on each can be characterized independently. Our benchmark does not include DML
operations, because we expect that insertion, copy-and-paste, and deletion of indi-
vidual data values or individual tuples will have acceptable response time for users.
We leave benchmarks that address insertion of large granularity data, for example
entire relations, for future study.

The portion of our benchmark that addresses performance on relational data
queries includes only single-operator SELECT and PROJECT queries. One reason
for this is that we first want to understand relative performance of implementations
under simple workloads before progressing to more complex queries. A second rea-
son is that comparison and analysis of multi-operator query performance falls, in
large part, into the domain of query optimization, which is outside the scope of this
work.

In the remainder of this section, we define our benchmarks for queries over re-

Figure 5.1: Benchmarks and Implementations Tested

lational data and provenance. For each, we define the structure of the data used in each implementation we tested for MMP; the workload for measurement; and the metrics used for measurement and comparison. As shown in Figure 5.1, we run the relational data benchmark and the provenance benchmark over two different MMP implementations: one built on a relational database infrastructure; and one built on a graph-database infrastructure.

5.1.1 Data query benchmark

The data set for our data benchmarks is a set of tables of varying size, composed from a single original table with 29 attributes. The data is excerpted from a database of US government grants from fiscal year 2009, and consists of roughly 257,000 tuples. Of the 29 attribute domains, 16 are strings, 10 are integers, one is a floating-point number, and the remainder are dates. The string domains vary from six characters up to 160 characters. Average tuple length is 340 bytes.

Using this initial table as a source, we produce tables of 512, 1024, 2048, 4096, 16384, 65535, and 1M (2^{20}) rows. Each table smaller than the original begins with the first tuple in the original table and is truncated at the desired cardinality. The largest table (1M tuples) is formed by concatenation of copies of the initial table. In

each table, an additional attribute, with domain of integer, is added to the schema, and filled with a unique integer for each tuple. This attribute is called *tupleKey* in the remainder of this discussion. We use this attribute to ensure that duplicate tuples do not appear in our test relations.

5.1.1.1 Data structure for relational database testing

To test relational implementations, the tables described above are created as relations in a relational database. The unique integer in each table is declared as the primary key for the corresponding relation, and the relation is sorted on that key. This allows benchmarks to test operations that may take advantage of sorted files. In order to test operations that take advantage of index structures in relational databases, a selected string attribute (the same string attribute in each relation) is used to construct an unclustered B+ tree index in the database.

5.1.1.2 Data structure for graph database testing

To test graph implementations, the tables described above are created as graph structures in a graph database that are analogous to relations. We implement two different representations. In the first, which we call *value-as-node*, a single graph node represents the relation. Additional nodes represent: each attribute in the relation's schema; each tuple in the relation; and each attribute value. Edges connect the relation node to each member tuple node. Additional edges connect the relation node to each member attribute node. Each attribute value node is connected by a single edge to the tuple node to which it belongs, and by a single edge to the attribute node to which it belongs. The value-as-node representation was chosen because it provides a direct representation of MMP components, and thus may permit an effective exploitation of the graph query language provided by the graph DBMS. In order to test operations that take advantage of index structures in graph databases, the same string attribute in
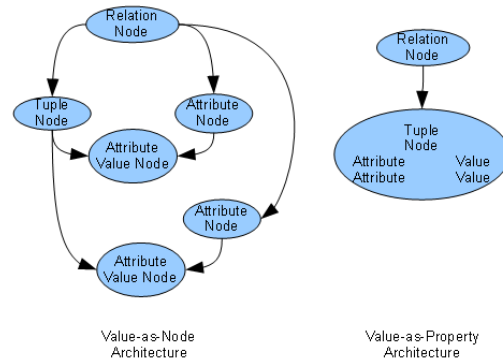
Figure 5.2: Data structures for Graph Database Testing

each graph database is used to construct an index. We test two graph DBMSs: Neo4j (neo4j.org) and HypergraphDB (code.google.com/p/hypergraphdb/). Hypergraphdb uses a proprietary index structure, and Neo4j uses the Lucene [28] indexing engine.

In the second data representation, which we call *value-as-property*, we retain the relation node and tuple nodes. Attribute names and attribute values of tuples are modeled as named properties of tuple nodes rather than as separate nodes. Figure 5.2 shows the structure of these two data representations for a relation with a single tuple with two atomic attributes (shown as simply "attribute" in the figure with "value" as the attribute value).

Other data representations analogous to relational structure are possible in graph databases. We chose this second representation because it is economical in terms of space and, likely, the number of disk I/Os needed.

5.1.1.3 Data query workload

The data query workload consists of the following queries:

- A query that selects all attributes from the table row identified by a specific primary-key value

- A query that selects all attributes from each table row where the indexed string attribute is equal to some specific value

- A query that selects all attributes from each table row where a selected non-indexed attribute is equal to some specific value

- A query that selects one string attribute from all table tuples

For relational DBMS testing, the tests above map to SQL queries as follows:

- SELECT * FROM relationName WHERE tupleKey = constant;

- SELECT * FROM relationName WHERE indexedAttribute = constant;

- SELECT * FROM relationName WHERE non-indexedAttribute = constant;

- SELECT attribute FROM relationName;

No graph databases we know of support a relational query language. Instead, access to data is achieved through the use of an API that provides simple graph traversal and information-retrieval functions. The tests above map to graph database API operations for the value-as-node structure as follows:

- Traverse the graph and retrieve all nodes that represent tuples. As each is retrieved, retrieve and interrogate the node representing its tupleKey attribute. If this attribute value matches the specified constant, retrieve all nodes that represent the tuple's attribute values. This approach simulates scanning the (unsorted) data to find the tuple of interest

- Use the index created during database creation to retrieve the identifiers of nodes that are indexed using the attribute value of interest, and then use those identifiers to retrieve all related attribute value nodes

- As in the first item above, traverse the graph and retrieve all nodes that represent tuples. As each is retrieved, retrieve and interrogate the node representing its tupleKey attribute. If this attribute value matches the specified constant, retrieve all nodes that represent the tuple's attribute values. This approach simulates scanning the (unsorted) data to find the tuple of interest

- Traverse the graph and retrieve all nodes that represent tuples, then retrieve the attribute value for the attribute of interest for each

Note that these queries are different for the value-as-property representations. In this case, there are no attribute value nodes to retrieve. Instead, we retrieve tuple nodes and then retrieve their property values.

5.1.2 Provenance query benchmark

Our data set for provenance benchmarks augments the data from our data benchmark with provenance information. Each data table has some number of *starting* tuples where tracing of provenance begins. Each remaining tuple in the dataset is part of the provenance of one of the starting tuples. We use tables with 512, 2048, 8192, and 32768 starting tuples. We study performance of queries over provenance at the granularity of tuples. The performance of MMP implementations on provenance queries at the attribute value level is left for future work.

Each data set has two variants. One has *linear* provenance. The other has *bushy* provenance. In linear provenance, a starting tuple has a provenance link to one ancestor tuple, which is in turn linked to one its ancestor tuple, and so on, for a total of 31 tuples in each provenance chain. In bushy provenance, a starting tuple has provenance links to two ancestor tuples, each of which is in turn linked to its two ancestor tuples, and so on, so that a total of 31 tuples appear in each provenance bush. Provenances of starting tuples do not intersect. That is, no tuple appears in
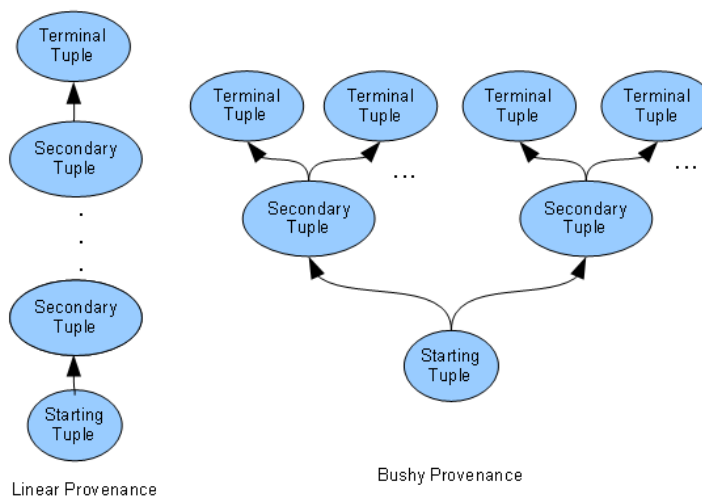
Figure 5.3: Examples of Linear and Bushy Provenance Structures

the provenance of multiple starting tuples, nor are starting tuples in the provenance of any other tuples. Certain secondary tuples, called *terminal* tuples, have no further provenance. These tuples represent original sources from which all other tuples are ultimately derived. Figure 5.3 shows examples of linear and bushy provenance for a single starting tuple.

5.1.2.1 Provenance structure for relational database testing

For linear provenance data sets, each relation has an additional attribute that is a foreign key to the primary key (tupleKey) of the relation. This foreign key is used to record the immediate ancestor of a tuple. For bushy provenance data sets, each relation has an adjunct provenance relation with a schema that includes two attributes, *child* and *parent*. Child is a foreign key referencing the primary key of the data relation, and identifying a descendant tuple. Parent is a foreign key referencing the primary key of the data relation, identifying a tuple that is part of the provenance of child. The two attributes form the primary key of the provenance relation. Note that starting tuples appear only in the child column of provenance, and terminal tu-
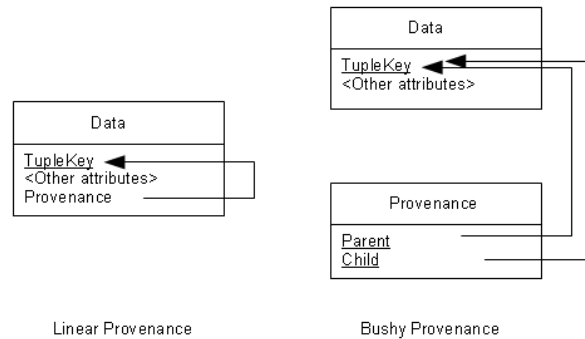
Figure 5.4: Schema Diagrams for Linear and Bushy Provenance

ples appear only in the parent column. Two indexes are created on the provenance relation: a clustered index on child, and an unclustered index on parent. Figure 5.4 shows schema diagrams for our linear and bushy provenance test structures, with data attributes omitted. Foreign key relationships are denoted by arrows.

5.1.2.2 Provenance structure for graph database testing

For both linear and bushy provenance, we represent provenance links as directed edges in a graph database. Each edge originates at a node representing a result tuple of an operation, and terminates at a node representing an input tuple to the same operation. Figure 5.3 illustrates the structure we use for modeling provenance in graph databases, where each tuple is represented by a node in the graph.

5.1.2.3 Provenance query workload

The MMP query language provides predicates for selecting data based on properties of individual paths in the data's provenance. Figure 5.5 shows the bushy provenance from Figure 5.3, along with the individual paths that must be computed prior to predicate evaluation. Each path is indicated in the figure as a dark solid line.

Our provenance query workload consists of a single query that computes all
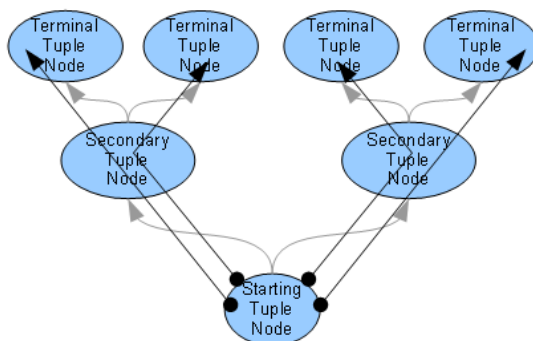
Figure 5.5: Enumerating Provenance Paths

provenance paths for each starting tuple in a relation, and then simulates comparing the original source tuple of each path to a constant. In the graph database, provenance paths are materialized by the database API as instances of a Java iterator class containing node identifiers of path members. In the relational database, provenance paths are materialized as tuples that have a common *path identifier*, comprised of two extra attributes: the identity of the starting tuple of the path, and the identity of the original data source tuple in the path. Note that this mechanism for naming provenance paths assumes that provenance is tree-structured, rather than having a more general, directed acyclic graph structure. This simplification does not affect performance results. An example of the query used in relational database testing is shown in Figure 5.6. Because single-generation provenance is recorded for each tuple, and because these must be combined into provenance path descriptions in order to compare paths against the path-based predicates defined in MMP, we use a recursive query. We begin with a data relation, called *d30x1m* in the query shown. This relation holds all data tuples, and is so named because there are 30 attributes in the relation schema and 1M tuples in the relation instance. We also use a relation that lists all one-generation predecessors of all tuples in *d30x1m*, called *d30x1mpreds* in the query shown; and we use a relation that holds a list of all tuples that represent

original sources (that is, those that have no further provenance), called *pathendpoints* to be used in the query shown.

In Part 1 of the query, we use recursion to populate a temporary relation, *ancestors*, with all 2-tuples such that the second element is the identity of a starting tuple in d30x1m, and the first element is the identity of one of its ancestors in d30x1m. Part 2 materializes *ancestors* as a temporary table once the recursion in Part 1 reaches a fixed point. In Part 3 of the query, we use recursion to populate another temporary relation, *descendants*, with all 2-tuples such that the second element is the identity of an original source in d30x1m, and the first element is the identity of one of its descendants in d30x1m. In Part 4, we join *ancestors* and *descendants* with the data relation, d30x1m, to form a new relation where each tuple contains a copy of some tuple from d30x1m, along with the identity of the starting tuple and original source tuple that identify the provenance path to which the copy belongs.

5.1.3 Performance Comparison Metrics

We measure two aspects of query performance: elapsed time, and number of page I/Os. Elapsed time directly measures one of the primary concerns of users, but is difficult to measure accurately when intervals are short. In addition, elapsed-time measurements are subject to distortion by other system activity. In experiments where elapsed time is very short, and in experiments where the ratio of elapsed times between models is very large, we omit reporting of elapsed time measurements. Disk activity may be of less concern to users. However, these measurements are much less likely to be distorted by system activity. In addition, disk activity is commonly used in database systems as a way of estimating and comparing query cost, and as a tool for estimating how query costs may scale to larger data set sizes.

```
-- Part 1: recursive subquery to find tuples
--         connected to each starting tuple
WITH RECURSIVE ancestors(thisTuple, startTuple) AS
-- result schema is (node, associated starting node)
(  ( -- base case: add all start tuples to result
     SELECT tupleKey, tupleKey
     FROM d30x1m
     where tupleKey < 32000 )  UNION ALL (
     -- recursive case: add parents of all in result
     -- until a fixed point is reached
     SELECT p.parent, a.startTuple
     FROM ancestors a, d30x1mpreds p
     where a.thisTuple = p.child ))
-- Part 2: materialize temporary table for later use
select * into temp_ancestors from ancestors;
-- Part 3: recursively find tuples connected to each source
WITH RECURSIVE descendants(thisTuple, rootTuple) AS
-- result schema is (tuple, associated source)
(  ( -- start by adding all sources
     SELECT tupleKey, tupleKey
     FROM pathendpoints )  UNION ALL (
     -- now add descendants of those in temp table
     -- until we reach a fixed point
     SELECT p.child, d.rootTuple
     FROM descendants d, d30x1mpreds p
     where d.thisTuple = p.parent
     and p.parent != p.child ))
-- Part 4: JOIN tables of ancestors and descendants
--  to give a complete pathname (start, end) to each tuple
select d30x1m.tupleKey as pathname_1,
       pathendpoints.tupleKey as pathname_2,
       temp_ancestors.thisTuple,
       d30x1m.* into finaldata
from d30x1m, pathendpoints, temp_ancestors, descendants
where d30x1m.tupleKey < 32000
and temp_ancestors.thisTuple = descendants.thisTuple
and temp_ancestors.startTuple = d30x1m.tupleKey
and descendants.rootTuple = pathendpoints.tupleKey;
```

Figure 5.6: Recursive SQL Query for Computing Provenance Paths on 32,000 starting tuples

5.2 Experimental Setup

All tests were run on an HP xw6200 workstation, with a single Xeon processor operating at 3.4GHz, with hyper-threading disabled. The system includes 2GB of main memory. Databases are stored on a RAID mirrored array of two 500GB Western Digital Caviar WD740GD-50FLCO disks, supported by a Silmage RAID controller, and connected to the system board by an SATA-300 link.

Data query workloads for relational database testing were run on MySQL server 5.1, version 14.14, distribution 51.48. Data query workloads for graph database testing were run on two graph databases: HypergraphDB version 1.0; and neo4j version 1.1.

Provenance query workloads for relational database testing were run on Postgres Plus Standard Server 8.4. We moved to the Postgres server from the MySQL server because at this time, MySQL does not support recursive SQL syntax needed to traverse tuple provenance. Provenance query workloads for graph databases were run on the neo4j database described above. We chose not to run provenance tests on HypergraphDB because its performance on the relational data tests was comparable to or worse than neo4j.

5.3 Experiments and Results

As an initial experiment to calibrate a reasonable scale for our experiments, we ran data query 1 over tables with sizes 512, 1024, 2048, and 4096 rows, using the value-as-node data structure on HypergraphDB. MySQL was used as the relational database for comparison. While each test was completed in under 1 second in MySQL, HypergraphDB required tens of minutes for the smallest test, and did not complete the next larger test within 2 hours. We then reduced the data tables to only 3 attributes and tested again. This time, response times were substantially faster for

Tuples in Relation	MySQL I/Os	Mean HGDB I/Os
512	1	13976
1024	1	34174
2048	1	211443
4096	1	830000

Table 5.1: Results for 3-attribute Relations Using Value-as-Node Structure

HypergraphDB, though still very long compared to MySQL. We then measured page I/Os for both MySQL and HypgergraphDB at 512, 1024, 2048, and 4096 tuples with three attributes. Results are shown in Table 5.1. Note the exponential growth of I/O cost for the graph database compared to the constant cost for the relational database.

A subsequent experiment showed that Neo4 performance was comparable to that of HypergraphDB with the value-as-node structure. As a result, we adopted the value-as-property structure for all subsequent experiments, and modified the test queries accordingly. The poor performance of the value-as-node structure appears to be caused by long retrieval times for individual graph nodes that model attribute values. Experiments show that there is apparently only a linear search method, akin to a file scan, to retrieve nodes in HypergraphDB. Nodes are not sorted on disk by their ID property, so no accelerated retrieval such as binary search is available.

5.3.1 Relational Data Query Tests

Next, we ran all four data queries over relations using the full 30-attribute schema, with relation cardinalities of 512, 1024, 2048, 4096, 16384, 65536, and 2^{20} tuples. Results of each are described below. Before each test run, the database server was stopped and restarted, to ensure that no query could take advantage of "warm-start" conditions.
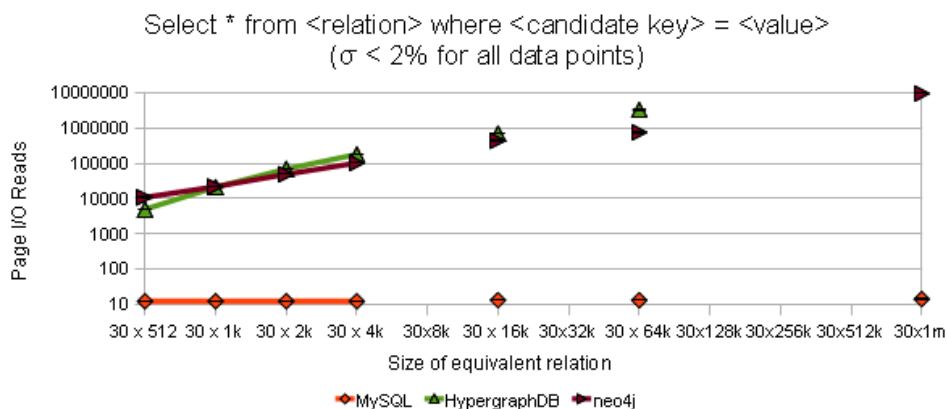
Figure 5.7: Test 1 Results. Size of equivalent relation is calibrated in number of attributes per tuple times number of tuples.

5.3.1.1   Test for Data Query 1

Test results for data query 1 are shown in Figure 5.7. Page I/Os are reported here as the metric. As expected, the relational database cost is small and grows slowly, consistent with a binary search over a relation sorted on its primary key. However, the graph databases have no facility to sort nodes. For these, a full scan of the graph is required. Graph database performance is similar for the two databases tested, up to the test case with 65536 tuples. Beyond this point, HypergraphDB performance lags significantly: run-time for Test 1 on HGDB for a relation with 2^{20} tuples exceeds sixteen hours. Neo4 completes the largest test case in roughly two hours.

As shown in Figure 5.7, the relational database outperformed the graph databases by about 3 orders of magnitude for small data sets, and relative performance continued to be a linear function of data set size out to the largest test data set, where the relational engine outperformed the graph engine by 6 orders of magnitude.

As shown in the figure, the graph databases incurred a significant cost even for the smallest relation size. After that, their performance scaled linearly with the size

of the relation tested[1]. Throughout the range from 512 up to at least 16,384 tuples, page cost per attribute value retrieved was consistently in the range of 0.8, while above that range, cost dropped to about half that number.

The observed cost per attribute value retrieved is consistent with our understanding of Neo4 architecture and the JVM it runs on. In Neo4, an in-memory table is used to relate the identity of a graph node to the on-disk address where the corresponding node object is stored. Node properties' names and values are accessed by table lookup as well. Thus a single dereference of a node identity to retrieve a node object, followed by de-referencing one pointer in the node object to retrieve a node property, should require at most 2 I/Os per attribute value retrieved.

We also know that attributes of nodes are stored in list form on disk, and that Neo4 makes extensive use of in-memory caching. Even though none of the retrieved values is subject to prior reference (that is, there is no temporal locality to exploit), we expect that spatial locality should enable the Neo4 in-memory cache to successfully serve some references to properties. We know that the average length of a tuple is 340 bytes. Assuming a typical list construct where each list entry consists of one attribute value and a 32-bit pointer to the next list item, all attributes of a tuple could fit into a single 4096-byte virtual page as used by Windows XP. Thus the minimum I/O cost to access a tuple would include one operation to fetch the tuple object, and one to fetch attribute values (assuming perfect spatial locality of the attribute value list), for a cost of two I/Os per tuple, or 0.067 I/Os per attribute value.

This analysis shows that the expected range of between 0.067 I/Os and 2.0 I/Os per attribute value bounds the observed range of 0.4 to 0.8. Unfortunately, in Java it is not possible to measure the degree to which objects are allocated contiguously. Java handles are dereferenced in the JVM object table to obtain the virtual address of the

---

[1]Our graph uses a log_{10} scale on the y axis, and a log_2 scale on the x axis. As shown in the figure, $log_{10}Y \div log_2X = k$, a constant. Recall that by the base conversion rule of logarithms, $log_{10}Y = log_2Y \div log_210$, and log_210 is also a constant. Then $log_{10}Y \div log_2X = log_2Y \div log2_X = k \times log_210$ and so performance scales linearly
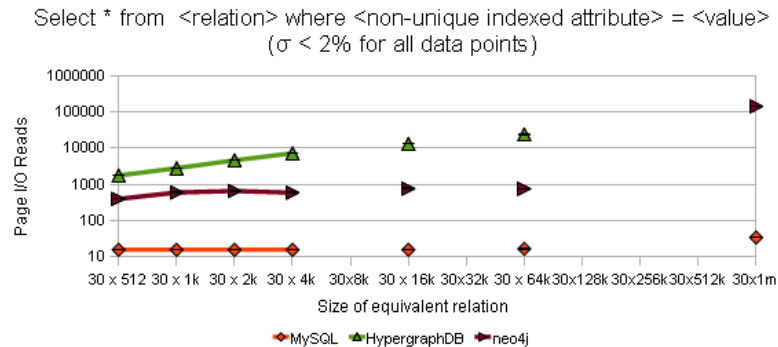
Figure 5.8: Test 2 Results

object, because objects may be moved in the Java heap as the garbage collector runs in the background. This level of indirection prevents us from establishing a more practical lower bound on cost, which would likely be higher than the 0.067 value described above due to non-contiguous allocation of property storage in the heap.

5.3.1.2   Test for Data Query 2

In response to the relatively poor performance of the graph databases in Test 1, we modified the original plan of Test 2 to retrieve only a single attribute for each tuple in the graph database. This change simulates a predicate test against only a single attribute. It should also provide nearly best-case relative performance in favor of the graph database. Test 2 results are shown in Figure 5.8. Recall that in this test, we use an index in both relational and graph databases to accelerate retrieval of tuples.

Relational database performance in this test is similar to that seen in Test 1: the cost of the binary search is not very different from the cost of descending the B+ index search structure. For HypergraphDB, the cost is fairly linear across relation size. For Neo4, cost is very nearly constant up to 65536 tuples per relation, and then jumps substantially for the largest relation size. As expected, the relational database outperforms both graph databases by roughly 2 orders of magnitude at minimum,

and by as much as 3 orders of magnitude.

Relational database performance on Test 2 is not surprising. We expect a small I/O cost that scales as log_2 of dataset size for traversing the index, plus a small cost for retrieving matching tuples.

In Neo4, instead of scanning all tuples, the required tuples can be found using the Lucene index maintained over the data. The resulting I/O cost is fairly flat, growing at about the same rate as the cost for the relational database for most of the tests. In this regime, we see per-attribute costs start at about 6 pages and decrease slightly with increasing dataset size. Since cost should reflect retrieving matching tuple identities from the index, searching through the attribute names to find the one that matches the query, and then retrieving the single matching attribute for each matching tuple, this measurement seems reasonable. Recall that the cost of looking up each tuple is amortized over only a single attribute lookup in this test, rather than over all 29, as seen in Test 1. As a result, per-attribute costs are higher, though overall cost of the test is substantially lower.

At the largest dataset size we see substantially more I/O cost for Neo4 on Test 2: about sixteen I/Os per attribute retrieved. This cost appears to be due to Neo4's memory cache spilling, perhaps due to the large size of the index.

In HypergraphDB, we see improvement relative to the relational database from Test 1 by an order of magnitude or more, yet performance is still linear to dataset size. Given the use of HGDB's built-in index service, this result was unexpected. We expected that the use of the HGDB index structure would impose a logarithmically increasing cost, and that subsequent retrieval of nodes after index lookup would take constant time, resulting in a nearly constant access cost across dataset size. The conclusion we draw is that the cost of accessing the HGDB index service is linear in the number of entries in the index, rather than logarithmic as we expected. That is, the HGDB index service appears to be list-structured rather than tree-structured.
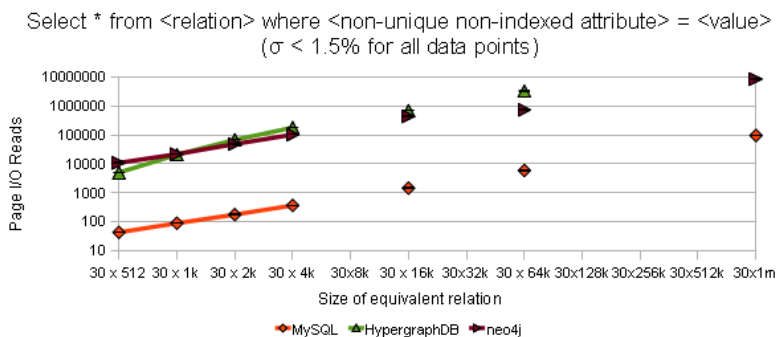
Figure 5.9: Test 3 Results

5.3.1.3 Test for Data Query 3

Results of Test 3 are shown in Figure 5.9 Test 3 is similar to Test 1, except that the data is neither indexed nor sorted on the attribute of interest. As expected, performance of the graph databases is identical to that in Test 1 because both require a full scan of all tuples and their attributes. Relational database performance on Test 3 differs as expected from Test 1, because the full input relation must be scanned. As a result, relational performance is linear to dataset size in Test 3. Query costs of the graph databases in Test 3 are roughly 2 orders of magnitude higher at all relation sizes than costs for the relational database.

Performance of the graph databases in Test 3 can be explained in the same way as for Test 1. Performance of the relational database for Test 3 is not surprising: because a full scan of the relation is required, performance should scale linearly with the size of the relation. With a total tuple size averaging 340 bytes, roughly 12 tuples fit on each page. Thus a full scan of the relation should require a page I/O cost roughly equivalent to the number of tuples in the relation divided by 12, which is consistent with test results.
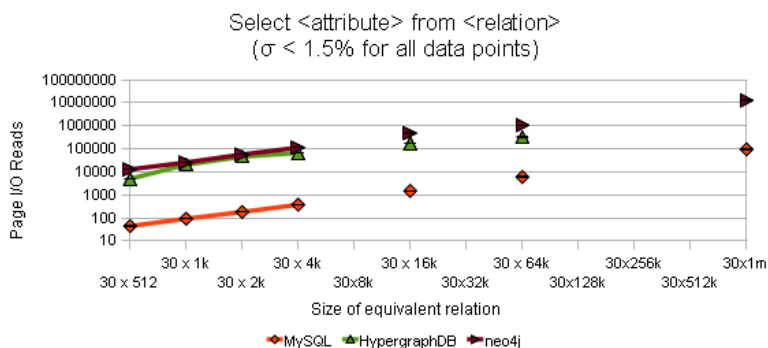
Figure 5.10: Test 4 Results

5.3.1.4 Test for Data Query 4

Results of Test 4 are shown in Figure 5.10. Test 4 and test 3 have similar data access patterns: both require a full scan of the input relation. Results of the two tests are similar, as expected, with the relational database outperforming the graph databases by roughly two orders of magnitude.

5.3.1.5 Test Results Using Warm-Start Caches

In order to observe the effect of data re-use on successive queries in the graph databases, we performed another test where we ran Data Tests 1, 2, and 3 repeatedly without re-starting the database server for Neo4. In each case, the repeated runs on relation sizes of 512, 1024, and 16384 tuples showed less than one percent additional page I/Os, indicating that Neo4's in-memory LRU cache performed well. For relations of size 2^{20} tuples, however, the indexed query performed well, but the two tests using scans of the entire input relation showed costs comparable to the non-cached tests, indicating that Neo4's cache was not large enough to hold more than a modest-sized relation. This result suggests that for larger data sets, the graph databases we tested will take only minimal advantage from in-memory caching.
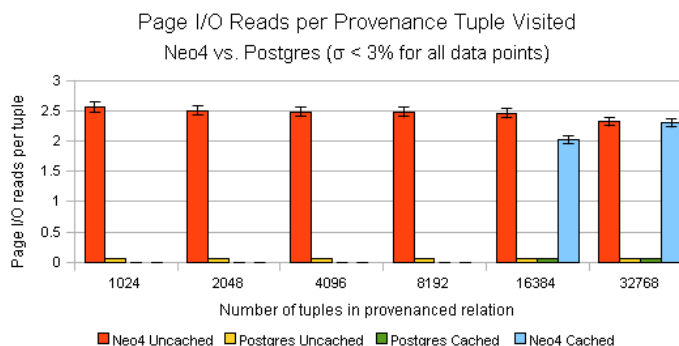
Figure 5.11: Linear Provenance Test Results

5.3.1.6 Conclusions on Data Tests

We have taken into account the benefits of in-memory caching by graph databases; chosen a data structure for the graph databases that heavily favors performance over use of a natural graph structure; and removed some of the advantage of query optimizers in relational databases by limiting tests to single operators. Still, test results lead us to expect that for a range of data set sizes, relational databases outperform graph databases by two to six orders of magnitude on single-operator selection and projection queries.

5.3.2 Provenance Predicate Tests

We initially ran our linear provenance benchmark on relations with cardinality $31m$ tuples, for $m = 1024, 2048, 4096, 8192,$ and 32768 starting tuples. These tests were run on a Postgres relational database and a Neo4 graph database. Tests were run under both cold-start and warm-start conditions, with average results from ten trials of each shown in Figure 5.11.

For Neo4, uncached cost is very consistently near 2.5 page I/Os per tuple visited. This result compares unfavorably to the Postgres uncached cost of 0.075 page I/Os per tuple: Neo4 I/O costs are a factor of 33 worse than Postgres. On cached tests up

to 8192 starting tuples, Neo4 and Postgres perform similarly, showing very little I/O. However, when the number of starting tuples rises to 16384 or more, the Neo4 cache spills, and performance of Neo4 drops to near uncached levels.

Performance of Postgres on this test can be explained by noting that on average, about thirteen tuples fit on each memory page. Each tuple is retrieved once in order to trace a provenance path. One page I/O per thirteen tuples corresponds closely to the measured 0.075 page I/Os per tuple.

Performance of Neo4 on this test can be explained by noting that each access to a tuple node requires one page I/O in the uncached case. Retrieval of the single relationship edge connecting to the next node in a provenance path also requires one page I/O. This total accounts for 2 of the 2.5 page I/Os seen on average during the test.

The performance advantage of the relational database should be an upper bound, because the linear provenance test unfairly favors the provenance representation used in the relational database. Because each tuple has only one ancestor in the linear case, each tuple need be accessed only once, and no auxiliary relations are needed to store provenance. In realistic provenance, each tuple may have an arbitrary number of immediate ancestors, a situation more fairly represented by the bushy provenance benchmark.

To establish a more realistic performance comparison, we ran the bushy provenance benchmark over relations with m = 512, 2048, 8192, and 32768 starting tuples, measuring both page I/O operations and elapsed time. Results for the tests are shown in Figure 5.12. Only uncached trials were measured. The figure shows that both in terms of I/O cost and elapsed time, the graph database starts off at a disadvantage. However, the relational query to compute provenance paths involves multiple recursive sub-queries and creation of multiple temporary tables. As the size of input relations increases, the cost of the relational query scales up rapidly. For relations
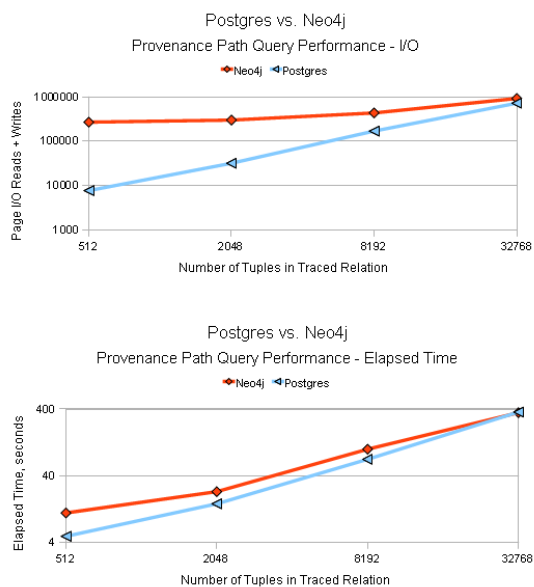
Figure 5.12: Bushy Provenance Test Results

with 32768 starting tuples, both I/O cost and elapsed time are comparable for the graph database and relational database solutions.

Performance of Neo4 on this test is comparable to performance on the linear provenance test, with I/Os per tuple within a factor of two for all comparable data set sizes. Postgres performance on this test is not directly comparable to Postgres performance on the linear provenance test: the bushy provenance test uses multiple recursive sub-queries, and both creates and re-uses several temporary relations that are materialized on disk. We verify Postgres performance on this test in two ways: first, we examined contents of each intermediate relation created by the query to ensure that provenance paths are constructed correctly. Second, we determined the upper bound on computational complexity for the query used in the test. The query is shown in Figure 5.6.

The recursive subquery of Part 1 traverses n binary trees, where n is the number of starting tuples. For each visited node in a tree, it scans the input relation to find the

node's parents. The initial (non-recursive) portion of Part 1 populates a result relation with all starting tuples. The cost of this portion is linear in n. Because the input file is sorted on the search key and selection begins at the start of the relation, only the portion corresponding to the n starting tuples is scanned. The recursive portion of Part 1 traverses n binary trees of ancestors, giving this part the recurrence relation $T(m) = 2T\left(\frac{m}{2}\right) + O(1)$, where m is the size of the tree. This part is thus linear in the number of nodes in the tree m, and also scales with n, so is O(*nm*). However, m is a constant in this test, so the recursive part is O(*n*). Part 2, which writes a temporary relation to disk, is linear in the size of the number of starting tuples as well, or O(*n*). Part 3 traverses paths from source to starting tuple, so it has recurrence relation $T(l) = T(l-1) + O(1)$ where l is the length of the path. This relation result gives O(l) cost, but l is a constant in this test. Part 3 also scales by the number of paths per starting tuple, which is also a constant, and by the number starting tuples. Thus Part 3 is O(n). Part 4 is done as a sequence of hash joins, so is O(r), where r is the size of the data relation. However, r is a constant multiple of the number of starting tuples, so Part 4 is also O(n). The overall I/O complexity of the query is the maximum of the complexities of the individual parts of the query, and so is O(n), which is consistent with observed linear scaling behavior.

5.3.2.1 Conclusions for Provenance Tests

The linear provenance portion of our benchmark heavily favors the relational implementation we tested. Using a bushy provenance model, it appears that, as we scale to larger data quantities, performance of the graph database we tested meets, but does not exceed, that of the relational database tested. Both show performance that scales linearly with number of starting tuples. If we assume that the rate of performance change stays roughly constant as dataset sizes increase further, graph database performance also appears to scale more favorably than that of the relational database.

5.3.3   Implications for MMP Implementations

Relational data query performance heavily favors the relational databases at the sizes we tested. This relative performance advantage seems unlikely to change substantially as data sizes increase. However, provenance query performance seems to favor a relational platform at relatively small data sizes, but favors a graph database platform as data sizes increase. Thus for small to moderate database sizes, a solely relational implementation seems reasonable, but our results point to no single, existing implementation platform as being an obvious choice for MMP with large datasets.

5.4   Other Ideas for Accelerating MMP Implementations

A natural choice for implementing MMP when larger datasets are expected might be a hybrid solution that combines the graph and relational platforms, using each where it offers the greatest performance advantage. There are several possibilities for structuring a hybrid implementation. In a naive approach, a relational database might store data and perform data operations, while a graph database might store provenance relationships and facilitate evaluation of provenance predicates in queries. A coordination layer of software would be needed to re-write queries in order to execute appropriate portions on the two underlying databases. More sophisticated hybrids are possible as well. Because our work focuses on development of our conceptual model, MMP, and showing that logical models that support MMP are feasible, exploring the space of hybrid solutions, and characterizing their performance, is beyond the scope of this work.

As already discussed, the MMP implementations we tested store provenance information one generation at a time (either as edges in a graph database or as tuples in a provenance relation). In order to process provenance queries, these individual provenance relationships must be traversed to form the provenance paths that

provenance predicates evaluate. Currently, this traversal is done on demand when a provenance-related query is run. This lazy materialization of paths is arguably efficient from a storage point of view because paths are only computed when needed. However, lazy materialization is also the major contributor to the cost of answering provenance queries. Lazy materialization of paths is also analogous to lazy materialization of single-generation provenance proposed by Widom [12], but later superceded by provenance models that eagerly compute single-generation provenance.

It makes sense to consider eager materialization of provenance paths as a performance enhancement for multi-generation provenance. Note that if a new tuple is materialized in the result of a query, its provenance paths are inherited from its immediate ancestor tuples, and extended by the single provenance link representing the query. If a new tuple is inserted or pasted, it inherits its provenance in a similar way. If an existing tuple is inserted again, it gains one new provenance path. Each of these computations is far cheaper than the on-demand computation of all provenance paths starting from single-generation provenance information.

One possible implementation is to eagerly compute provenance paths as entries in a path index. Path indexes, especially for XML data, have seen significant interest in the literature over the last 10 years [17, 10]. A path index stores path descriptions as index keys, along with a pointer to the node that the path is incident to. Path descriptions consist of the ordered names or properties of path edges, and may include the properties of intervening nodes. We extend the path index notion by letting the index key be a combination of a provenance path description and the identity of the relation to which the incident (starting) tuple belongs. Then evaluation of a provenance predicate would consist of translating the predicate into a pattern that can be looked up using the index, and combining that pattern with the identity of the target relation of the selection operator, and then looking up the resulting key in the index. Materialization of query result tuples is then a matter of retrieving tuples using the

index value.

One problem with this approach is that provenance predicates specify only the interesting characteristics of matching paths, rather than entire paths, while path index entries describe paths in detail. However, He and Singh [23] have addressed the problem of matching incomplete graph "patterns" to complete graphs. This work may be applicable to the problem of looking up path patterns in an index of full path descriptions. Another problem with this approach is that the size of the path index grows very quickly. For example, each query result tuple has provenance paths nearly identical to all its immediate ancestor tuples. We leave research into making effective provenance path indexes as future work.

Other solutions beyond those we tested may hold promise. For example, we considered modeling provenance relationships with RDF triples. However, it has been shown by Erling [15] and is generally accepted in the community that queries over RDF data using SPARQL or other RDF query languages are outperformed by storing triples in a relational database, writing queries over them in SQL, and translating the result into RDF. This result suggests that our approach of storing provenance directly in a relational database should be faster than an architecture using an RDF store for provenance. In addition, the main language for querying RDF, SPARQL, does not support path variables, or the ability to specify paths of arbitrary length. As a result, we chose not to evaluate storing provenance or data in languages such as RDF and XML.

5.5 Chapter Summary

In this chapter, we defined a benchmark suite for comparing MMP implementations with regard to their performance on relational data queries and provenance predicates. Our data benchmarks represent common, real-world data queries that perform simple data selection and projection. Our provenance benchmarks represent common

operations used in real-world provenance queries that trace ancestry of database components. We documented our experiments running the benchmarks on both relational database and graph database implementations, using production-quality databases as platforms.

As part of this work, we learned that while graph databases provide a workable platform for an MMP implementation, only part of the capabilities of graph databases are needed for MMP. While graph databases support arbitrary graph or hypergraph structures, the MMP model needs only linear path structures for evaluation of provenance predicates and synthesis of provenance graphs for user visualization. This simplification suggests, for example, that if a hybrid platform were used for MMP implementation, only part of graph database functionality might need to be supported in the hybrid.

Results of our studies showed that the relational databases we tested outperform the graph databases by 2 to 6 orders of magnitude on the data portion of our benchmark. We learned that for all but very small dataset sizes, the relational and graph database tested perform similarly on the provenance portion of our benchmark, with the graph database scaling better with increasing data set size. In sum, the results of our performance studies show that none of the implementation alternatives we tested satisfy the goal of a performant implementation for both relational data and provenance queries. We have suggested a number of approaches to improve performance of relational or graph database implementations of MMP, as well as some ideas on structuring a hybrid implementation that might take advantage of the strengths of both of these. These ideas we leave for future work.

Chapter 6

A Logical Model to Support MMP Implementation

Inherent in MMP is substantial redundancy: data is replicated at every timestep (i.e., on every face) in MMP at which it exists. Thus this chapter introduces MMP^L, a logical model that faithfully supports MMP while eliminating data redundancy. MMP^L is based on a temporal database, where data is defined to be visible during a finite time period (for example by use of start-time and stop-time attribute values for each data component), yet still remains in the database once that time period has concluded. This model of data is consistent with the need in MMP to represent that data has a creation time and (possibly) a deletion time, yet data remains after deletion, even though it does not participate in further operations.

MMP defines provenance links that connect relations, attributes, tuples, and attribute values in one face directly to the appropriate relations, attributes, tuples, and attribute values, respectively, in the prior face. Thus, MMP treats each component in each face as if it were addressable as an independent object. We note that in modern relational databases, many components already are addressable with "behind-the-scenes" identifiers. For example, tuples are typically addressable by their contents. Similarly, attributes are addressable by their names, which are unique across the database if we make the universal relation assumption. Relations can also be thought of as having a distinct address, because their names are required to be unique. Finally, it requires only a simple extension to imagine that a distinct address for an

attribute value can be formed as the concatenation of the ID of the tuple to which it belongs and the attribute in whose column it exists.

An MMP^L instance includes a temporal database d^L with the following extensions. Each component c (relation, attribute, and attribute value, in addition to tuple) in an MMP^L instance has a start time $c.Ts$ (which indicates the time at which it was created) and a stop time $c.Te$ (which indicates the time at which it was deleted, and ∞ otherwise). Each component also has a distinct identifier, or *ID*, that uniquely identifies it. Taken together, we call the set of all these identifiers *ALL-IDs*. We define a function $myID : C^L \rightarrow$ ALL-IDs, where C^L is the set of all components in d^L, that, given a component in an MMP^L instance, returns the unique ID of that component. When a component is selected either by a user browsing the database with a graphical user interface, for example, $myID$ can be invoked to uniquely identify the selected component.

The following constructs added to MMP^L are not typically found in any temporal database. An MMP^L instance also includes a set S^L of tokens representing external source referents. Each token is labeled with the name of the external source it represents. Like components, each token is assumed to have a unique identifier that can be obtained by applying $myID$ to the source's description as used in an applied operation.

An MMP^L instance also includes a set of edges E^L, each edge of which corresponds to a provenance link in the corresponding MMP instance. Each edge $e \in E^L$ is labeled with the operation that induced it, the user that applied the operation, and the timestamp at which the operation was applied. We define this label as $\lambda_E(e)$. The timestamp portion of the label uniquely identifies the face in the corresponding MMP instance from which the edge it is attached to originates.

MMP^L implements the MMP language. In MMP^L, each revision takes as input the database d^L, and current edge set E^L, and produces the new d^L and E^L.

6.1 Transforming Conceptual Models into Logical Models

An MMP^L instance M^L can be defined that corresponds to an MMP instance M as follows. Recall that M has face set D, where each $d_n \in D$ is a relational database; R_n is the set of relations in d_n; for each $r_j \in R_n$, $T_{n,j}$ are the tuples in r_j and $A_{n,j}$ are the attributes of r_j; and for each tuple $t_{n,j,k} \in T_{n,j}$, $V_{n,j,k}$ are its attribute values such that each $v_{n,j,k,l} \in V_{n,j,k}$ corresponds to one attribute $a_{n,j,l} \in A_{n,j}$. Also recall that for each d_n, the set of components of d_n is $C_n = R_n \cup T_{n,j} \cup A_{n,j} \cup V_{j,k,l}, 1 \leq j \leq |R_n|, 1 \leq k \leq |T_{n,j}|$, and $C = \bigcup_{m=1}^{n} C_m$. Note that any component in $d_i, i \leq n$ appears in d_n, though possibly marked as Expired. Thus copying all components in d_n gets all the components in all faces of M. Further recall that S is the set of external source referents in M. Let M^L be an MMP^L instance with database d^L, edge set E^L, and external source referent set S^L. We assume that M has had n revisions applied prior to creating M^L. That is, the current (most recent) face of D is d_n. We define $time(d_n)$ to be the timestamp portion of $\lambda_D(d_n)$. Similarly, we define $op(d_n)$ to be the operation identifier portion of $\lambda_D(d_n)$, and $user(d_n)$ to be the user portion of $\lambda_D(d_n)$. We define a mapping $\Lambda : MMP \to MMP^L$ that takes as input an MMP instance M and produces an MMP^L instance M^L. Comments in definitions are delimited using "/* */".

$$\Lambda(M), \text{ applied to } M \text{ at time } n =$$

/* Step 1a: create the data representing the faces in M */
$\forall r_j \in R_n, 1 \leq j \leq |R_n|$, create a relation $r_j^L \in R^L$ with the same name.
Set $sameComponentAs(r_j^L) = r_j$.
$\forall a_l \in A_{n,j}, 1 \leq l \leq |A_{n,j}|$, create an attribute $a_l^L \in A_j^L$ with the same name.
Set $sameComponentAs(a_l^L) = a_l$.
$\forall t_k \in T_{n,j}, 1 \leq k \leq |T_{n,j}|$, create a tuple $t_k^L \in T_j^L$.
Set $sameComponentAs(t_k^L) = t_k$.

$\forall v_{n,j,k,l} \in V_{n,j,k}$, where j, k, and l range as shown in the above statements, create an attribute value with identical value $v_{n,j,k} \in V_{j,k}^L$.

Set $sameComponentAs(v_{j,k,l}) = v_{n,j,k,l}$.

/* Step 1b: set the creation time for each component in the database */

$\forall c \in d^L, c.Ts = time(d_x)$ where d_x is the earliest face in M in which $sameComponentAs(c)$ appears.

/* set the deletion time for each component, if it has been deleted */

$\forall c \in d^L, c.Te = time(d_y)$

where d_y is the earliest face prior to n, if one exists, in which

$Expired(sameComponentsAs(c)) = True$, or $c.Te = \infty$ otherwise.

/* Step 2: for each external source referent in S, create a matching token in S^L */

$\forall s \in S$, add to S^L a token s^L labeled with the external source identifier, and where $s^L = sameSourceAs(s)$.

/* Step 3: for each provenance link in L that originates from a component c in d_x and terminates at a set of components B in d_{x-1}, create a matching edge in E^L */

$\forall l_p(c, B) \in L$, where $B \cap S = \emptyset, c \in C_x$, and $B \subseteq C_{x-1}, E^L$ contains a hyper-edge $e(o, T)$ such that $o = myID(sameComponentAs(c))$,

and $T = \{myID(sameComponentAs(b)) | b \in B\}$.

$$e.op = op(d_x),$$

$$e.user = user(d_x), \text{ and}$$

$$e.time = time(d_x).$$

/* Step 4: for each provenance link in L that originates from a component c in d_x and terminates at an external source referent, create a matching edge in E^L */

$\forall l_p(c, s) \in L$, where $s \in S, c \in C_x, E^L$ contains an edge $e(o, t)$ such that

$$o = myID(sameComponentAs(c)),$$

and

$$t = myID(sameSourceAs(s)).$$

$$e.op = op(d_x),$$

$$e.user = user(d_x), \text{ and}$$

$$e.time = time(d_x).$$

Note that in our definition there is a bijection $sameComponentAs : C^L \leftrightarrow C$ that relates any component in d^L to its corresponding component of the same type in M, and a bijection $sameSourceAs : S^L \leftrightarrow S$ that relates external source components with identical labels.

6.1.1 Equivalence Classes of Language Operators

We do not define the effect of all MMP operations on MMP^L here. Instead, we define the following classes of operators from the language with the same characteristics, and define the effect of one sample operator (italicized) from each class on MMP^L in detail.

1. *Drop Attribute*, Create Source, Create Attribute, Create Relation, Drop Relation

2. *Insert Tuple*, Insert Value, Drop Value, Drop Tuple, Confirm Value, Doubt Value

3. *Paste Tuple*, Paste Value, Paste Relation

4. *Queries*

Class 1 consists of operations that affect individual components, and induce no provenance. The example operator for Class 1 also demonstrates the effect of deletion on model components. Class 2 consists of operations that affect one or more

components, and induce provenance consisting of a single provenance link terminating at an external source referent. Class 3 consists of operations that affect one or more components, and induce provenance consisting of a single provenance link terminating at another component (not an external source referent). Class 4 consists of queries. Although queries create one or more components and the induced provenance links terminate at other components, queries may induce more than one provenance link, so we address queries separately.

For each operator described, let $u \in U$ describe a user of M^L, and $t \in TS$ be the time at which u applies the stated operation to M^L, resulting in revision $\Re^L$. We assume for simplicity that each $t \in TS$ is integral, and that an operation applied at time t has a successor applied at $t + 1$, and so on. In the following definitions, we omit specification of the tests for success prior to executing each operation. These are the same criteria for success that we defined for corresponding MMP operators. We assume that changes to components in d^L occur immediately before the actions specified in the definitions below. Where appropriate, we include effects on *Ts* and *Te* of components in d^L.

6.1.1.1   Class 1: Drop Attribute

$$\Re^L(d^L, E^L, \text{Drop Attribute}(r_j, a_l), u, t_{op}) = d^L$$

with the following changes:

$a_{j,l}.Te = t_{op}.$

$\forall v_{j,k,l} \in V_{j,k}^L, 1 \le k \le |T_{j,k}^L|, v_{j,k,l}.Te = t_{op}.$

6.1.1.2 Class 2: Insert Tuple

$$\Re^L(d^L, E^L, \text{Insert Tuple}(r_j, (a_1, v_1, \ldots, a_m, v_m), s), u, t_{op}) = d^L$$

with the following changes:

Create a temporary relation r_{temp} with the same schema as r_j.

Create in r_{temp} a tuple t.

$\forall a_n$ specified in $\Re^L$, create an attribute value in t, attribute a_n with value v_n.

For all other attributes in the schema of r_{temp}, set the corresponding attribute value in t to NULL.

$t.Ts = t_{op}$, and $t.Te = \infty$.

$\forall v_n \in t, v.Ts = t_{op}, v.Te = \infty$.

$r_j = r_j \cup r_{temp}$.

/* add a provenance link from the target tuple to the external source */

$$E^L = E^L \cup e(myID(t), myID(s)).$$

$$e.op = \text{``Insert Tuple''}, e.user = u, e.time = t_{op}.$$

Delete r_{temp}.

6.1.1.3 Class 3: Paste Tuple

We assume that an attribute value argument is supplied for each attribute in the target relation schema. For any attribute values v with $v.Te \neq \infty$, use the value NULL instead of v.

$$\Re^L(d^L, E^L, \text{Paste Tuple}(r_j, r_{js}, t_{js,ks}), s), u, t_{op}) = d^L$$

with the following changes:

Create a temporary relation r_{temp} with the same schema as r_j.

Create in r_{temp} a tuple t, which is a copy of $t_{js,ks}$.

$t.Ts = t_{op}$, and $t.Te = \infty$.

$\forall v_n \in t, v.Ts = t_{op}, v.Te = \infty$.

$r_j = r_j \cup r_{temp}$.

/* add a provenance link from the target tuple to the external source */

$$E^L = E^L \cup e(myID(t), myID(t_{js,ks})).$$

$$e.op = \text{"Paste Tuple"}, e.user = u, e.time = t_{op}.$$

Delete r_{temp}.

6.1.1.4   Class 4: Queries

$\Re^L(r_{out} = (\pi\sigma(r^{in}_{1,1} \times \ldots r^{in}_{1,n_1}) \cup \ldots \cup \pi\sigma(r^{in}_{M,1} \times \ldots \times r^{in}_{M,n_M})), u, t_{op}) =$

if the query is well-formed and $r_{out} \notin R^L \wedge \forall r^{in}_{X,Y}, 1 \leq X \leq M, 1 \leq Y \leq nM, r^{in}_{X,Y} \in R^L$,

then

/* remove all portions of the input relations that have been deleted */

for each relation r mentioned in the query, substitute in the query a relation r' formed as follows:

$r' = \pi_X(\sigma_{Te \neq \infty}(r))$, where X is the set of attributes for which $Te \neq \infty$

for all remaining attribute values v in r', if $v.Te \neq \infty, v = NULL$

/* execute the resulting query */

Create r_{out} in D^L as the query result as defined by the semantics of relational algebra queries.

$r_{out}.Ts = t_{op}$ and $r_{out}.Te = \infty$.

For each attribute a in r_{out}, $a.Ts = t_{op}$ and $a.Te = \infty$.

For each tuple t in r_{out}, $t.Ts = t_{op}$ and $t.Te = \infty$.

For each attribute value v in r_{out}, $v.Ts = t_{op}$ and $v.Te = \infty$.

/* add a provenance link from the new relation to all input relations */

$$E^L = E^L \cup \bigcup_{X=1}^{M} e(myID(r_{out}), \bigcup_{Y=1}^{nX} myID(r_{X,Y}^{in})).$$

$$e.op = ``\langle querytext \rangle'', e.user = u, e.time = t_{op}.$$

/* add a provenance link from each new tuple to its parent tuples */

$$\forall t_{j,k} \in T_j^L, E^L = E^L \cup \bigcup_{X=1}^{M} e(myID(t_{j,k}), \bigcup_{Y=1}^{nX} myID(t_{(X,Y),u}))$$

such that $t_{(X,Y),u}$ was the tuple from $r_{X,Y}^{in}$ that contributed to forming $t_{j,k}$.

$$e.op = ``\langle query\ text \rangle'', e.user = u, e.time = t_{op}.$$

6.2 Faithful Support of MMP by MMP^L

MMP^L faithfully supports MMP if, beginning with an instance M of MMP, and a corresponding instance M^L of MMP^L, where $\Lambda(M) = M^L$, then after application of an arbitrary composition of revisions $\Re_1, \Re_2, \ldots \Re_n$ to M, yielding M', and application of a corresponding composition of revisions $\Re_1^L, \Re_2^L, \ldots \Re_n^L$ to M^L, yielding M'^L, we have M' and M'^L such that $\Lambda(M') = M'^L$. The commutative diagram corresponding to this definition is shown in Figure 6.1.

Figures 6.2 through 6.6 show an example of faithful support of an MMP instance by an MMP^L instance. In Figure 6.2, we show the creation of two external source referents and a single relation in an MMP instance M. At the bottom of the figure, we show a mapping to an initial MMP^L instance M^L. Note that Ts of the relation in M^L is set by Λ to the timestamp on the face (time 3) at which the relation first appears in M, and Te of the relation is set to ∞, representing that it has not been deleted. These two model instances on the left and right side of 6.2 correspond to the upper and lower left instances shown in Figure 6.1, and the Λ mapping between them corresponds to the downward arrow on the left of Figure 6.1.
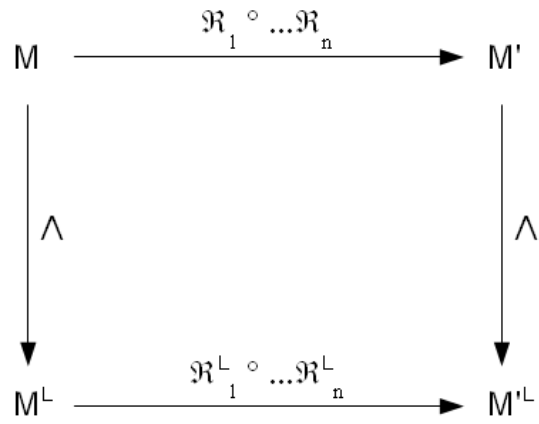
Figure 6.1: Commutative Diagram for MMP
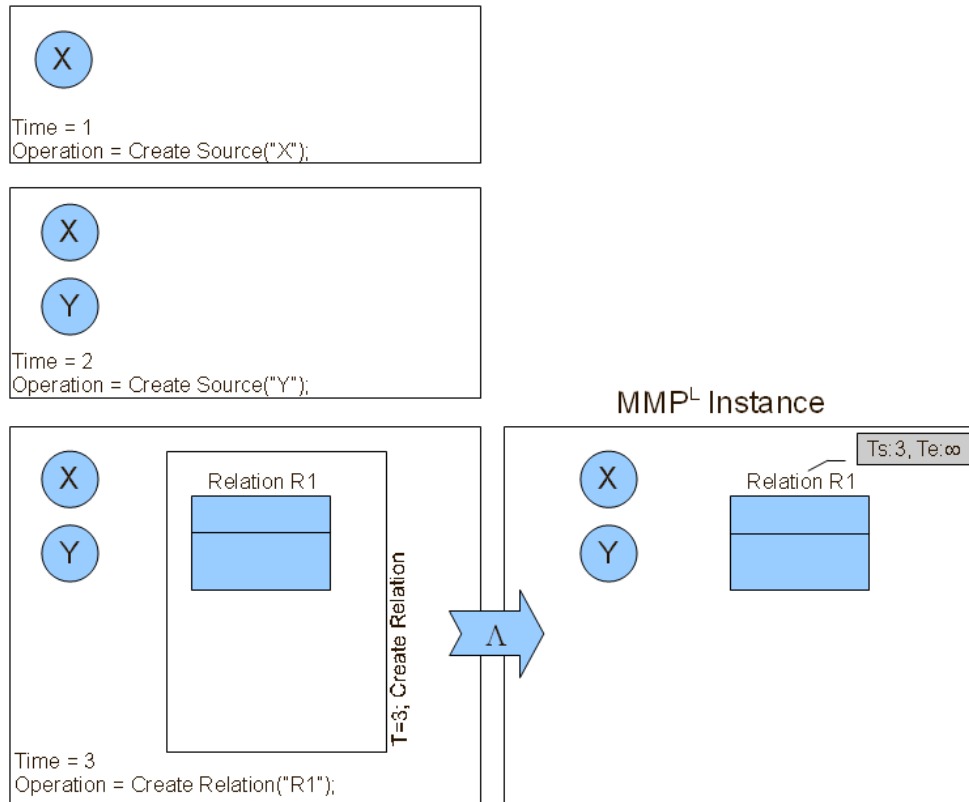
Figure 6.2: Faithful Support Example - Part 1
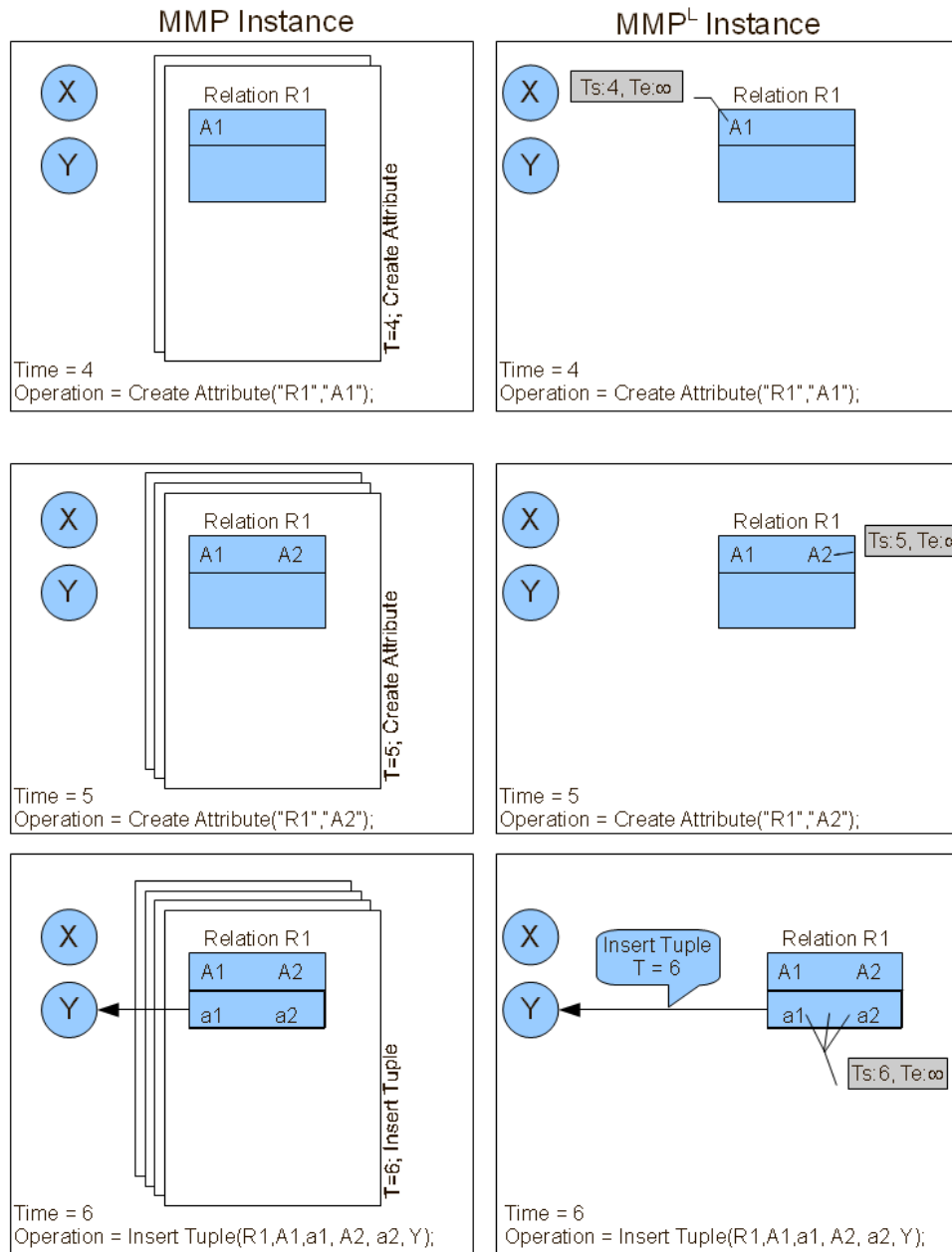
Figure 6.3: Faithful Support Example - Part 2

Figure 6.3 shows three subsequent time steps in the parallel evolution of M and M^L. At each time step, the same operation is applied to each. At time 4, attribute $A1$ is added to the corresponding relations via a Create Attribute operation. In M, the face induced by this operation is labeled with the timestamp 4. In M^L, the start time of the new attribute is set to timestamp 4, and its end time is set to ∞, indicating that it has not been deleted. Note that in the images of M^L, we show only the most recently changed Ts and Te values for clarity. We omit the user specifier from labels.

At time 5, a second attribute is added. At time 6, a tuple is inserted. In M, the new face induced by the operation is labeled with timestamp 6. In M^L, the Ts properties for the new tuple and its attribute values are set to 6, and Te for each is set to ∞. In M, the induced provenance link from the new tuple to its source is added. In M^L, the corresponding provenance edge is introduced and labeled with the operation and timestamp.

Figure 6.4 shows three additional time steps for the corresponding model instances. At time 7, another tuple is inserted. In this example, one of the new attribute values is NULL. Note that in this case, the new tuple and the non-NULL attribute value in M^L both receive Ts and Te values as before. At time 8, value $a1$ from the first tuple in each relation is pasted into the NULL value in the second tuple. In M, this induces a provenance link from the new value to its source in the prior face. In M^L, the corresponding provenance edge originates at the new attribute value and terminates at the corresponding source attribute value. The new provenance edge in M^L is labeled with the same operation and timestamp as the new face in M. At time 9, the first tuple in each relation is deleted. In M, this causes the deleted tuple t to have its value of $Expired_t(t)$ set to True, and the attribute values v_1 and v_2 in t to have their $Expired_v(v_1)$ and $Expired_v(v_2)$ values set to True (denoted by "x" at the top right of each in the figure). In M^L, deletion causes Te for the deleted tuple and its attribute values to be set to the time at which the operation was applied (time 9).
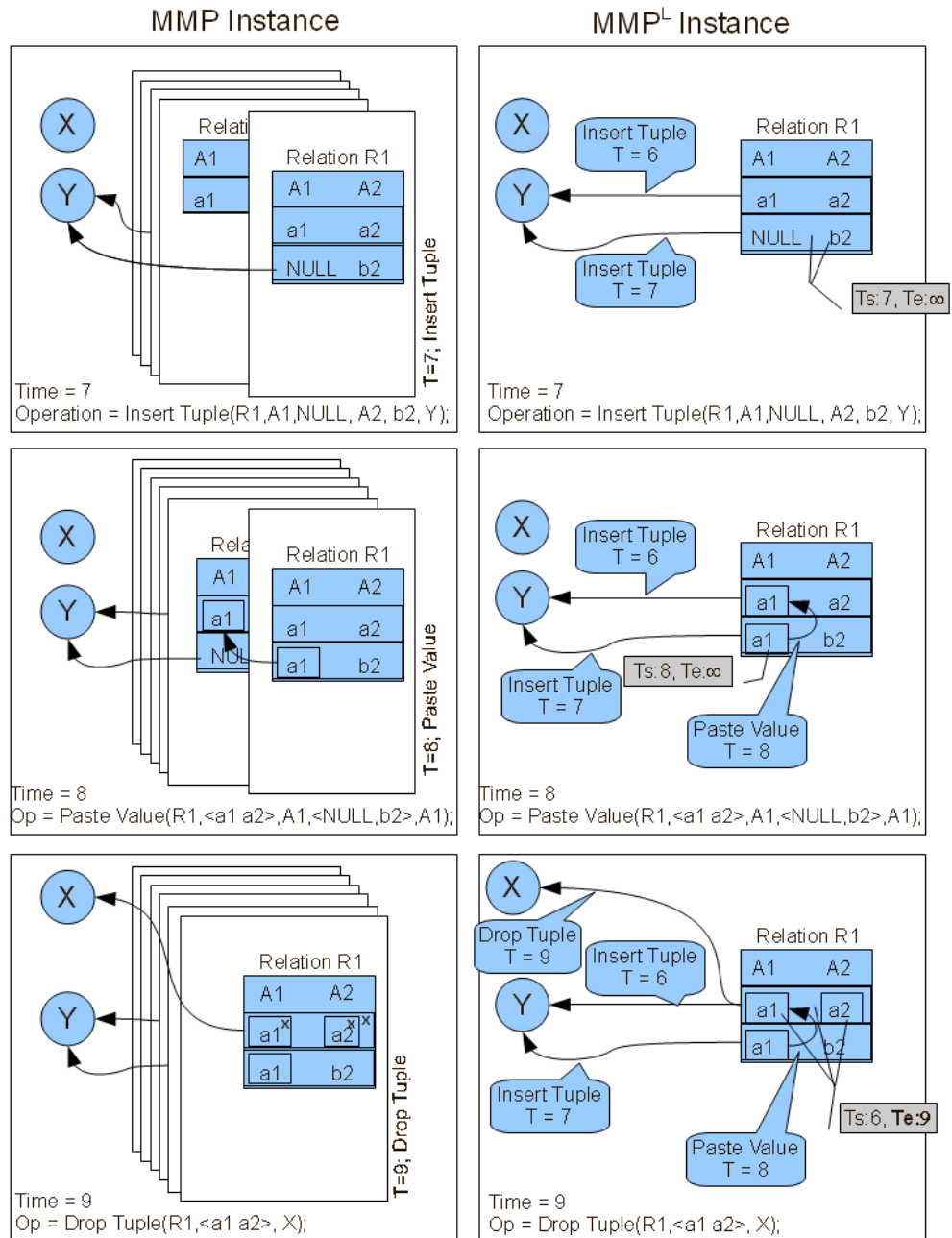
Figure 6.4: Faithful Support Example - Part 3

Figure 6.5: Faithful Support Example - Mapping M to M^L After Time 9
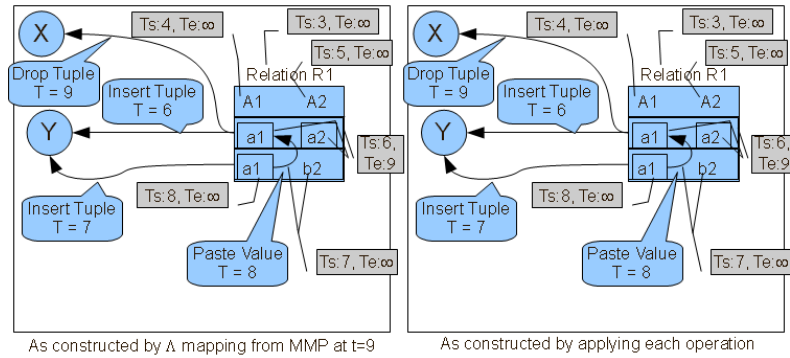
Figure 6.6: Comparing $\Lambda(M)$ to M^L

Figure 6.5 shows mapping M to an M^L instance using Λ after the application of the operation at time 9. This mapping corresponds to the rightmost downward arrow in Figure 6.1. Using the steps in our definition of the Λ mapping, Step 1a produces the relation $R1$ and its contents (shown at the top of the figure). Step 1b defines the Ts and Te values for each component created in Step 1a. Step 2 introduces the external source referents defined in M. Finally, Step 3 introduces the provenance links and their labels resulting from the applied operations.

Figure 6.6 shows at left the result of the mapping applied in Figure 6.5, and shows at right the M^L instance from the bottom of Figure 6.4. Inspection shows that the two are identical. Thus in this example, the commutativity diagram of Figure 6.1 holds.

We now prove faithful support of MMP by MMP^L by induction on the number of revisions applied to the MMP instance.

6.2.1 Basis Case for Induction

Consider Figure 6.1. If zero revisions are applied to M, then $M' = M$. Similarly, if zero revisions are applied to M^L, then $M'^L = M^L$. So given that $\Lambda(M) = M^L$, it

must be that $\Lambda(M') = M'^L$.

6.2.2   Inductive Case

For the inductive case, we assume a composition of n corresponding revisions $\Re_1, \Re_2, \ldots, \Re_n$, and $\Re_1^L, \Re_2^L, \ldots, \Re_n^L$ have been applied to an instance of MMP and corresponding instance of MMP^L to arrive at M and M^L, respectively, such that when applied at time n, $\Lambda(M) = M^L$. Our proof shows that if we extend the applied sequence by one additional revision, $\Re_{n+1}$ and its corresponding revision $\Re_{n+1}^L$, respectively yielding M' and M'^L, that $\Lambda(M') = M'^L$. We consider the data and provenance portions of M and M^L separately.

6.2.2.1   Data Portion of Inductive Case

For the data portion, the induction hypothesis states that

- the current face of M, d_n, has a corresponding set of relations to those in d^L, and each of those relations has the same attributes

- each corresponding relation mentioned above has corresponding sets of tuples, such that each corresponding relation has the same number of tuples, and that corresponding pairs of tuples have the same attribute values in the same attributes

- each component c in d^L has $c.Ts = time(d_x)$, where x is the first face in M in which c appeared

- each component c in d^L has $c.Te = time(d_y)$, where y is the time at which c was deleted, or $c.Te = \infty$ otherwise (recall that all expired components from M appear in d^L)

Now suppose that we apply an operation to M, yielding M' with current face d_{n+1}, and we apply the same operation to M^L, yielding M'^L with database d'^L. We consider four operations, one from each class of operations defined above.

Insert Tuple. Consider the case where the operation applied in the inductive step is an Insert Tuple operation. The uppermost box in Figure 6.7 shows the derivation of $\Lambda(\Re_{InsertTuple}(M))$ and the resulting new tuple $t_{n+1,j,k}$ in the target relation r_j of d_{n+1}. Applying Λ to the revised M creates M'^L, which contains a new tuple, $t^L = sameComponentAs(t_{n+1,j,k})$, with identical attribute values to $t_{n+1,j,k}$ for all attributes.

The uppermost box in Figure 6.8 shows the derivation of $\Re^L_{InsertTuple}(\Lambda(M))$, which unions into r^L_j a tuple t^L with identical attribute values to the $t_{n+1,j,k}$ described above. Note that t^L mentioned in Figure 6.8 corresponds to $t_{n+1,j,k}$ mentioned in Figure 6.7, and their correspondence is defined by $t^L = sameComponenAs(t_{n+1,j,k})$. The two tuples and their attribute values correspond and are thus identical, and are members of corresponding relations. As shown in the figure, the start and end times of the two prospective tuples t^L are also identical. Thus the commutativity diagram holds for the Insert Tuple operation. We claim without proof that the commutativity diagram also holds for all other operations in the same operation class.

Drop Attribute. Consider the case where the operation applied in the inductive step is a Drop Attribute operation. The second box in Figure 6.7 shows the derivation of $\Lambda(\Re_{DropAttribute}(M))$. The resulting M' differs from M only in the value of $Expired_a(a_l)$ and the value of $Expired_v(v)\forall v$ in that attribute. The application of Λ sets $Te = t_{op}$ for the corresponding attribute of the operation in M'^L, and for all attribute values in that attribute.

The second box in Figure 6.8 shows the derivation of $\Re^L_{DropAttribute}(\Lambda(M))$, which sets $Te = t_{op}$ for the target attribute of the operation, and for all attribute values in that attribute. The two attributes are thus identical, as are their attribute val-

ues. Thus commutativity holds for the Drop Attribute operation. We claim without proof that the diagram also holds for all other operations in the same operation class.

Paste Tuple. Consider the case where the operation applied in the inductive step is a Paste Tuple operation. The third box in Figure 6.7 shows the derivation of $\Lambda(\Re_{PasteTuple}(M))$ and the resulting new tuple $t_{n+1,j,k}$ in the target relation r_j of d_{n+1}. Applying Λ to the revised M creates M'^L, which contains a new tuple, $t^L = sameComponentAs(t_{n+1,j,k})$, with identical attribute values to $t_{n+1,j,k}$ for all attributes.

The third box in Figure 6.8 shows the derivation of $\Re^L_{PasteTuple}(\Lambda(M))$, which unions into r^L_j a tuple t^L with identical attribute values to the $t_{n+1,j,k}$ described above. Note that t^L mentioned in Figure 6.8 corresponds to $t_{n+1,j,k}$ mentioned in Figure 6.7, and their correspondence is defined by $t^L = sameComponentAs(t_{n+1,j,k})$. The two tuples and their respective attribute values correspond and are thus identical, and are members of corresponding relations. As shown in the figures, the start and end times of the two prospective tuples t^L are also identical. Thus the commutativity diagram holds for the Paste Tuple operation. We claim without proof that the commutativity diagram also holds for all other operations in the same operation class.

Queries. Finally, consider the case where the operation applied in the inductive step is a relational algebra query. The fourth box in Figure 6.7 shows the derivation of M'^L due to the query revision. The query results in creation of the new relation r_{out} in M'. For the result relation r_{out} and all its components, the appropriate expired function is set to False. The resulting model face is given a label that includes the time at which the revision was applied, t_{op}. Applying Λ to M' results in M'^L, which differs from M^L only by the addition of the new relation r^L_{out}. This new relation and all its components have Ts set to t_{op} and Te set to ∞. The new relation and all its components correspond to those in M' that induced them in M'^L, and their correspondence is defined by the $sameComponentAs$ function in each case.

The fourth box in Figure 6.8 shows the derivation of M'^L by application of the same query applied in Figure 6.7. Note that r_{out}^L and all its components have the same names and values as r_{out} and its components in M', because the queries applied in each case have the same semantics. Note also that Ts and Te are set to the same values by $\Re^L$ as those set by Λ applied to M'. Thus all components resulting from the queries are identical and have the same Ts and Te values. The commutativity diagram thus holds for query operations.

6.2.2.2 Provenance Portion of Inductive Case

For the provenance portion of our commutativity diagram, we argue the induction case on an operation-by-operation basis.

Insert Tuple. Consider the case where the operation applied in the inductive step is an Insert Tuple operation. The uppermost box in Figure 6.9 shows the new edge in E'^L resulting from $\Lambda(\Re_{InsertTuple}(M))$. $\Re$ performed on M creates a new face d_{n+1} initially identical to d_n, then adds to the appropriate relation in d_{n+1} the new tuple $t_{n+1,j,k}$, its attribute values $V_{n+1,j,k}$, and adds a single provenance link $l_p(t_{n+1,j,k}, s)$. Applying Λ to the revised M creates M^L at timestep $n+1$, denoted M'^L in the figure. E'^L includes a single new edge. The uppermost box in Figure 6.10 shows the new edge in E_{n+1}^L resulting from $\Re_{InsertTuple}^L(\Lambda(M))$. We now compare the new edge shown in Figure 6.9 to the new edge resulting from $\Re_{InsertTuple}^L(\Lambda(M))$ in 6.10. Note that t^L mentioned in Figure 6.10 corresponds to $t_{n+1,j,k}$ mentioned in Figure 6.9, and their correspondence is defined by $t^L = sameComponenAs(t_{n+1,j,k})$. Substituting, we find that the new edges are identical, so the commutativity diagram holds for the Insert Tuple operation. We claim without proof that the commutativity diagram also holds for all other operations in the same operation class

Drop Attribute. Consider the case where the operation applied in the inductive step is a Drop Attribute operation. As shown in the second box in Figure 6.9 and

$$M \xrightarrow{\quad \Re \quad} M'$$

$$\downarrow \Lambda(M, n+1)$$

$\Re_{Insert\ Tuple}$

- unions tuple $t_{n+1,j,k}$, with specified attribute values $V_{n+1,j,k}$, into r_j, resulting in M'
- sets $Expired_t(t_{n+1,j,k}) = False$, and for all v in $t_{n+1,j,k}$, sets $Expired_v(v) = False$
- labels new face with time = t_{op}

$\Lambda(M')$

- maps $t_{n+1,j,k}$ in r_j to identical t^L in r_j^L, where $r_j^L = sameComponentAs(r_j)$, such that $t^L = sameComponentAs(t_{n+1,j,k})$
- sets $t^L.Ts = t_{op}$, $t^L.Te = \infty$, and for all v^L in t^L, $v^L.Ts = t_{op}$, $v^L.Te = \infty$

$\Re_{Drop\ Attribute}$

- for target attribute a_i in relation r_j, sets $Expired_a(a_i) = True$
- for all attribute values v in column a_i, sets $Expired_v(v) = True$
- labels new face with time = t_{op}

$\Lambda(M')$

- sets $a_i^L.Te = t_{op}$, where $a_i^L = sameComponentAs(a_i)$
- for all attribute values v^L in column a_i^L, sets $v^L.Te = t_{op}$

$\Re_{Paste\ Tuple}$

- unions $t_{n+1,j,k}$, a copy of tuple t_{ks} from relation r_{js}, into relation r_j
- sets $Expired_t(t_{n+1,j,k}) = False$, and for all v in $t_{n+1,j,k}$, sets $Expired_v(v) = False$
- labels new face with time = t_{op}

$\Lambda(M')$

- maps $t_{n+1,j,k}$ in r_j to identical t^L in r_j^L, such that $t^L = sameComponentAs(t_{n+1,j,k})$
- sets $t^L.Ts = t_{op}$, $t^L.Te = \infty$, and for all v^L in t^L, $v^L.Ts = t_{op}$, $v^L.Te = \infty$

$\Re_{Query}$

- creates result relation r_{out} in M'
- for all components c in r_{out}, sets $Expired(c) = False$
- labels new face with time = t_{op}

$\Lambda(M')$

- creates r_{out}^L in M'^{L} such that $r_{out}^L = sameComponentAs(r_{out})$
- creates all components c^L in r_{out}^Ls identical to corresponding components in r_{out} such that $c^L = sameComponentAs(r_{out})$
- sets Ts = t_{op}, Te = ∞, for r_{out}^L and its components

Figure 6.7: Derivation of $\Lambda(\Re_{operation}(M))$ (Data portion)

M

$\Lambda(M,n)$

M^L ——————— $\mathfrak{R}^L$ ——————→ M'^L

$\mathfrak{R}^L_{Insert\ Tuple}$ unions a tuple t^L, with specified attribute values v^L
and NULLS elsewhere into r_j^L; sets
$t^L.Ts = t_{op}$, $t^L.Te = \infty$, and for all v^L in t^L, $v^L.Ts = t_{op}$, $v^L.Te = \infty$

$\mathfrak{R}^L_{Drop\ Attribute}$ for target attribute a_i^L in relation r_j^L, sets $a_i^L.Te = t_{op}$.
For all attribute values v^L in column a_i^L, sets $v^L.Te = t_{op}$

$\mathfrak{R}^L_{Paste\ Tuple}$ unions t^L, a copy of tuple t_{ks}^L from relation r_{js}^L, into relation r_j^L;
sets $t^L.Ts = t_{op}$, $t^L.Te = \infty$, and for all v^L in t^L, $v^L.Ts = t_{op}$, $v^L.Te = \infty$

$\mathfrak{R}^L_{Query}$ creates result relation r_{out}^L in M'^L
for r_{out}^L and all components c in r_{out}^L, sets $Ts = t_{op}$, $Te = \infty$

Figure 6.8: Derivation of $\mathfrak{R}^L_{operation}(\Lambda(M))$ (Data portion)

$$M \xrightarrow{\quad \mathfrak{R} \quad} M'$$

$$\downarrow \Lambda(M, n+1)$$

$$M'^{L}$$

$\mathfrak{R}_{\text{Insert Tuple}}$ adds to M provenance link $I_p(t_{n+1j,k}, s)$, resulting in M'

$\Lambda(M')$ then adds to E'^L the new corresponding edge
e(myID(sameComponentAs($t_{n+1j,k}$)), myID(sameSourceAs(s)))

$\mathfrak{R}_{\text{Drop Attribute}}$ adds no provenance links to M, resulting in M'.
The expired flag for the dropped attribute is changed to TRUE.
$\Lambda(M')$ thus adds nothing to E'^L, but sets T_{stop} for the attribute to t_{op}

$\mathfrak{R}_{\text{paste Tuple}}$ adds to M provenance link $I_p(t_{n+1j,k}, t_{n,js,ks})$ resulting in M'

$\Lambda(M')$ then adds to E'^L the new corresponding edge
e(myID(sameComponentAs($t_{n+1j,k}$)), myID(sameComponentAs($t_{n,js,ks}$)))

$\mathfrak{R}_{\text{Query}}$ adds to M, resulting in M'

provenance link set $\bigcup\limits_{X=1}^{M} I_p(r_{n+1j}, \bigcup\limits_{Y=1}^{nX} r^{in}_{X,Y})$ for relation r_{n+1j} and

provenance link set $\bigcup\limits_{X=1}^{M} I_p(t_{n+1j,k}, \bigcup\limits_{Y=1}^{nX} t_{n,jxm(Y)})$ for tuples $t_{n+1j,k}$ in T_{n+1j}

$\Lambda(M')$ then adds to E'^L the new corresponding edges

a provenance edge set $\bigcup\limits_{X=1}^{M} e(mV(sCA(r_{n+1j}), n+1), \bigcup\limits_{Y=1}^{nX} mV(sCA(r^{in}_{X,Y}), n))$ and

a provenance edge set $\bigcup\limits_{X=1}^{M} e(mV(sCA(t_{n+1j,k}), n+1), \bigcup\limits_{Y=1}^{nX} mV(sCA(t_{n,jxm(Y)}), n))$

Figure 6.9: Derivation of $\Lambda(\mathfrak{R}_{operation}(M))$ (Provenance portion)

M

$\Lambda(M,n)$

M^L ——————— $\Re^L$ ——————→ M'^L

$\Re^L_{\text{Insert Tuple}}$ adds to M^L the new edge $e(myID(t^L), myID(s^L))$, resulting in M'^L

$\Re^L_{\text{Drop Attribute}}$ adds nothing to edges in M^L resulting in M'^L
The value of T_{stop} for the dropped attribute changes to t_{op}

$\Re^L_{\text{Paste Tuple}}$ adds to M^L the new edge $e(myID(t^L), myID(t^L_{js,ks}))$ resulting in M'^L

$\Re^L_{\text{Query}}$ adds to M^L, resulting in M'^L

a provenance edge set $\bigcup\limits_{X=1}^{M} e(myID(r_{out}{}^L), \bigcup\limits_{Y=1}^{nX} myID(r^{in}_{X,Y}{}^L))$

a provenance edge set $\bigcup\limits_{X=1}^{M} e(myID(t_{j,k}{}^L), \bigcup\limits_{Y=1}^{nX} myID(t_{(X,Y),k}{}^L))$

Figure 6.10: Derivation of $\Re^L_{operation}(\Lambda(M))$ (Provenance portion)

the second box in Figure 6.10, no edges are added. The only change, made in both versions of d^L, is that Te for the dropped attribute is set to the timestamp of the operation. Thus our commutativity diagram holds for Drop Attribute, and we claim for all other operations in its class.

Paste Tuple. Consider the case where the operation applied in the inductive step is a Paste Tuple operation. The induced edges to be compared are shown in the third boxes in Figure 6.9 and Figure 6.10. The comparison here is identical to that in the Insert Tuple case, except for the terminal of the new provenance links. In Figure 6.10, this terminal is $myID(t^L_{js,ks})$. In Figure 6.9, the terminal is $myID(sameComponentAs(t_{n,js,ks}))$. Note that $t^L_{js,ks} = sameComponentAs(t_{n,js,ks})$, by our definition of correspondi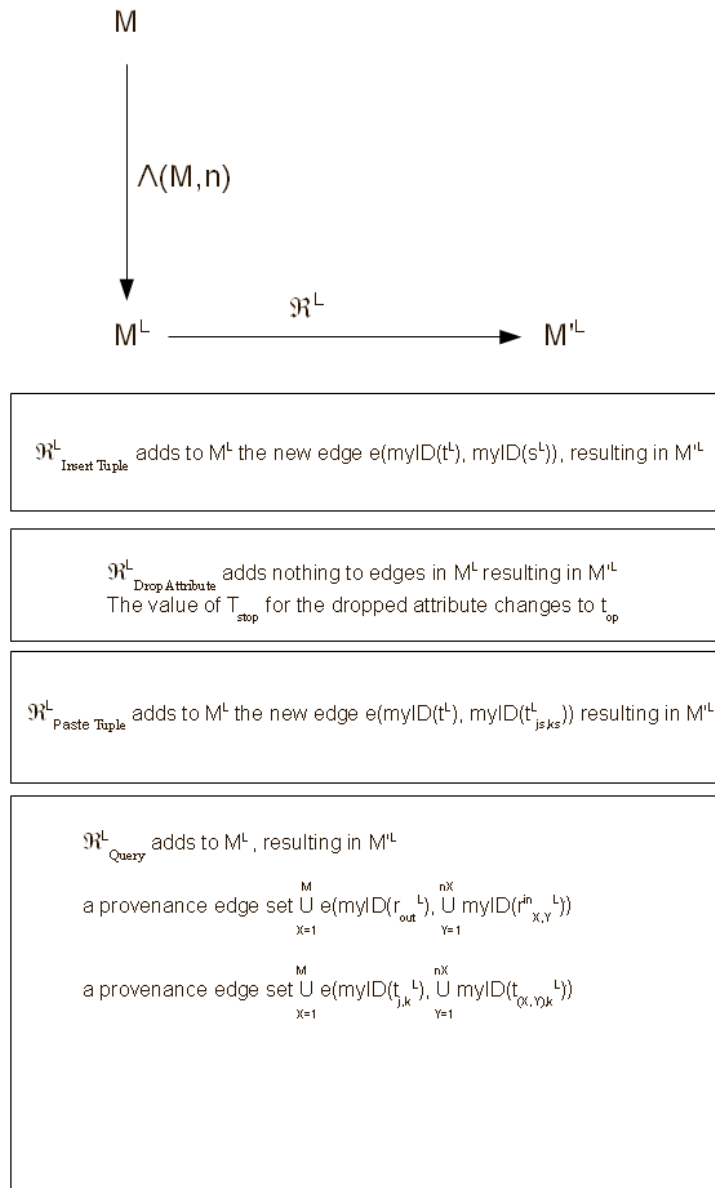ng components. Substituting, we find that these terminals are identical. As a result, our commutativity diagram holds for Paste Tuple and, we claim, for all operations in its class.

Queries. Consider the case where the operation applied in the inductive step is a query. As with the preceding cases, we construct $\Lambda(\Re_{Query}(M))$, and construct $\Re^L_{Query}(\Lambda(M))$. The fourth box in Figure 6.9 shows the application of a query and the derivation of $\Lambda(\Re_{Query}(M))$ and the resulting new edges in E'^L. $\Re$ performed on M creates a new face $d_n + 1$ identical to d_n, then adds to it the result relation of the query $r_{n+1,j}$, the tuples in the new relation $t_{n+1,j,k} \in T_{n+1,j}$, the attributes in the new relation $a_{n+1,j,l} \in A_{n+1,j}$, and the attribute values in each new tuple $V_{n+1,j,k}, 1 \leq k \leq |T_{n+1,j}|$. $\Re$ also adds provenance links for the new relation and its tuples, as defined in Section 3.6.4. Applying Λ to M' creates M'^L at timestep $n + 1$. The fourth box in Figure 6.10 shows the derivation of $\Re^L_{Query}(\Lambda(M))$ and the constituents of the resulting E'^L. $\Lambda(M)$ creates M^L. $\Re^L$ applied to M^L induces M'^L. We now compare the new edges resulting from $\Lambda(\Re_{query}(M))$ in 6.9 to the new edges resulting from $\Re^L_{query}(\Lambda(M))$ in 6.10.

Note that r^L_{out} mentioned in Figure 6.10 corresponds to $r_{n+1,j}$ mentioned in Figure

6.9, and their correspondence is defined by

$$r^L_{out} = sameComponentAs(r_{n+1,j}).$$

Thus $myID(r^L_{out})$ is the same vertex in M'^L as

$$myVertex(sameComponentAs(r_{n+1,j}, n+1))$$

in each edge induced in both figures. Also, note that $r^{inL}_{X,Y}$ mentioned in 6.10 corresponds to $r^{in}_{X,Y}$ in Figure 6.9, and their correspondence is defined by

$$r^{inL}_{X,Y} = sameComponentAs(r^{in}_{X,Y}).$$

Thus $myID(r^{inL}_{X,Y})$ is the same vertex in M'^L as

$$myVertex(sameComponentAs(r^{in}_{X,Y}))$$

in each edge induced in both figures. Note also that $t^L_{j,k}$ mentioned in Figure 6.10 corresponds to $t_{n+1,j,k}$ mentioned in Figure 6.9, and their correspondence is defined by

$$t^L_{j,k} = sameComponentAs(t_{n+1,j,k}).$$

Thus $myID(t^L_{j,k})$ is the same vertex in M'^L as

$$myVertex(sameComponentAs(t_{n+1,j,k}))$$

in each edge in the figures. Finally, $t^L_{(X,Y)k}$ mentioned in 6.10 corresponds to $t_{n,jy,m(Y)}$ mentioned in Figure 6.9, and their correspondence is defined by

$$t^L_{(X,Y)k} = sameComponentAs(t_{n,jy,m(Y)}).$$

Thus $myID(t^L_{(X,Y)k})$ is the same vertex as

$$myVertex(sameComponentAs(t_{n,jy,m(Y)}))$$

in each induced edge. Substituting these correspondences, we find that the new edges induced in Figure 6.10 are identical to those induced in Figure 6.9, so our commutativity diagram holds for query operations.

In Section 6.2.1, we proved the basis case for induction by showing that for zero revisions, our commutativity diagram shown in Figure 6.1 holds. In Section 6.2.2.1, we proved the inductive case for the data portion of our model. In Section 6.2.2.2, we proved the inductive case for the provenance portion of our model. From these proofs, we conclude that for any composition of any number of revisions, the commutativity diagram in Figure 6.1 holds. As a result, we claim that an MMP^L instance faithfully supports a corresponding MMP instance.

6.3 Efficiency of the Logical Model

The MMP^L model allows for less redundancy in data storage than MMP because data is stored only once, rather than duplicated on each model face. However, this savings comes at a cost. In MMP^L, we must provide start and end time metadata for each database component, while in MMP we need to provide only the means to define $Expired()$ for each component. Also in MMP^L we must provide time, operation, and user labels for each provenance edge, while in MMP this information is only provided once for each model face.

Assume an MMP instance M and a corresponding MMP^L instance M^L. Assume that the cost of storing a data component c^L in M^L and the cost of storing the corresponding component $c = sameComponentAs(c^L)$ in M are the same. We define that cost to be k_{data}. Assume that the cost of storing the definition of $Expired(c)$ is a constant, $k_{expired}$, and that the total cost of storing $c^L.Ts$ and $c^L.Te$ is a constant $2 \times k_{expired}$.

Note that each provenance link in M has a corresponding provenance edge in M^L, and that each corresponding pair of links has the same number of origins and

terminals. Assume that the cost of storing link l^L in M^L and its corresponding link l in M differ only in a constant cost, k_{label}, which is the cost of the label attached to l^L that does not appear on l. Assume that the cost of the label attached to each face in M is also k_{label}, because both types of labels carry the same information.

Suppose n operations have been applied to M and M^L. Suppose that for each such operation, k_{new} components are added to the database. Recall that each DDL, DML, and DCL operation induces only a single provenance link with a single origin and terminal in M and only a similar single provenance edge in M^L. Recall that each query induces one or more provenance links (edges) for the result relation and one or more provenance links (edges) for each tuple in that relation. Suppose that the amount of storage for a link, on average, is k_{link}. Suppose that the ratio of queries to total operations is r_o. Suppose that the average number of tuples defined in a query result is t.

We characterize the amount of data storage S_{data}, label storage S_{label}, and link storage S_{link} for M as follows:

/* each data face has k_{new} components more than the prior face */

/* and each component takes $k_{data} + k_{expired}$ storage space */

$$S_{data} = (k_{data} + k_{expired})(\sum_{x=1}^{n} n \times k_{new}) = n(n+1)(k_{new})(k_{data} + k_{expired})/2.$$

/* one label is attached to each face */

$$S_{label} = n(k_{label}).$$

/* 1 link for each non-query, plus $1 + t$ links for each query */

$$S_{link} = n(1 - r_o)k_{link} + n(r_o)k_{link}(1 + t) = n(k_{link})(2 - r_o + t).$$

We characterize the amount of data storage S^L_{data}, label storage S^L_{label}, and link storage S^L_{link} for M^L as follows:

/* each operation induces k_{new} components */

/* and each component takes $k_{data} + 2k_{expired}$ storage space */

$$S_{data}^L = n(k_{new})(k_{data} + 2k_{expired}).$$

/* one label attached to each provenance edge */

$$S_{label}^L = n(1 - r_o)k_{label} + n(r_o)k_{label}(1 + t).$$

/* same storage for links as in M */

$$S_{link}^L = n(1 - r_o)k_{link} + n(r_o)k_{link}(1 + t) = n(k_{link})(2 - r_o + t).$$

Subtracting, we characterize the additional cost of M over M^L due to the three storage contributions:

/* extra cost in M of data and timestamps */

$$\Delta S_{data} = n(n + 1)(k_{new})(k_{data} + k_{expired})/2 - n(k_{new})(k_{data} + 2k_{expired})$$

/* extra cost in M of labels */

$$\Delta S_{label} = n(k_{label}) - (n(1 - r_o)k_{label} + n(r_o)k_{label}(1 + t)).$$

/* extra cost in M of links */

$$\Delta S_{link} = 0.$$

Consider ΔS_{data}. For small values of n (very few applied operations), if the number of components introduced by each operation is large, M^L may have larger storage cost than M because of the extra timestamp value Ts. For example, suppose $n = 1$ and k_{new} is large. Then we see that ΔS_{data} is negative, dominated by $k_{new} \times 2k_{expired}$. This would means that M has better data efficiency than M^L. However, given that each operation introduces a new face in M, n will tend to be much larger than one, so ΔS_{data} will typically be dominated by the $n(n+1)(k_{new})(k_{data}+k_{expired})/2$ term. This means that M^L will typically be $O(n^2)$ more storage-efficient than M.

Now consider ΔS_{label}. If almost all operations are DDL, DML, or DCL operations, then r_o is near zero, so ΔS_{label} is nearly zero. As a result, M and M^L would have similar costs for label storage. However, suppose all operations are queries, so that r_o is one. Then ΔS_{label} is dominated by the average number of tuples in result relations. If t, the average tuples per relation, is large, this tends to make ΔS_{label} a large negative value, so that M^L would be less storage efficient than M with regard to labels.

We note that other logical models are possible that may address this issue of label overhead in a query-dominated workload. For example, a model where operation labels are stored only once, instead of once per provenance link, would eliminate the concern over ΔS_{label}.

6.4 Chapter Summary

In this chapter, we proposed a logical model for data and provenance. We defined the structure and language of this model, and defined a transformation to create an instance of this logical model from an instance of MMP. We defined classes of operations in the language of MMP such that all members of a class have similar effects on the model instance. Using sample operations from each class of language operations, and using this transformation, we showed that an instance of the logical model faithfully supports a corresponding MMP instance. Because the logical model does not have the high degree of redundant data found in MMP, the logical model may be an appropriate implementation model for MMP.

Chapter 7

Related Work

In the past decade, significant contributions have been made to the literature on provenance models. Three major areas of provenance work are evident: provenance models for coarse-grained data (typically whole datasets) in scientific workflows; provenance models for fine-grained data (typically tuples) in databases; and provenance models for entire files in filesystems. MMP addresses only the second of these. We focus on provenance for fine-grained data, and we specialize MMP to relational data for this work. While many models in the literature address only provenance of tuples in the relational setting, we address provenance at all granularities: relations, attributes, tuples, and attribute values. Because of this focus, we discuss related work in the literature only for fine-grained data provenance in database settings.

As an aid in contrasting the provenance models discussed in this chapter, we define the following distinguishing attributes, or dimensions, for provenance models. *History* is the first of these. We describe provenance models as having *ancestry-only* history if it only documents the data items contributing to a derived item. This description encompasses both the *Why-provenance* and *Where-provenance* defined by Buneman [8], also called the *original source* and *positive contributing source* by Glavic and Dittrich [16]. Because the difference between Why-provenance (original source provenance) and Where-provenance (positive contributing source provenance) are not intuitive, we offer the following example to distinguish them. Given

two relations r_1 with attributes a and b and r_2 with attributes b and c, and a relational algebra query

$$\pi_a(r_1 \bowtie_{r1.b=r2.b} r_2),$$

suppose that the query produces a result tuple t_r. The Where-provenance of t_r consists of all tuples t_1 from r_1 and t_2 from r_2 such that both $(t_r.a = t_1.a)$ and $(t_1.b = t_2.b)$. The Why-provenance of t_r consists of all tuples t_1 from r_1 where $t_r.a = t_1.a$. We describe such models as documenting ancestry-only history because both Why-provenance and Where-provenance document only the sources from which data was derived, and are silent about how data was derived, who derived it, or when it was derived. In addition to ancestry-only, we define two other classifications of the history dimension. We say that a provenance model documents *abstact history* if it includes ancestry-only history as well as a representation of how the ancestors combined to form the derived item, but that representation is not sufficient to fully reproduce the result given the inputs. For example, in the query above, suppose that t_r resulted from tuple t_{1x} joining with tuple t_{2y}, as well as from tuple t_{1w} joining with tuple t_{2z}. An abstract history provenance model might describe this provenance as

$$(t_{1x} \bullet t_{2y}) + (t_{1w} \bullet t_{2z}).$$

In contrast, we say a provenance model documents *full history* if it provides enough information to fully reproduce t_r given r_1 and r_2. For example, such a model might document the entire query text, and perhaps the time at which the query was issued, and the user who issued it.

Another dimension by which we characterize provenance models is whether they compute and record provenance at the time an operation derives a result, or whether provenance is derived later, when a user wishes to inspect it. We call the former *eager* provenance and the latter *lazy* provenance.

We also classify provenance models by whether provenance is recorded as an annotation to data, or whether provenance has an independent existence in the model.

We call the former *provenance-as-attribute* and the latter *provenance-as-entity*, inspired by entity-relationship (ER) model. This distinction may lead to substantial differences in model functionality. For example, a provenance-as-attribute model may allow explicit manipulation of provenance information by the DML or query language just as other attributes of data may be manipulated, whereas a provenance-as-entity model may define behaviors specific to provenance, and may possibly prevent direct alteration of recorded provenance.

Finally, we classify provenance models by whether they record provenance for only some of the granularities of data supported by the accompanying data model (*some-granularity*) or for all granularities supported by the data model (*all-granularity*).

In Section 7.1, we consider the only prevalent conceptual model of provenance in the literature. In Section 7.2, we consider logical models for data and provenance that have been implemented, at least at the level of a functional prototype. In Section 7.3, we examine work by Todd Green that addresses the relative expressive power of these logical provenance models, a result we presented and used in Chapter 4. Because we studied performance trade-offs in Chapter 5, in Section 7.4 we examine literature on performance of provenance models.

7.1 The Open Provenance Model

The Open Provenance Model [29] defines a conceptual model for provenance that applies both to coarse-grained data (e.g., datasets in scientific workflows) and fine-grained data. Though OPM defines provenance semantics, it is not intended to be instantiated directly. Instead, it serves as a technology-agnostic standard for designing provenance models for entities, whether digital or not, so that their provenance can be exchanged between systems that share OPM as a standard. OPM does not define provenance representations or syntax. Also, OPM does not define a language for

manipulating or interrogating provenance. Instead, OPM focuses on defining how provenance relationships are described.

OPM defines three kinds of entities: artifacts, processes which are performed on or caused by artifacts, and agents, which perform these processes. Provenance in OPM is represented as a directed, acyclic graph of dependencies between these three artifact types. Provenance graphs in OPM may include compositions of more than one such dependency. This feature allows OPM to represent multi-generation provenance and other complex dependency chains between entity types.

OPM allows for full-history as well as abstract-history and ancestry-only implementations. OPM is not specified as eager or lazy, and allows for both some-granularity and all-granularity implementations. The specification of OPM is agnostic about whether OPM is a provenance-as-attribute or a provenance-as-entity model. Like OPM, MMP is a conceptual model of provenance, and is technology-agnostic and representation-agnostic. However, MMP does provide a specific polynomial representation in order to be comparable to other systems with specific representations. MMP adopts the OPM notion of three kinds of entities. Components in MMP are artifacts; operations in the MMP language are processes; and users of MMP instances are agents. Provenance in MMP is modeled as a graph, as in OPM, but the MMP graph model is a subset of the OPM model. Each edge in an OPM provenance graph represents one of five types of causal relationships: a process *used* an artifact; an artifact *was generated by* a process; a process *was triggered by* a process; an artifact was *derived from* an artifact; or a process was *controlled by* an agent. Each provenance link in an MMP instance has the structure in OPM syntax shown in Figure 7.1. In the figure, the "used" artifact is the input component to the operation corresponding to the "process" and represented by a provenance link. The "was-generated-by" artifact is the result component of the operation. The identity of the process and agent are captured in MMP in the label of the face to which the result component belongs.
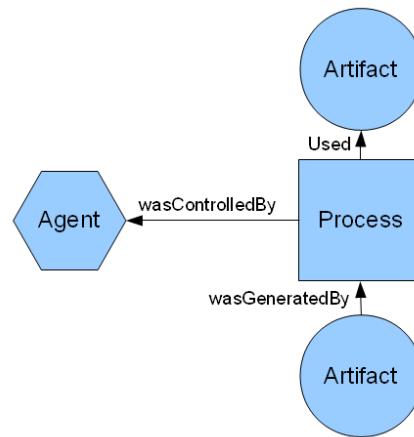
Figure 7.1: MMP Provenance Links Represented in OPM Syntax

MMP does not employ other OPM edge types or other configurations of OPM edges. For example, MMP does not record relationships of processes triggering other processes, because operators are modeled as being directly applied by users or other agents, rather than by hierarchies of processes.

MMP differs from OPM in that it models both data and its provenance, rather than only provenance. In addition, MMP differs from OPM in that it defines a language for manipulating and querying data that induces provenance for resulting data. In addition, MMP defines a language for interrogating provenance.

7.2   Provenance Models in the Literature

Current literature includes several provenance models that define specific representations of provenance and specific methods of storing provenance. We call these logical provenance models, and compare them to MMP^L defined in Chapter 6. Here we briefly describe and classify each relevant logical provenance model from the literature.

7.2.1 Lineage Tracing for General Data Warehouse Transformations

In one of the earliest articles on provenance of relational data, Cui and Widom [12] address the problem of tracing data items in a data warehouse back to the source items from which they were derived. Using a set of data manipulations typically seen in a data warehousing environment, they formally define the lineage discovery problem and present algorithms for tracing lineage in such environments. The resulting provenance model, called the *Lineage* model, is lazy, computing provenance by use of inverse queries run when users wish to trace provenance. This approach differs from most relational data provenance models in the literature, which are eager models. The model is an ancestry-only model, recording only the set of tuples that cause a result tuple to appear. Like many other models in the literature, the Lineage model is a provenance-as-attribute model, recording provenance as an extra attribute for each tuple. The Lineage model does nothing to prevent direct manipulation of these annotations. However, this model escapes concerns of direct manipulation because the annotations are derived on demand. Because only provenance of tuples is recorded, we classify this model as a some-granularity model.

The Lineage model differs from MMP in that it records only tuple-level provenance, while MMP is an all-granularity model. The Lineage model's provenance-as-attribute approach also differs from MMP's provenance-as-entity approach. Lineage also differs from MMP in that it computes provenance only for relational algebra operators (i.e., queries), while MMP additionally addresses DML and DDL operators. Recall that in Chapter 4 we contrasted MMP and the Lineage model with regard to how well they fill gaps in the provenance literature and meet the requirements of our motivating use cases.

7.2.2 Annotation Management Systems

Bhagwat *et al.* [4] present a general-purpose annotation-management system for relational databases. The system they describe acts as a provenance model when annotations consist of the identities of ancestor data. Unlike provenance annotations in the Lineage model and others we describe here, Bhagwat *et al.* describe annotations attached to each attribute value, rather than entire tuples. Their model is thus a some-granularity model and a provenance-as-attribute model. Because the system is intended for general annotations, it is not specific to ancestry-only versus abstract- or full- history: an implementation of the model they define could use any of these approaches. Because the described model propagates annotations as queries are computed, we classify this model as an eager model. Because Bhagwat's system is limited to propagating annotations through queries, it addresses only provenance due to query operations.

In their system, the authors define three distinct annotation propagation schemes, useful for different purposes. The default scheme propagates annotations according to where result data is copied from in query input data. Another scheme, called default-all, propagates annotations in a way consistent with all equivalent queries of the stated query. The third choice is a propagation scheme that allows user definition of how annotations propagate. The choice of propagation scheme to be used is phrased in a variant of SQL. The authors show how queries in this SQL variant are re-written into one or more SQL queries that correctly propagate annotation attribute values.

Unlike Bhagwat's system when used as a provenance model, MMP is a provenance-as-entity model. That is, MMP provenance is recorded and maintained in a separate model orthogonal to data. Because Bhagwat's model is not specific to provenance the two are not comparable with respect to the type of history they record. However, the two are comparable in that both are eager models, and in that MMP is an

all-granularity model while Bhagwat's model is a some-granularity model.

7.2.3 CPDB

Buneman's paper on the Copy-Paste Database (CPDB) [5] defined the data cura-
tion setting. Curated databases in disciplines such as bioinformatics are typically
maintained by significant manual correction, integration, and manipulation. Bune-
man noted that as a result, provenance information for such data is a key factor in
assessing data quality. The CPDB provenance model was motivated by the prove-
nance needs of users operating in such settings. Unlike other models in the litera-
ture, CPDB is a full-history model, although it only addresses DML operations, not
queries or DDL operations. Like most other models considered here, CPDB is an
eager model and a provenance-as-attribute model. Even though provenance is stored
in an auxiliary relation, it is subject to treatment as a data attribute. CPDB is unique
among the models we consider in that, such as MMP, it is an all-granularity model.
As noted in Chapter 1, even though DML operators such as insertion are supported,
CPDB does not address multiple insertion of identical data (nor tracking of multiple
histories) as MMP does.

In work subsequent to CPDB, Buneman *et al.* developed a framework based on
CPDB for managing provenance due to queries as well as data manipulations in a
single model [7]. They use this extended model to compare two ways in which input
data items are rearranged to create output data items: implicit provenance, where a
query or update only specifies the output and provenance is provided implicitly by
a default semantics, as done in Trio [1], which we discuss in the next section; and
explicit provenance, where an operation defines both output data and the description
of the provenance of each output item. The latter approach is similar to the user-
defined provenance approach defined by Bhagwat [4].

7.2.4 Trio

Trio, developed at Stanford University, supports both data uncertainty and provenance [1]. The key insight in Trio is that many application areas require both notions of uncertainty in data and lineage of data, yet previous literature considered only one or the other. For example, lineage enables correlating uncertainty in query results with uncertainty in query input data. The goals of the Trio project are to combine these two notions into a simple and usable data model, provide a query language that extends SQL in order to interrogate data, its uncertainty, and its lineage together, and provide a working system to demonstrate these ideas.

We restrict our consideration of Trio to data operations without uncertainty. Like Lineage and CPDB, Trio supports relational data, and records provenance in the form of annotations to tuples at the time queries are executed. We thus classify the Trio provenance model as an eager, some-granularity, provenance-as-attribute model. Like Lineage, this provenance includes where data came from, but not what manipulations were done, nor who performed them. We thus classify the Trio provenance model as an ancestry-only model. Trio's language supports queries as well as data manipulation, but does not support multiple insertions as MMP does. This limitation prevents Trio from meeting some needs of our motivating use cases as described in Chapter 1. Trio is the only current model besides MMP that retains deleted data. It is also the only current model that provides a provenance-specific built-in function, *Lineage()*, to help users in writing provenance-related queries. Recall that in Chapter 2, we compare Trio and MMP with respect to gaps in the literature identified in Chapter 1. Also recall that, in Chapter 4, we compared Trio and MMP with respect to their expressive power with regard to provenance queries.

7.2.5 Panda

The Panda (Provenance ANd DAta) project at Stanford was first reported in the literature in 2010. The goal of Panda is to address some of the same limitations in existing provenance models addressed by MMP. At present, the only information available on Panda is a short paper [24] that describes project plans. This paper reports that Panda will include a model that fully integrates data-based and process-based provenance, making it a full-history model. Panda is also set to include built-in operators for exploiting provenance after it has been captured as part of an ad-hoc query language over provenance together with data, perhaps similar to the MMP provenance predicate language. Like MMP, Panda intends to support provenance for a full range of data granularities, making it an all-granularity model. Panda also promises exploration of lazy vs. eager approaches. At this time, it is unclear whether Panda will be a provenance-as-attribute model (such as its predecessor, Trio) or provenance-as-entity model (such as MMP).

7.2.6 Orchestra

Orchestra [21, 19] is a system designed to allow sharing of data among peer databases. Each Orchestra peer is assumed to have a locally controlled and edited database instance. Orchestra also assumes that peer databases are related by schema mappings that allow one peer to map desired data from another peer into its own schema. Orchestra uses a publish-subscribe model whereby each peer publishes updates to its data at will, and each peer receives those updates at will, using the schema mappings to re-map the received data into a locally usable schema. This mapping includes the application of trust filters that express which received data the receiving peer judges trustworthy. Orchestra bases this trust assessment on provenance information, so that local users can decide which data to trust and which not to trust based on the origins of data. The Orchestra system uses an eager provenance model

to record provenance. This provenance is recorded alongside data, and treated as a property of individual tuples, making Orchestra a provenance-as-attribute model that uses a some-granularity approach.

The provenance representation used in Orchestra expresses both ancestor data and a loose (algebraic) description of how data was derived, so we classify it as an abstract-history model. This representation uses semirings of polynomials [20], similar to MMP provenance polynomials. Recall that Figure 1.1 shows an example of a simple query and how the provenance of its result tuples is represented in several provenance models, including the Orchestra model. In an Orchestra provenance polynomial, the multiplication operator indicates that the combined presense of input tuples give rise to an output, and the addition operator indicates that each of its input tuples gives rise independently to a result. In Orchestra, these polynomials are restricted so that there is no concept of derivations that include multiple operations applied over time. Because of this restriction, multiple insertions are not part of the Orchestra model, and there is no notion of multi-generation provenance in Orchestra.

7.3 Comparing Expressiveness of Popular Provenance Models

Green studied containment and equivalence of (unions of) conjunctive queries on relations annotated with elements of commutative semirings, such as the Orchestra provenance model, the Lineage model, the Why-provenance model, and the Trio model [18]. Green shows that containment of conjunctive queries and unions of conjunctive queries is decidable for these models. He also characterizes the complexity of proving containment in each case. In particular, Green showed that these models are related by surjective semiring homomorphisms. He uses these relationships to characterize their relative expressive power. Recall that we used this result in Chapter 4 in order to show the relative expressive power of the provenance polynomial representations included in the MMP model.

7.4 Performance of Provenance Models

Little has been published concerning performance of provenance-related queries. The only significant work in the area is that of Karvounarakis *et al.* [25]. In this recent work, the authors design a query language for use in a collaborative data sharing system (i.e., Orchestra), propose a set of test queries that represent expected use cases, and examine query performance. They show results for how query performance varies with the number of peer databases that provide input data, the number of rules in the schema mappings, and the number of tuples in the local database derived by the mappings. Our work in Chapter 5 is similar to this work in that we study query performance for a representative set of provenance queries. However, our focus is the study of performance as a function of the number of generations in data's provenance, instead of the number of rules in mappings and number of tuples in the database.

7.5 Chapter Summary

Most provenance models in the literature specify how provenance is stored and how it is internally represented. OPM and MMP are exceptions, specifying only what information is recorded and what its semantics are. MMP also specifies limits on how provenance may be manipulated. These differences lead us to categorize OPM and MMP as conceptual provenance models, and the others as logical provenance models.

Because OPM is a very abstract model, characterizing it by the four descriptors we define in this chapter offers limited information. However, the other models (Lineage, Bhagwat's model, Why-provenance, Orchestra, Trio, and our own MMP) can be characterized as follows:

- MMP and Why-provenance are full-history models, Orchestra and Trio are

abstract-history models, and Lineage is an ancestry-only model. Bhagwat's annotation system does not discuss provenance directly, so it could implement any of these.

- Lineage is a lazy model, while the others discussed here are eager.

- MMP is a provenance-as-entity model; Orchestra, Trio, Lineage, and Bhagwat's model are provenance-as-attribute models.

- MMP and Why-provenance are all-granularity models, while the others are some-granularity models.

Chapter 8

Conclusion

We motivated this dissertation by noting the importance of provenance in assessing the origins and quality of data used in a variety of domains, including budget forecasting, project management, scientific workflows, battlefield information integration, intelligence operations, and others. We focused our research by identifying five opportunities to make provenance easy to use in domains such as these: 1) current models do not model provenance resulting from a mix of DDL, DML, and query operations; 2) in current models, users must parse and interpret each provenance representation manually; 3) in current models, users must assemble multi-generation provenance manually before querying or browsing it; 4) query languages used in current models are designed for relational data, and so are not well-suited to phrase queries over provenance; and 5) current models do not distinguish provenance from data in order to provide suitable management for provenance. We also included a goal of allowing for multiple insertion of identical data.

These opportunities motivated definition of and research on a conceptual model of provenance and data. Because we wanted to be able to compare and contrast our model against others in the literature for purposes of evaluation, we chose to specialize the data portion of our model to relational data, while taking the goal of making the provenance portion of the model distinct so that it could be retargeted to other data models in the future.

We thus defined a conceptual model for data and provenance where provenance is orthogonal to data: both provenance and data have first-class status, where in most other models in the literature provenance is treated as an attribute of data; the provenance model is not defined exclusively to apply to relational data, but can instead apply to (we conjecture) many data models; data and provenance can be maintained separately, and be manipulated in distinct ways by operators defined on the model; and data is seen to exist at instants in time while provenance interconnects those instants. In order to make it easy for users to interrogate the provenance portion of our model together with the data portion, we defined a language for manipulating and querying data based on both data and provenance represented in the model. This language allows us to phrase an interesting class of queries typically unaddressed by other models in the literature: the class of provenance selection queries, which select data based on characteristics of its multi-generation provenance ("Show me data in this table that resulted from a manipulation performed last Tuesday"). Languages used in other models are often limited to answering queries about provenance, given data of interest ("Show me where this tuple came from"), and are often limited to answering queries about only immediate (single generation) provenance.

We formally defined MMP in this work. We evaluated MMP in terms of how well it addresses the opportunities in the literature mentioned above, how informative the provenance part of MMP is relative to other models, how expressive our provenance query language is with respect to a variety of provenance query classes, and how the syntactic complexity of queries in our language compares to those of others in the literature.

To evaluate the expressiveness of our language and evaluate the syntactic complexity of writing queries in it, we defined a performance benchmark suite that includes data, provenance, and queries representative of real-world use models. During our evaluations we found that the MMP provenance model is more informative than

others in the literature, notably the provenance polynomials model of Green [21] that has already been shown to be the most informative of prominent models in the literature. We also found that the MMP language can express a wider variety of provenance selection queries than other models, and has equal or better syntactic query complexity on average than these other models.

To evaluate query performance trade-offs for possible MMP implementations, we applied selected queries from our benchmark suite to several implementation prototypes built on different technologies. We found that provenance queries performed comparably when provenance information was stored in a graph database and in a relational database, as data scales to realistic volumes. However, we found that the data portion of queries we tested performed much better (by 2 to 6 orders of magnitude) when data was stored in a relational database than in a graph database. The performance studies also showed promising results for scalability in provenance queries. Performance of provenance queries scaled linearly with size of data over the entire range we tested.

Our conceptual model is relatively inefficient in terms of the amount of redundancy of information recorded. Data in MMP is replicated to model its existence at each point in time when the database is manipulated. To ensure that a practical implementation of MMP is possible, we showed that a logical model with minimal redundancy could be defined to faithfully support our conceptual model. Such a logical model can serve as the basis for implementation of MMP.

8.1 Discussion

Our work achieved several positive outcomes, discussed above. In addition, we were able to retain the relational data model without change, making MMP intuitive for users familiar with relational databases. Our formal definition of MMP added functionality to relational algebra, DML, and DDL operators for managing provenance.

However, our definition did not change the effect of these operations on data. The sole exception is that we modified deleted data to be retained, but not participate in future operations, so that provenance based on deleted data could be represented. We were also able to achieve our goal of keeping data and provenance distinct, yet queryable together in MMP. Orthogonality was surprisingly useful in allowing the definition of a clean and simple query language. That is, keeping data and provenance separate allowed us to inspect provenance using a set of selection and projection predicates while not needing to address provenance elsewhere in the MMP language. Our performance studies also revealed interesting results about orthogonality in MMP. The distinct, orthogonal portions of our model (data and provenance) performed differently on the relational and graph database technologies we tested, supporting the idea that they should be implemented in independent (orthogonal) ways.

The results of our performance studies showed that neither a purely relational approach nor a purely graph approach to MMP implementation would yield adequate performance. It seems unlikely that a technology other than a relational database will perform better for relational data queries. It also seems unlikely that a technology other than a graph substrate will perform better for queries that traverse provenance graphs. Taken together, these ideas suggest that the next step in exploring implementation options for MMP should be to consider a hybrid approach such as the one used as a logical model in Chapter 6. Such an approach might adapt an existing temporal-relational database to add provenance link structures between components. To speed up path matching for provenance queries in such an implementation, the database would also need to be equipped with some kind of path-indexing capabilities. We believe that construction of such a platform would be a good area for future work.

8.2 Future Work

In the course of this work, we have encountered several avenues for further investigation. Extending MMP to other data models would be a good first step toward determining whether our general provenance model can be re-used. Another area of exploration is to determine how to connect distinct MMP instances so that provenance information may be extended across instance boundaries. Combining results from these two areas of research may tell us to what extent we can extend provenance across instances that support distinct data models. One specific area of study would be to examine the problem of assessing data quality in workflows for development of targeted therapies for cancer. Such workflows are representative of many use cases in the medical domain. These workflows use heterogeneous data from a variety of sources. Relational databases are used for storing reference information about gene expression. Text-mined data from electronic health records are analyzed for indications of gene expressions that may cause cancers. DNA sequencing and gene profiling datasets are input to workflows that use these other forms of data to yield prioritizations of likely genetic causative factors. The workflows used are composed of processing modules that evolve over time. All of these data (and processing) use distinct data models, yet have provenance that must interoperate in order to assess quality of workflow results.

Usability of models such as MMP remains an open area of investigation. We conjecture in this work that users may want to browse provenance graphs for selected data. We also conjecture that users are willing to learn additional query language syntax in order to express provenance-related queries. However, we have only anecdotal evidence to support these conjectures, and very little has been published in the literature about the usability of provenance models.

As we conclude above, the next step in implementation choices for MMP should be research into a hybrid platform, based on a temporal-relational database, with

provenance links as an added form of relationships between components. It is clear from our results in Chapter 5 that acceleration of finding provenance paths that match our provenance predicates will be key to achieving practical provenance-related query execution time. We believe that path indexes should be explored in such a platform.

Overall, this work contributes a conceptual model for data and provenance. This model addresses several opportunities in the literature of provenance models and meets some of the needs of users in a variety of use models. More importantly, this work contributes not only MMP, but also methods for evaluating models such as MMP, and a variety of ideas for further exploration.

References

[1] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: a system for data, uncertainty, and lineage. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1151–1154. VLDB Endowment, 2006. [cited at p. 2, 12, 193, 194]

[2] A. Bairoch, B. Boeckmann, S. Ferro, and E. Gasteiger. Swiss-prot: juggling between evolution and stability. *Briefings in Bioinformatics*, (1):28–39, March 2004. [cited at p. 1]

[3] Omar Benjelloun, Anish Das Sarma, Alon Halevy, Martin Theobald, and Jennifer Widom. Uldbs: Databases with uncertainty and lineage. *The VLDB Journal*, 17(2):243–264, 2008. [cited at p. 90, 109, 114]

[4] Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya. An annotation management system for relational databases. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 900–911. VLDB Endowment, 2004. [cited at p. 2, 90, 192, 193]

[5] Peter Buneman, Adriane Chapman, and James Cheney. Provenance management in curated databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 539–550, New York, NY, USA, 2006. ACM. [cited at p. 2, 11, 114, 193]

[6] Peter Buneman, Adriane Chapman, James Cheney, and Stijn Vansummeren. A provenance model for manually curated data. In *IPAW*, pages 162–170, 2006. [cited at p. 90, 107]

[7] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.*, 33(4):1–47, 2008. [cited at p. 12, 193]

[8] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 316–330, London, UK, 2001. Springer-Verlag. [cited at p. 2, 20, 109, 186]

[9] David N. Card and Robert L. Glass. *Measuring software design quality*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990. [cited at p. 126]

[10] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. Apex: an adaptive path index for xml data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 121–132, New York, NY, USA, 2002. ACM. [cited at p. 154]

[11] Gao Cong, Wenfei Fan, and Floris Geerts. Annotation propagation revisited for key preserving views. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, CIKM '06, pages 632–641, New York, NY, USA, 2006. ACM. [cited at p. 69]

[12] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000. [cited at p. 2, 11, 90, 109, 154, 191]

[13] Michael Deininger, Elisabeth Buchdunger, and Brian J. Druker. The development of imatinib as a therapeutic agent for chronic myeloid leukemia. *Blood*, 105(7):2640–2653, 2005. [cited at p. 3]

[14] Brian J. Druker, Moshe Talpaz, Debra J. Resta, Bin Peng, Elisabeth Buchdunger, John M. Ford, Nicholas B. Lydon, Hagop Kantarjian, Renaud Capdeville, Sayuri Ohno-Jones, and Charles L. Sawyers. Efficacy and safety of a specific inhibitor of the bcr-abl tyrosine kinase in chronic myeloid leukemia. *New England Journal of Medicine*, (14):1031–1037, 2001. [cited at p. 3]

[15] Orri Erling and Ivan Mikhailov. Integrating open sources and relational data with sparql. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 838–842, Berlin, Heidelberg, 2008. Springer-Verlag. [cited at p. 155]

[16] B. Glavic and K. Dittrich. Data provenance: A categorization of existing approaches. In *Proceedings of Die 12. BTW-Tagung der Gesellschaft fr Informatik*, BTW'07, pages 227–241, 2007. [cited at p. 186]

[17] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. [cited at p. 154]

[18] Todd J. Green. Containment of conjunctive queries on annotated relations. In *ICDT '09: Proceedings of the 12th International Conference on Database Theory*, pages 296–309, New York, NY, USA, 2009. ACM. [cited at p. 109, 121, 196]

[19] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Update exchange with mappings and provenance. In *VLDB '07: Proceedings of*

the 33rd international conference on Very large data bases, pages 675–686. VLDB Endowment, 2007. [cited at p. 90, 195]

[20] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, New York, NY, USA, 2007. ACM. [cited at p. 13, 92, 100, 114, 196]

[21] Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Olivier Biton, Zachary G. Ives, and Val Tannen. Orchestra: facilitating collaborative data sharing. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1131–1133, New York, NY, USA, 2007. ACM. [cited at p. 2, 12, 98, 102, 109, 121, 195, 201]

[22] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977. [cited at p. 126]

[23] Huahai He and Ambuj K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 405–418, New York, NY, USA, 2008. ACM. [cited at p. 100, 155]

[24] Robert Ikeda and Jennifer Widom. Panda: a system for provenance and data. In *Proceedings of the 2nd conference on Theory and practice of provenance*, TAPP'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association. [cited at p. 2, 195]

[25] Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. Querying data provenance. In *Proceedings of the 2010 international conference on Manage-*

ment of data, SIGMOD '10, pages 951–962, New York, NY, USA, 2010. ACM. [cited at p. 130, 197]

[26] A.V. Levitin. How to measure size, and how not to. In *Proceedings of the Tenth COMPSAC Conference*, Washington DC, USA, 1986. IEEE Computer Society Press. [cited at p. 126]

[27] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. [cited at p. 126]

[28] M. McCandless, E. Hatcher, and O. Gospodnetic. *Lucene in Action*. Manning Publications, 2010. [cited at p. 133]

[29] Luc Moreau, Juliana Freire, Joe Futrelle, Robert McGrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. In Juliana Freire, David Koop, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 5272 of *Lecture Notes in Computer Science*, pages 323–326. Springer Berlin / Heidelberg, 2008. [cited at p. 188]

[30] S. S. Sahoo, R. S. Barga, J. Goldstein, and A. Sheth. Provenance algebra and materialized view-based provenance management. *Microsoft Research Technical Report (MSR-TR-2008-170)*, 2008. [cited at p. 112]