Butterfly: A Model of Provenance

by

Yaobin Tang

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

March 2009

Professor Murali Mani, Thesis Advisor

Professor Elke A. Rundensteiner, Reader

Professor Michael Gennert, Head of Department

Abstract

Semantically rich metadata is foreseen to be pervasive in tomorrow's cyber world. People are more willing to store metadata in the hope that such extra information will enable a wide range of novel business intelligent applications. Provenance is metadata which describes the derivation history of data. It is considered to have great potential for helping the reasoning, analyzing, validating, monitoring, integrating and reusing of data.

Although there are a few application-specific systems equipped with some degree of provenance tracking functionality, few formal models of provenance are present. A general purpose, formal model of provenance is desirable not only to widely promote the storage and inventive usage of provenance, but also to prepare for the emergence of so called provenance management system.

In this thesis, I propose Butterfly, a general purpose provenance model, which offers the capability to model, store, and query provenance. It consists of a semantic model for describing provenance, and an extensible algebraic query model for querying provenance. An initial implementation of the provenance model is also briefly discussed.

Evolution is fascinating to watch. To me it is the most interesting when one can observe the evolution of a single man.

Shana Alexander

Hominid and human evolution took place over millions and not billions of years, but with the emergence of language there was a further acceleration of time and the rate of change.

William I. Thompson

The more chaos there is, the more science holds on to abstract systems of control, and the more chaos is engendered.

William I. Thompson

The task of art today is to bring chaos into order.

Theodor Adorno

Contents

Acknowledgments

I would like to express my deepest gratitude to my advisor Professor Murali Mani for his patience and efforts in supervising my study at Worcester Polytechnic Institute. I would like to thank my thesis reader Professor Elke A. Rundensteiner for the time she spent in reading my thesis and her valuable questions and comments.

Chapter 1

Introduction

A very small cause which escapes our notice determines a considerable effect that we cannot fail to see, and then we say that the effect is due to chance.

Jules H. Poincare

1.1 Motivation

With today's abundant computer storage and powerful processing capability, people become more and more aggressive in collecting extra data: data intentionally generated to assist the understanding of other data or processes. Simple form of model and query of such data cannot satisfy non-expert users' growing appetite for intelligent support in applications. For example,

- an online catalog vendor wants to track the interaction of customers with the UI to discover the sequential pattern of operations which ends in purchasing a product; find the most visited (used) web page (interface) to improve user experience; query the connection between two visited web pages to better understand user behavior and enhance cross-selling.
- a scientist wants to log detailed running steps and intermediate results of an experiment saving the opportunity for future inspection or reproduction of the result;

providing poof to peer scientists about the authenticity of the experiment; contributing to the pool of experimental recipes for reuse.

- a food manufacturer wants to record the production and distribution process of a product, so whenever the product is found flawed, it is possible to trace back to the origin of the problem; map the affected vendors, shops, and regions; estimate the ensuing loss and compensation; submit report to supervisory authority for conformity check.

There are some common patterns in the aforementioned scenarios:

- People are interested in collecting transient ancillary information, which is usually not captured or simply discarded. The form of such information is diverse varying with domain.
- Ad hoc queries are asked about complex relationship among data. Answers to these queries can benefit people with insight into the domain.

Having realized the usefulness of ancillary data and having envisioned its popularity, we have launched the `MetaWare` project, which aims at providing a general solution to the management of metadata.

1.2 Butterfly

We are particularly interested in provenance, a special kind of metadata that assists in understanding how things are *related* to each other in their derivation history. Inquiry of provenance is pervasive in everyday life, and a lot of applications (like the previous examples) profit from the ability to know the provenance of an entity.

However, two obstacles are in the way of managing provenance:

- Before we can store provenance, we need a way to describe them.

- Once provenance is stored, we need a query language to extract information from them.

Our solution to the first problem is a *semantic model* of provenance. Our solution to the second problem is an algebraic *query model*. The combination of two is a general purpose provenance model, which we name `Butterfly` (named after the butterfly effect). We are also working on an initial implementation of the provenance model.

Assumptions We believe a provenance model should be agnostic of application domains. We argue that a provenance management system itself should not be held responsible for capturing provenance. Instead, users should decide on what and how they capture provenance according to their particular needs and views.

Usage Scenario

We imagine a typical scenario of applying `Butterfly` would look like this: A programmer is assigned to build a provenance-aware application for a hospital (e.g., Electronic Health Record or simply paperless office application). He knows there is a handy middleware called `Butterfly` which he can easily integrate into his application for processing provenance. What he needs to do is embed some provenance processing codes into appropriate places of the business logic. He can decide what interesting events to record and how to describe them (e.g., granularity) using the semantic model. The provenance processing codes talk to `Butterfly` in the definition language. After the host application has run several months, a patient revisits the hospital. A doctor opens an editor and composes some queries to pull out the sequence of treatment the patient has received in the past. Later, the patient files a complaint. A supervisor queries the provenance base to investigate if there is any violation of regulation in the sequence of treatment.

Contributions

Although there are many systems that utilize the idea of keeping track of data provenance, most of their provenance tracking components are highly coupled to their application logic (less reusable) and their provided queries are simple. Standardization of a general purpose provenance model has recently aroused attention from the provenance research community. For example, [21] is most relevant to our work in terms of its purpose. It independently proposes the Open Provenance Model (OPM). The fundamental difference between OPM and our work is mainly trifold: *a*) OPM classifies causal relationship into five exact relationships. We use a loose, uniform relationship and leave the exact interpretation of relationship to varied domains. *b*) OPM queries are based on logical inference from the five causal relationships. We use an algebraic approach instead. *c*) Our goal is to provide a *general purpose, formal, functional* model of provenance. In order to do that, we specify the structural representation of each provenance concept.
More in-depth discussion of contribution can be found in chapter 5.

1.3 Related Work of Data Provenance

Data provenance, also called data lineage, describes how a piece of data was obtained from its predecessor.

[23] develops taxonomy of provenance techniques to compare nine key provenance systems based on why they record provenance, what they describe, how they represent and store provenance, and the methods to disseminate it.

[7] proposes a meta-model for architectures of lineage retrieval systems. The authors suggest that the meta-model should have three components: workflow model, metadata model, and lineage retrieval component, and lineage retrieval component should depend on workflow model and metadata model. Four lineage retrieval prototype systems are analyzed under the proposed meta-model framework.

Many projects in data provenance rose from the domain of scientific computing and experiments. In such environment, service-oriented architecture is widely used. A service-oriented architecture is a network of services in which data consumers and data producers are connected. The history of data processing can be documented by logging the executions of each service node and identifying its input and output data sets, or by creating metadata that describe the produced data sets or invoked executions, then metadata can be chained together to answer provenance queries. According to the terminology of [23], the former approach is process-oriented, and the latter one is data-oriented.

The Chimera project [13] proposes explicit representation of computational procedures and their invocations in a virtual data catalog (VDC), for discovery of available computational methods, and on demand data generation in the Data Grid environment. With the provided language, a user can explicitly store declarations of transformations or their invocations in the VDC. The VDC serves as a resource recipe, and simple queries over the VDC can be written to find out computational procedures and their invocations.

In the myGrid project [24], semantically rich execution logs [27] are automatically produced during workflow invocations to record services called, input and output data, etc. Data and services are annotated using ontologies to allow provenance inference.

CMCS [12] develops an metadata infrastructure which allows the extraction, translation, and manipulation of metadata. Lineage relationship between data entries can be visualized. In the CMCS metadata schema, the Dublin Core Element Set [1] is used to record lineage relationship. When files are loaded into the data repository, metadata generators are executed to create the respective metadata.

ESSW [14] is a script based data management infrastructure for earth science products. It uses Perl scripts to automatically collect metadata values and lineage information in the experiments. XML documents are created using collected metadata values and predefined metadata templates. These XML documents are stored in a relational database. The relationship between metadata IDs is stored in a table for tracking lineage. [6] is similar to [14], where a lineage metadata model is proposed, and metadata is associated to every constituent of a workflow. Two approaches for composing lineage metadata are discussed, one using XML metadata documents, the other using Resource Description Framework (RDF).

Since a relational database query can be thought of as a tree of transforming operators, provenance (lineage) is also a subject of database research [4,5,8,10]. However, in contrast to the scientific computing domain, provenance tracking is focused on locating the source data that is responsible for the production of the result data, rather than the process of how the result data is obtained. [20] names such type of provenance as input provenance. [9] defines two types of provenance: where-provenance for describing the original data from which the result data is copied from, and why-provenance for describing the data that affects the existence of the result data. [4] and [5] use logging to store provenance information during execution of simple queries. More particularly, [4] logs the causal relationship between output rows and input rows. [5] tags every piece of data (at attribute level) in the source tables with a unique identifier, and propagates the identifier along with the data it tags during the query processing. Provenance can be obtained by logging, or it can be computed on demand in some cases. For example, [10] uses inverse query to compute the origin of given data on demand. This approach is limited, because, generally speaking, “inverse query” does not exist [26].

There are other ad hoc systems that store and utilize provenance information for a variety of application specific needs [3, 17, 25]. For example, in a command line environment, when commands are executed, their parameters, output files are monitored and their dependency is intercepted and stored. [25] makes use of such provenance information for result caching. [3] implements the auditing facility in the S system, an interactive programming environment for data analysis, by logging to an audit file and creating audit data structure for query processing.

Most projects in data provenance are confined to their specific domains. [20] presents and analyzes use cases from e-science experiments of different disciplines in the hope of determining the technical requirements of a software architecture that supports recording, storing and using provenance data. Based on the result of [20], project PASOA [15] attempts to provide a general provenance architecture that satisfies the need of applications whose system architectures are of service oriented type.

Roadmap The rest of this thesis is structured as follows: We introduce the semantic model in chapter 2, the query model in chapter 3. We briefly discuss the implementation issues of our provenance model in chapter 4 and conclude with chapter 5.

Chapter 2

Semantic Model of Provenance

What distinguishes a mathematical model from, say, a poem, a song, a portrait or any other kind of 'model', is that the mathematical model is an image or picture of reality painted with logical symbols instead of with words, sounds or watercolors. These symbols are then strung together in accordance with a set of rules expressed in a special language, the language of mathematics.

John Casti

Definition 2.0.1: A *name space* is an infinite countable set, denoted as $\aleph$.
e.g., the set of all possible ASCII strings is a name space.

Definition 2.0.2: A *named value* is $(name, value\text{-}list)$. $name \in \aleph$. *value-list* is a list, and it can be empty.
e.g., (Buyer, (John Green)) is a named value. Note if *value-list* contains only one element, we can omit the parentheses of *value-list*. e.g., (Money, \$40k) is also a named value.

Definition 2.0.3: An *identifier* is a named value.
Two identifiers are equal if and only if their *names* are equal. For example, (R08, (Data File MedReport)) is an identifier, and is equal to (R08, ()).

The semantic model provides a tool for describing provenance. Particularly, it defines several essential concepts that we believe are closely related to the modeling of provenance. Our goal in this chapter is to define what are *provenance entity* and *provenance relationship.*

2.1 Application Environment

Definition 2.1: An *application environment* is

$$(NSpace, \mathcal{CLK}, \mathcal{ADDR}, \mathcal{DICT}).$$

NSpace is a name space. $\mathcal{CLK}$ is a set of system-recognizable time (e.g., in the format of MM-DD-YY). $\mathcal{ADDR}$ is a set of system-recognizable addresses (e.g., URI address). $\mathcal{DICT} \subseteq NSpace$, is a defined vocabulary.

Application environment mandates a minimum integrity constraint on the data, and provides some degree of interoperability. Note the following concepts are all defined with respect to an environment.

2.2 Annotation

Definition 2.2: An *annotation* is a set of named values. Each named value (*name*, *value-list*) is called an *entry*, and $name \in \mathcal{DICT}$.
e.g., {(total-income, $40k), (mortgage-limit, $100k), (mortgage-type, fixed-rate)} could be an annotation of a mortgage application.

Annotation provides an extensible way to describe something. Due to diversity of domains, we impose only little restriction upon annotation.

2.3 Static Entity Record

Defintion 2.3: A *static entity record* is

$$(\textit{entity-id}, \textit{entity-address}, \textit{entity-type}, \textit{annotation}, \textit{snapshot-time}, \textit{record-id}).$$

entity-id is an identifier. *entity-address* $\in \mathcal{ADDR}$. *entity-type* $\in \mathcal{DICT}$. *annotation* is an annotation as in Definition 2.2. *snapshot-time* $\in \mathcal{CLK}$. *record-id* $\in$ *NSpace*.

Static entity record captures some *aspects* of an entity at a particular *moment* (*snapshot-time*). e.g,

entity-id=(S01, (Aspect, $\cdots$))
entity-address=100 Inst. Rd
entity-type=Human
annotation={(Income, \$40k), $\cdots$ }
snapshot-time=11-01-07
record-id=R01

is a static entity record capturing some facts about a person at some moment. Note there is nontrivial distinction between *entity* and *static entity record*. The former one refers to the evolving physical existence, while the latter one refers to a virtual representation (as a record) of some aspects of that entity at a particular moment. With that being mentioned, it is clear that: *a*) Our model does not intend to manipulate (e.g., store) the actual entities (e.g., data files, pictures), but our model will manipulate their corresponding representations. *b*) Theoretically, static entity record is about some facts in the past, and should not be updated.

entity-id, *entity-address*, *entity-type* all refer to an *entity*. They are self-explanatory. One example of *entity-address* is Uniform Resource Identifier (URI). Currently, *entity-type* is simply drawn from $\mathcal{DICT}$. We intend to introduce hierarchy of entity types in the future for richer modeling. Past facts (states) of an entity can be amassed by grouping according to *entity-id* (recall the equality definition of identifier). Static entity record with the latest *snapshot-time* represents the closest approximation of the corresponding entity. *record-id* uniquely identifies the *record* itself not the *entity*.

2.4 Activity Record

Definition 2.4.1: An *activity type* is

$$(\textit{type-name}, \textit{incoming-list}, \textit{outgoing-list}, \textit{annotation}).$$

type-name is an identifier. *incoming-list* is an ordered list of named values (each named value in the *incoming-list* is called an *in-pipe*). *outgoing-list* is an ordered list of named values (each named value in the *outgoing-list* is called an *out-pipe*). *annotation* is an annotation as in Definition 2.2.

incoming-list declares the *roles* and *types* of the *contributing* entities in an activity. For example,

((Doctor, Human) (Patient, Human) (X-Ray, Machine))

is an *incoming-list*, with "Doctor", "Patient", "X-Ray" as roles and "Human", "Machine" as types. Types should be taken from $\mathcal{DICT}$.

Similarly, *outgoing-list* specifies the roles and types of the *consequential* entities in the activity. For example,

((Radiographic-Image, Image) (Diagnosis, Report))

is an *outgoing-list*. The types "Image" and "Report" should be taken from $\mathcal{DICT}$ as well.

Define a symbol $\underline{\text{ACT}}$, which denotes a dummy activity type. It symbolizes an insignificant or unknown activity.

Flexibility Because our model is meant to be simple but still flexible and capable, we avoid forced detail modeling of activity but reserve the potential for doing that. For example, based on *grouping* by *type-name*, we can support the concepts of hierarchical view and equivalent view of activities as follows: Recall *type-name* (as in Definition 2.4.1) is an identifier, with the form of (*name*, *value-list*). Let *value-list* = (*event*, *generality*, *explanation*). Intuitively, *event* clusters relevant activities of an event. Activities with a smaller *generality* value offer a more detailed view of the event. When *generality* values are the same, activities with a smaller *explanation* value are considered a more plausible explanation of the event. Figure 2.1 shows an example. It shows there is a more detailed view of "Building Caught Fire". The fire is more likely a result of lightning (*explanation*=1) than short-circuiting (*explanation*=2). There is also a more detailed view of "Firefighters Put Out Fire".

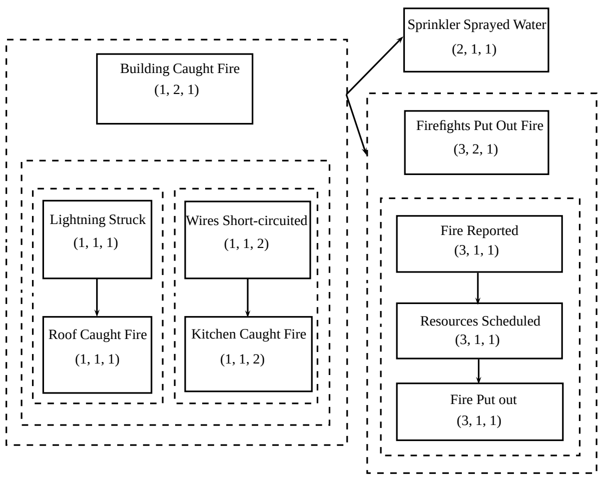


Figure 2.1: Flexibility—Grouping of Activities

Definition 2.4.2: An *activity record* is

$$(\textit{activity-id}, \textit{activity-type}, \textit{activity-span}, \textit{annotation}, \textit{record-id}).$$

activity-id is an identifier. *activity-type* is an activity type as in Definition 2.4.1. *activity-span*=(*start*, *end*), *start* and *end* $\in \mathcal{CLK}$. *annotation* is an annotation as in Definition 2.2. *record-id* $\in$ *NSpace*.

In contrast to static entity record, activity record is used to describe any *dynamic* element of an application system. It is an instance of an activity type, and a *representation* of an activity taking place in the physical world (e.g., an invocation of a computing function).

Each component of activity record is self-explanatory. We recommend to store in *annotation* additional application specific information about the running of an activity.

2.5 Provenance Entity And Relationship

A **provenance entity** is either a static entity record (Definition 2.3) or an activity record (Definition 2.4.2).

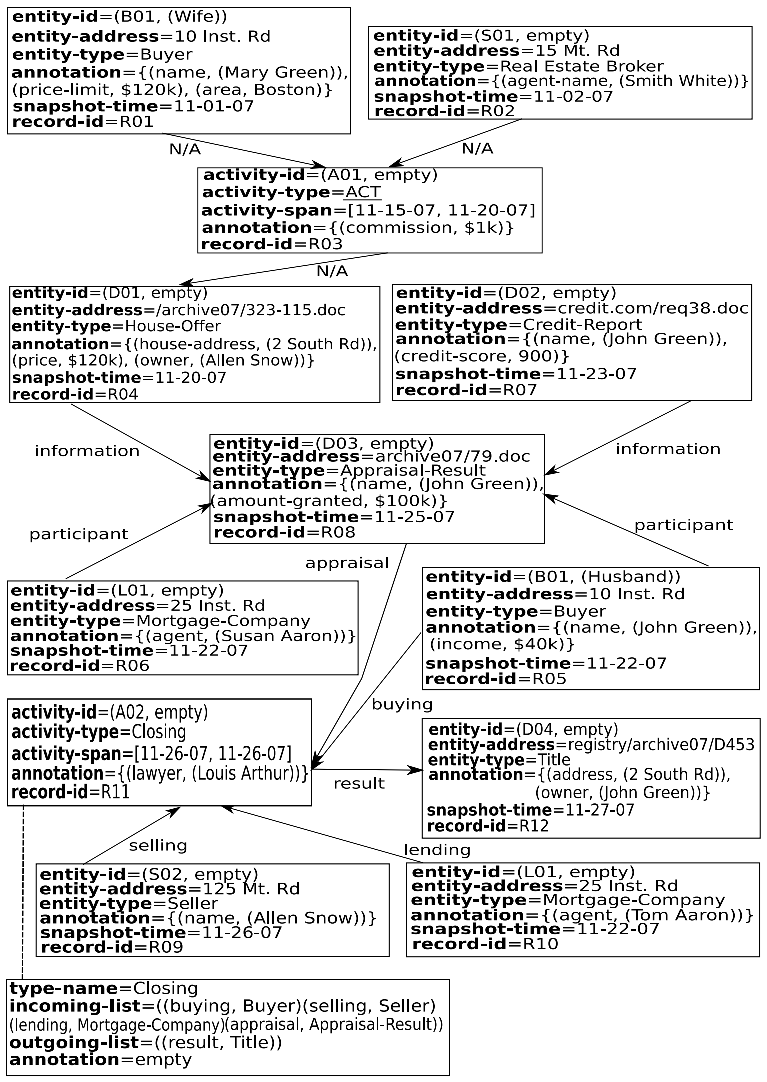
Definition 2.5.1: A **provenance relationship** is a relationship between two provenance entities:

$$(\textit{causal-entity}, \textit{consequential-entity}, \textit{role}, \textit{annotation}, \textit{relationship-id}).$$

causal-entity, *consequential-entity* are both provenance entities. $\textit{role} \in \mathcal{DICT}$. *annotation* is an annotation as in Definition 2.2. $\textit{relationship-id} \in \textit{NSpace}$.

Relationship between *causal-entity* and *consequential-entity* can be thought of as a parent-child relationship. *role* refines the relationship by supplementing the role that *causal-entity* played in the creation of *consequential-entity.* Compatibility check is required. e.g., when *causal-entity* is a static entity record and *consequential-entity* is an activity record, *role* and *causal-entity* must be meaningful to *consequential-entity* (e.g., in Figure 2.2, the "buying" role and "Buyer" type match an *in-pipe* of *activity-type* "Closing").

The semantic model provides a flexible way to describe provenance of both static and dynamic elements of an application system. Figure 2.2 shows a simplified example of house mortgage, where a prospective buyer got a house offer through a real estate broker, applied mortgage from a mortgage company, closed the transaction with the seller and got a new house title. There are two activities, one of which is of dummy type and the other is of "Closing" type. Provenance relationship is shown as directed link from *causal-entity* to *consequential-entity*, with *role* as the label of the link, *annotation* and *relationship-id* omitted for simplicity.


entity-id=(B01, (Wife))
entity-address=10 Inst. Rd
entity-type=Buyer
annotation={(name, (Mary Green)), (price-limit, $120k), (area, Boston)}
snapshot-time=11-01-07
record-id=R01

entity-id=(S01, empty)
entity-address=15 Mt. Rd
entity-type=Real Estate Broker
annotation={(agent-name, (Smith White))}
snapshot-time=11-02-07
record-id=R02

activity-id=(A01, empty)
activity-type=<u>ACT</u>
activity-span=[11-15-07, 11-20-07]
annotation={(commission, $1k)}
record-id=R03

entity-id=(D01, empty)
entity-address=/archive07/323-115.doc
entity-type=House-Offer
annotation={(house-address, (2 South Rd)), (price, $120k), (owner, (Allen Snow))}
snapshot-time=11-20-07
record-id=R04

entity-id=(D02, empty)
entity-address=credit.com/req38.doc
entity-type=Credit-Report
annotation={(name, (John Green)), (credit-score, 900)}
snapshot-time=11-23-07
record-id=R07

entity-id=(D03, empty)
entity-address=archive07/79.doc
entity-type=Appraisal-Result
annotation={(name, (John Green)), (amount-granted, $100k)}
snapshot-time=11-25-07
record-id=R08

entity-id=(L01, empty)
entity-address=25 Inst. Rd
entity-type=Mortgage-Company
annotation={(agent, (Susan Aaron))}
snapshot-time=11-22-07
record-id=R06

entity-id=(B01, (Husband))
entity-address=10 Inst. Rd
entity-type=Buyer
annotation={(name, (John Green)), (income, $40k)}
snapshot-time=11-22-07
record-id=R05

activity-id=(A02, empty)
activity-type=Closing
activity-span=[11-26-07, 11-26-07]
annotation={(lawyer, (Louis Arthur))}
record-id=R11

entity-id=(D04, empty)
entity-address=registry/archive07/D453
entity-type=Title
annotation={(address, (2 South Rd)), (owner, (John Green))}
snapshot-time=11-27-07
record-id=R12

entity-id=(S02, empty)
entity-address=125 Mt. Rd
entity-type=Seller
annotation={(name, (Allen Snow))}
snapshot-time=11-26-07
record-id=R09

entity-id=(L01, empty)
entity-address=25 Inst. Rd
entity-type=Mortgage-Company
annotation={(agent, (Tom Aaron))}
snapshot-time=11-22-07
record-id=R10

type-name=Closing
incoming-list=((buying, Buyer)(selling, Seller) (lending, Mortgage-Company)(appraisal, Appraisal-Result))
outgoing-list=((result, Title))
annotation=empty

Figure 2.2: Example—Semantic Model of Provenance

Chapter 3

Query Model of Provenance

The ability to express an idea is well nigh as important as the idea itself.

Bernard Baruch

Intuition becomes increasingly valuable in the new information society precisely because there is so much data.

John Naisbitt

We have defined the semantics of provenance entity and provenance relationship in chapter 2. In this chapter, our goal is to develop an algebraic query model to manipulate provenance entity and relationship. This query model bases on two sub-models: a *content based model* for content based query of provenance and a *structure based model* for structure based query of provenance. By combining the power of two sub-models, we can express a lot of interesting provenance queries. In order to define the content based query model, we need to first develop two concepts: *provenance entity store* and *provenance relationship store*. They are corresponding to provenance entity and provenance relationship in the semantic model.

3.1 Content Based Query Formalism

Datum

Definition 3.1.1: A datum D is *accessible* [1], if there exists a function set F, such that for any sub-datum $d \in D$, there exists $t \in F^*$ [2], $t(D) = d$. F is called an *access set* of D, and t is called an *access sequence* of d.

Generally speaking, a datum can be retrieved either by its "name", or, if data are ordered, by its "position". Let us denote the universal datum set as $\overline{D}$, and define two basic functions for accessing a given datum.

$$\odot : \overline{D} \times \overline{NAME} \rightarrow \overline{D} \quad \text{Access By Name}$$

$$@ : \overline{D} \times \overline{INTEGER} \rightarrow \overline{D} \quad \text{Access By Position}$$

For example, let d be a static entity record as shown below:

```
entity-id=(S02 ())
entity-address=125 Mt. Rd
entity-type=Seller
annotation={(name, (Allen Snow))}
snapshot-time=11-26-07
record-id=R09
```

$d\odot$**annotation**$\odot$**name@1** is an access sequence to retrieve the first name of the seller. Note the access sequence type that consists of $\odot$ and @ is used in our content based model, as a major way of accessing a given datum. So $\{\odot, @\}$ is called the *general access set.*

Definition 3.1.2: Let d_1 and d_2 be two accessible data, with access set F_1, F_2 respectively. The *combination* of d_1 and d_2, denoted as $d_1 \oplus d_2$, is an accessible datum with access set of $F_1 \cup F_2 \cup \{@\}$, and $(d_1 \oplus d_2)@\mathbf{1} = d_1$, $(d_1 \oplus d_2)@\mathbf{2} = d_2$.[3]

[1] Atomic datum (i.e., the one without structure) like number, string, $\emptyset$, is accessible by definition.

[2] F^* is defined as follows: Let $F'=\{f(d) : v_1 = c_1, \cdots, v_n = c_n | f(d, v_1, \cdots, v_n) \in F$, c_i is constant, $1 \leq i \leq n\}$, $F^* = F' \cup F'F' \cup F'F'F' \cup \cdots$.

[3] Specifically, if d_1 is a special symbol ϵ, define $d_1 \oplus d_2 = d_2$.

Definition 3.1.3: Let $P = \{\odot, @, \oplus\}$, P^* is called the *general projection set*. Let d be a datum, and $p \in P^*$. The *projection* p of d is defined as $\pi(d, p) = p(d)$.

Definition 3.1.4: An *image* of datum is $i : \overline{D} \to \overline{D}$. Two special images: An *assertion* of datum is $a : \overline{D} \to \{\top, \bot\}$. A valuation of datum is $v : \overline{D} \to \overline{ORD}$[4].

Intuitively, an image maps a datum into another structure, e.g., atomic number, string, etc. An assertion checks whether a datum satisfies some condition. Here are two examples:
$i(d)$: if d is not static entity record, produces $\emptyset$; else if $d \odot$ **entity-type**=**Seller**, produces $(d \odot \textbf{record-id}, \textbf{S})$; else produces $(d \odot \textbf{record-id}, \textbf{O})$;
$a(d)$: if d is not static entity record, produces $\bot$; else if $d \odot$ **snapshot-time** is before **01-01-08** produces $\top$; else produces $\bot$;

Store of Data

Definition 3.1.5: A *store* of data is defined as follows:

- $\emptyset$ is a store.
- If S is a store, d is an accessible datum, $d \oplus S$ is a store.

Common Operations of Store

I. AGGREGATION

Definition 3.1.6: The *general aggregation* operation is $\Sigma(S, s, i, +)$. S is a store, s is a datum, $i : \overline{D} \to \overline{D}$ is an image of datum, $+ : \overline{D} \times \overline{D} \to \overline{D}$ is called an *adder*.

$$\Sigma(S, s, i, +) = \begin{cases} s & \text{if } S = \emptyset \\ i(S@\mathbf{1}) + \Sigma(S@\mathbf{2}, s, i, +) & \text{otherwise} \end{cases}$$

[4] $\overline{ORD}$ is a completely ordered set, e.g., the set of real number. So $\leq, \geq, =$, etc. are defined.

What follow are a few special aggregations:

Definition 3.1.7: The *selection* of a store S with assertion a, is defined as $\sigma(S,a)$=$\Sigma(S,\emptyset,a',\oplus)$.

$$a'(d) = \begin{cases} d & \text{if } a(d) = \top \\ \epsilon & \text{otherwise} \end{cases}$$

Definition 3.1.8: The *projection* of a store S with an access sequence p, is defined as

$$\pi(S,p) = \Sigma(S,\emptyset,p,\oplus).$$

Definition 3.1.9: The $\exists$-*assertion* of a store S with an assertion a, is defined as

$$\exists(S,a) = \Sigma(S,\bot,a,\vee).$$

The $\forall$-*assertion* of a store S with an assertion a, is defined as

$$\forall(S,a) = \Sigma(S,\top,a,\wedge).$$

The *count-assertion* of a store S with an assertion a, is defined as

$$\textit{count-assertion}(S,a) = \Sigma(S,0,a',\text{arithmetic-plus}). \quad a'(d) = \begin{cases} 1 & \text{if } a(d) = \top \\ 0 & \text{otherwise} \end{cases}$$

The $\in$ -*assertion* of a datum s and store S, is defined as

$$\in\textit{-assertion}(s,S) = \exists(S,a) \qquad a(d) : d = s.$$

The *difference* of store S_1 and S_2 is defined as

$$\ominus(S_1,S_2) = \sigma(S_1,a). \qquad a(d) = \neg \in\textit{-assertion}(d,S_2).$$

The *intersection* of store S_1 and S_2 is defined as

$$\overline{\cap}(S_1, S_2) = \sigma(S_1, a). \quad a(d) = \in\text{-}assertion(d, S_2).$$

Definition 3.1.10: The *maximum* of a store S with a valuation v, is defined as

$$maximum(S, v) = \Sigma(S, -\infty, v, \max^{5}).$$

II. COMPOSITION

Definition 3.1.11: The $\oplus$-*composition* of Store S_1 and S_2 is defined as

$$\oplus\text{-}composition(S_1, S_2) = \begin{cases} S_1 & \text{if } S_2 = \emptyset \\ \oplus\text{-}composition(S_2@\mathbf{1} \oplus S_1, S_2@\mathbf{2}) & \text{otherwise} \end{cases}$$

Definition 3.1.12: The $\otimes$-*composition* of Store S_1 and S_2 is defined as

$$\otimes\text{-}composition(S_1, S_2)$$

$$= \begin{cases} \emptyset & \text{if } S_1 \text{ or } S_2 = \emptyset \\ \oplus\text{-}composition(\Sigma(S_2, \emptyset, S_1@\mathbf{1} \oplus \_, \oplus), \otimes\text{-}composition(S_1@\mathbf{2}, S_2)) & \text{otherwise} \end{cases}$$

3.2 Provenance Entity Store

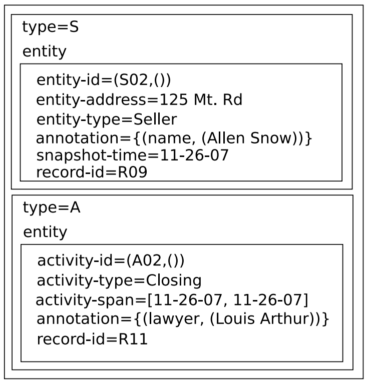
Recall from chapter 2, in our semantic model, a **provenance entity** is either a static entity record (Definition 2.3) or an activity record (Definition 2.4.2). A provenance entity is accessible. Since there are two types of provenance entity, we can define, in our query model, a provenance entity as follows:

Definition 3.2.1: A *provenance entity* is (*type*, *entity*). *type*$\in \{\mathbf{S}, \mathbf{A}\}^{6}$. If *type*=$\mathbf{S}$, *entity* is a static entity record; If *type*=$\mathbf{A}$, *entity* is an activity record.

[5] Since the range of v is a completely ordered set, *max* is defined.
[6] $\mathbf{S}, \mathbf{A}$ are character constants.

Definition 3.2.2: A *provenance entity store* is a store of provenance entities.

Here is an example of provenance entity store:



type=S
entity
entity-id=(S02,())
entity-address=125 Mt. Rd
entity-type=Seller
annotation={(name, (Allen Snow))}
snapshot-time=11-26-07
record-id=R09

type=A
entity
activity-id=(A02,())
activity-type=Closing
activity-span=[11-26-07, 11-26-07]
annotation={(lawyer, (Louis Arthur))}
record-id=R11

Example: Operators of Provenance Entity Store

The complete expressive power of content based query is characterized in section 3.1. Based on the query model in section 3.1, as an example, we point out several manipulating operators of provenance entity store. These operators are only **syntactic sugar** of the general query operations in section 3.1. They are not intended to be exhaustive.

Operator 3.2.1 (filtering by type): application of $\sigma(\mathbf{S}, \mathbf{a})$

The *filtering-by-type* operator takes a provenance entity store and a *type* as input, and produces a provenance entity store with all and only provenance entities of the *type*.

$$\sigma_{type}(S, t) = \sigma(S, \_^{7} \odot \mathbf{type} = t)$$

For example, we can use this operator to retrieve all activity records in a provenance entity store. The next two operators are used for interfacing with the structure based query model.

[7] Underscore "_" is used as a place holder for the assertion variable.

Operator 3.2.2 (filtering by record-id): application of $\pi(\sigma(\otimes\text{-}\mathbf{composition}(\mathbf{S_1}, \mathbf{S_2}), \mathbf{a}), \mathbf{p})$
The *filtering-by-record-id* operator takes a provenance entity store and a store of *record-ids* as input, and produces a provenance entity store with all and only provenance entities whose *record-ids* are in the *record-id* store. Let us denote *filtering-by-record-id* as σ_{rid}.

$$\sigma_{rid}(S, R) = \pi(\sigma(\otimes\text{-}composition(S, R), a), @\mathbf{1})^{8}$$

$$a(d): \ d@\mathbf{1} \odot \textbf{entity} \odot \textbf{record-id} = d@\mathbf{2}.$$

Operator 3.2.3 (projecting of record-id): application of $\pi(\mathbf{S}, \mathbf{p})$
The *projecting-record-id* operator takes a provenance entity store as input, and produces a store of *record-ids* of all the provenance entities in the store. Let us denote *projecting-record-id* as π_{rid}.

$$\pi_{rid}(S) = \pi(S, \odot\textbf{entity} \odot \textbf{record-id}).$$

3.3 Provenance Relationship Store

Recall from chapter 2, in our semantic model, a **provenance relationship** is a relationship between two provenance entities (Definition 2.5.1), and it can be represented as:

$$(\textit{causal-entity}, \textit{consequential-entity}, \textit{role}, \textit{annotation}, \textit{relationship-id}).$$

Since provenance entities are stored in a provenance entity store, references (i.e. record-ids) to provenance entities, instead of the entities themselves, are kept in a provenance relationship. In our query model, a provenance relationship is defined as follows:

Definition 3.3.1: A *provenance relationship* is

$$(\textit{from}, \textit{to}, \textit{role}, \textit{annotation}, \textit{relationship-id}).$$

This definition is the same as Definition 2.5.1 except that $from$ is the *record-id* of the

[8] It can also be defined as $\sigma_{rid}(S, R) = \sigma(S, \in\text{-}assertion(\_ \odot \textbf{entity} \odot \textbf{record-id}, R))$

causal-entity, and *to* is the *record-id* of the *consequential-entity*. A provenance relationship is accessible.

Definition 3.3.2: A *provenance relationship store* is a store of provenance relationships. Here is an example of provenance relationship store:

```
from=R06
to=R08
role=participant
annotation={(comment, lender)}
relationship-id=R14
```

```
from=R10
to=R11
role=lending
annotation={(payment, cheque)}
relationship-id=R15
```

Operators of Provenance Relationship Store

The operators of provenance relationship store are similar to those of provenance entity store. Here we point out two *syntactic sugar* operators for interfacing the provenance relationship store with the structure based query model.

Operator 3.3.1 (filtering by relationship-id):

The *filtering-by-relationship-id* operator takes a provenance relationship store and a store of *relationship-ids* as input, and produces a provenance relationship store with all and only provenance relationships whose *relationship-ids* are in the *relationship-id* store.

$$\sigma_{\textit{rel-id}}(S, R) = \sigma(S, \in\text{-}\mathit{assertion}(\_ \odot \textbf{relationship-id}, R))$$

Operator 3.3.2 (projecting of relationship-id):

The *projecting-relationship-id* operator takes a provenance relationship store as input, and produces a store of *relationship-ids* of all the provenance relationships in the store.

$$\pi_{\textit{rel-id}}(S) = \pi(S, \odot\textbf{relationship-id})$$

3.4 Content Based Query Model

Definition 3.4: The concepts of accessible datum, data store, provenance entity store, provenance relationship store, and other assisting concepts together with the defined operations constitute the *content based query model* of provenance.

Compared with ordinary record data, one distinctive aspect of provenance data, besides its inherently complex content structure, resides in its implication of complex relationship structure. For ordinary record data, the main goal of query design is being able to retrieve the content of data (content-oriented). However, for provenance data, it is also important to understand the underlying complex relationships (structure-oriented). Because of that, we divide the query model of provenance into two sub-models: one for complex content based query and one for structure based query. A provenance query seamlessly integrates these two sub-models.

Next, we introduce the structure based model of query.

3.5 Structure Based Query Model

Definition 3.5.1: A *structure graph* of provenance is

$$(N, E, f), \qquad f : E \to N \times N^{9}$$

Informally, $N \subseteq$ *NSpace*, is a set of *record-ids* of provenance entities. $E \subseteq$ *NSpace*, is a set of *relationship-ids* of provenance relationships. f is a function that specifies the *causal-entity* and *consequential-entity* of a relationship.

Notation: Let $S = (N, E, f)$ be a structure graph. Define

$$S \odot \mathbf{N} = N, \qquad S \odot \mathbf{E} = E \qquad S \odot \mathbf{f} = f$$

[9] f is a set of ordered triples. $\forall t = (e, n_1, n_2) \in f$, define $t@\mathbf{1} = e, t@\mathbf{2} = n_1, t@\mathbf{3} = n_2$.

Definition 3.5.2: The *general predicate* of structure graph is defined as follows: Let S be a structure graph.

$$G(e, n_1, n_2, S) = \begin{cases} \top & \text{if } \exists e, n_1, n_2 \in S \odot \mathbf{f} \\ \bot & \text{otherwise} \end{cases}$$

The general predicate is not very convenient in expressing structure graph properties. We can define more intuitive forms as follows.

Definition 3.5.3: The *primitive function set* of structure graph is

$$\{from, to, from^{-1}, to^{-1}, next, last\}^{10}.$$

Let S be a structure graph.

$$from(e, S^{11}) = \{t@\mathbf{2} \mid \exists t \in S \odot \mathbf{f},\ t@\mathbf{1} = e\}$$

$$to(e, S) = \{t@\mathbf{3} \mid \exists t \in S \odot \mathbf{f},\ t@\mathbf{1} = e\}$$

$$from^{-1}(n, S) = \{e \mid \exists e \in S \odot \mathbf{E},\ from(e, S) = n\}$$

$$to^{-1}(n, S) = \{e \mid \exists e \in S \odot \mathbf{E},\ to(e, S) = n\}$$

$$next(n, S) = \{t@\mathbf{3} \mid \exists e \in S \odot \mathbf{E},\ \exists t \in S \odot \mathbf{f},\ t@\mathbf{2} = n\}$$

$$last(n, S) = \{t@\mathbf{2} \mid \exists e \in S \odot \mathbf{E},\ \exists t \in S \odot \mathbf{f},\ t@\mathbf{3} = n\}$$

Intuitively, given an edge e, $from$ (to) retrieves the start (end) node of e. Given a node n, $from^{-1}$ (to^{-1}) retrieves the edges that start (end) at n; $next$ ($last$) retrieves the children (parents) of n. For example, to express the fact that there is a length-two path from node

[10] $S \odot \mathbf{f}$ is a 3-dimensional relation. It is decomposed into three 2-dimensional relations. Each relation is represented by two functions.

[11] When the context is clear, S can be omitted.

a to node b, in the general predicate form, it is

$$\exists e_1, e_2, c(G(e_1, a, c) \wedge G(e_2, c, b)),$$

in the primitive function set form, it is

$$\exists c(a \in last(c) \wedge c \in last(b)).$$

Definition 3.5.4: The *general selection* operation of structure graph is defined as follows: Let $S = (N, E, f)$, $f_n : N \rightarrow \{\top, \bot\}$, $f_e : E \rightarrow \{\top, \bot\}$.

$$\sigma_g(S, f_n, f_e) = S' = (N', E', f')$$

$$N' = N|_{f_n}{}^{12},\ E' = E|_{f_e} \cap \{t@\mathbf{1} | \exists t \in S\odot\mathbf{f}, t@\mathbf{2}, t@\mathbf{3} \in N'\},\ f' = f|_{E'}.$$

S' is a structure graph. We can define a few operators using the general selection operation. For example:

Example 1

Operator 3.5.1 (filtering by record-id): application of $\sigma_g(S, f_n, f_e)$

The *filtering-by-record-id* operator takes a structure graph (N, E, f) and a set N' of *record-ids* as input, and produces a structure graph with $N \cap N'$ as the set of *record-ids*, and $\{e \mid e \in E,\ \text{and}\ f(e) = (a, b),\ \text{and}\ a, b \in N \cap N'\}$ as the set of *relationship-ids*.

$$\sigma_{op3.5.1}(S, N') = \sigma_g(S, \_ \in N', \_ \rightarrow \top)$$

Example 2

Operator 3.5.2 (filtering by relationship-id): application of $\sigma_g(S, f_n, f_e)$

The *filtering-by-relationship-id* operator takes a structure graph (N, E, f) and a set E' of *relationship-ids* as input, and produces a structure graph with $E \cap E'$ as the set of

[12] $N|_{f_n} = \{a | a \in N, f_n(a) = \top\}$

relationship-ids and $\{n \mid \exists e \in E \cap E',\ \text{and } f(e) = (n, *) \text{ or } (*, n)\}$ as the set of *record-ids.*

$$\sigma_{op3.5.2}(S, E') = \sigma_g(S, (\exists e \in E' \wedge e \in (from^{-1}(\_) \cup to^{-1}(\_))), \_ \in E')$$

<u>Example 3</u>

Retrieving nodes within 2 hops starting from nodes in N': application of $\sigma_g(S, f_n, f_e)$

$$\sigma_g(S, f_n, \_ \rightarrow \top)$$

$$f_n : \exists a, b(a \in N' \wedge (\_ \in next(a) \vee (b \in next(a) \wedge \_ \in next(b))))$$

FOLTC VS. Structure Graph Operations

Because in each of the above examples, f_n and f_e are expressible in first order logic, it suffices to express the queries in the relational algebra. However, the expressiveness of first order logic is limited. For example, it cannot express the fact "node b is reachable from node a". First order logic with transitive closure (FOLTC) can express this connectivity fact. However, it still cannot express the fact "there exists a set E, such that p is a shortest path from a to b, and all the edges of p are in E".

Generally speaking, FOLTC, as a query language, has the following limitations:

1. In real life provenance application, f_n and f_e may be very complex. Thus, it is impossible or counterintuitive to express the query in FOLTC.

2. The evaluation of FOLTC has been an issue. Users have no access to optimization, and are unable to take advantage of large set of efficient graph algorithms, known domain knowledge, as well as suitable storage structure.

The advantage of structure graph operations over FOLTC includes:

1. It offers a higher, and more intuitive level of structure query constructs. "Structure Graph" becomes the first class member of query. A structure graph operator can be as complex as necessary, not limited by the expressiveness of FOLTC.

2. Users can construct a large pool of structure graph operators directly from existing graph algorithms. The algebraic query architecture makes the introduction of new operators easy, and the integration of structure graph operators to non structure based operators seamless. Users can provide different implementations for an operator, and have full control over the underlying storage structure.

3. The general selection operation $\sigma_g(S, f_n, f_e)$, as a special structure graph operation, still contains the query power and elegancy of FOLTC. It can wrap and bring FOLTC into the algebraic query architecture.

We have identified a bunch of structure graph operators that we think are important in supporting the structure based query of provenance. However, this family of operators are not intended to be complete.

STRUCTURE BASED QUERY OPERATOR FAMILY

CLASS I: SELECTION

Operators belong to this class filter a structure graph according to some criteria. The general selection operation $\sigma_g(S, f_n, f_e)$ provides the formalism for this class of operators. Simple examples include Operator 3.5.1, Operator 3.5.2. Other examples are:

Operator 3.5.3 (descendant)
The *descendant* operator takes a structure graph (N, E, f) and a set N' of *record-ids* as input, and produces a structure graph with $(N \cap N') \cup \{n \mid n \in N, \exists s \in N \cap N'$, and there is a path from s to $n\}$ as the set of *record-ids*, and $\{e \mid e \in E$, $\exists s \in N \cap N'$ and e is on a path starting from $s\}$ as the set of *relationship-ids.*

Operator 3.5.4 (ancestor)
The *ancestor* operator takes a structure graph (N, E, f) and a set N' of *record-ids* as input, and produces a structure graph with $(N \cap N') \cup \{n \mid n \in N, \exists d \in N \cap N'$, and there

is a path from n to $d\}$ as the set of *record-ids*, and $\{e \mid e \in E$, $\exists d \in N \cap N'$ and e is on a path ending at $d\}$ as the set of *relationship-ids*.

Operator 3.5.5 (in-betweener)

The *in-betweener* operator takes a structure graph (N, E, f) and a set N' of *record-ids* as input, and produces a structure graph with $\{n \mid \exists a, b \in N \cap N', n \in N$ is on a path from a to $b\} \cup (N \cap N')$ as the set of *record-ids*, and $\{e \mid \exists a, b \in N \cap N', e \in E$ is on a path from a to $b\}$ as the set of *relationship-ids*.

A Pattern Matching Operator: A slightly more complex example of selection is filtering a structure graph according to a pattern.

Definition 3.5.5: A *labeling function* is l : *NSpace* $\rightarrow$ *NSpace*. Given a simple path $p = (v_1, e_1, v_2, e_2, \cdots, e_{n-1}, v_n)$, the *label of* p *under* l, denoted as $l^*(p)$, is $l(v_1)l(v_2)\cdots l(v_n) \in$ *NSpace**. A *path pattern* is a *regular expression* over the alphabet of *NSpace*.

Operator 3.5.6 (path matching)

The *path-matching* operator takes a structure graph S, a labeling function l and a path pattern P as input, and produces a structure graph S'.

Let E'=$\{e \mid \exists p$, p is a path, $l^*(p) \in L(P)$, and e is an edge on $p\}$.

$$S' = \sigma_{op3.5.2}(S, E')$$

Figure 3.1 is an example of path pattern matching. The pattern is SX^*CX^*D.

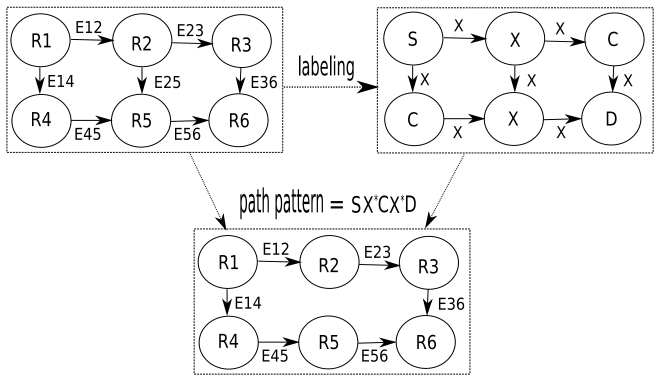


Figure 3.1: Path Pattern Matching

CLASS II: COMPOSITION

Operator 3.5.7 (union)

The *union* of two structure graphs $S = (N, E, f)$ and $S' = (N', E', f')$ is

$$S \cup_s S' = (N \cup N', E \cup E', f \cup f')$$

Operator 3.5.8 (intersection)

The *intersection* of two structure graphs (N_1, E_1, f_1) and (N_2, E_2, f_2) is

$$S \cap_s S' = (N, E, f)$$

$$N = N_1 \cap N_2,\ f = \{t \mid t \in f_1, t@\mathbf{1} \in E_2, t@\mathbf{2}, t@\mathbf{3} \in N_2\},\ E = dom(f).$$

CLASS III: ABSTRACTION

The abstraction of a structure graph is a transformation of the original structure graph into another structure graph which is expected to be conceptually more comprehensible and revealing than the original one.

Operator 3.5.9 (abstracting by record-id)

The *abstracting-by-record-id* operator takes a structure graph (N, E, f) and a set N' of *record-ids* as input, and produces a structure graph with $N \cap N'$ as the set of *record-ids*, and the set of *relationship-ids* defined as follows: Let C_0=$\{(a, b, 0) \mid a, b \in N \cap N', \exists e \in E, f(e) = (a, b)\}$, C_1=$\{(a, b, 1) \mid a, b \in N \cap N'$, there exists a path, with length larger than 1, from a to b, and no intermediate node on the path is in $N \cap N'\}$, $C = C_0 \cup C_1$. For every $(a, b, c) \in C$, if $c = 0$, then for any $e \in E$ with $f(e) = (a, b)$, include e in the set of *relationship-ids*; else if $c = 1$, generate a new unique *relationship-id* for (a, b) and include it in the set of *relationship-ids*.

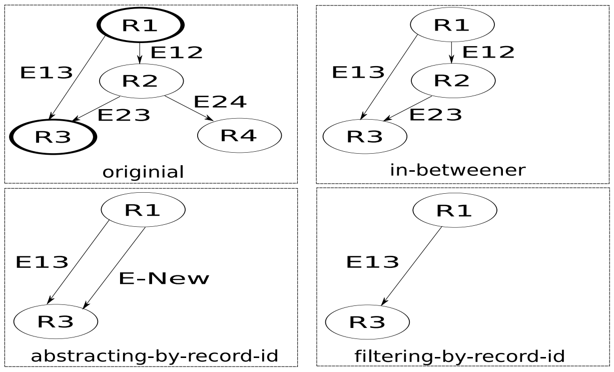


Figure 3.2: Reducing A Structure Graph By *record-id*

Figure 3.2 contrasts three aforementioned operators, which reduce the same structure graph in different ways[13]. As an example of application, the *abstracting-by-record-id* operator can be used to make an "activity" view [14] of the provenance structure.

<u>CLASS IV: NAVIGATION</u>

The navigation of structure graph provides a set of interactive operations for users to navigate the structure graph step by step. The set of navigational operators is corresponding to the *primitive function set* (**Definition 3.5.3**).

[13] $\{R1, R3\}$ is the target *record-id* set.

[14] i.e. Skip all the "static entities", while still maintaining the connectivity

CLASS V: METRICS AND PROPERTIES

Based on the concept of structure graph, it is also possible to define ad hoc operators that fit particular needs for *analysis* of provenance structure. For example, *a*) operator that returns "shortest path" between two provenance entities. *b*) operator that returns the provenance entities that have "out-degree" larger than a number.

Definition 3.5.6: The concept of *structure graph* and its relevant query operators constitute the *structure based query model* of provenance.

3.6 Combining Two Sub-models of Query

Figure 3.3 roughly illustrates how the content based model and the structure based model fit together to carry out provenance query.

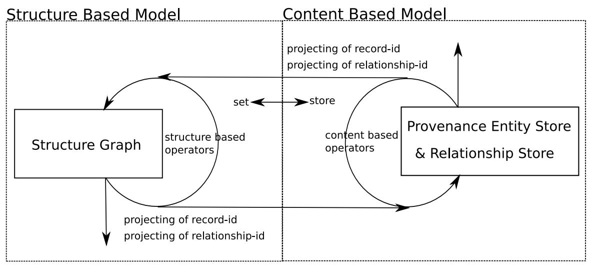


Figure 3.3: Query Model of Provenance

A Fictitious Example

We conclude this chapter with an illustrative scenario of food safety tracking. In today's globalized economy, the production and consumption of goods are often distributed at different places. For example, a manufacturer may use parts and material from different countries, and the final products are shipped around the globe. In this case, it is helpful to keep track of the provenance of goods for quality assurance, dispute settlement, etc.

Figure 3.4 shows an assumptive workflow of powdered milk production and consumption.

Powdered milk is produced in country Z, and is exported to country X and country Y. Some confectionary companies in country X use imported powdered milk from country Z in their production of chocolate candy. In this scenario, we don't consider how data is collected, but focus on the intuition of how provenance queries are used.

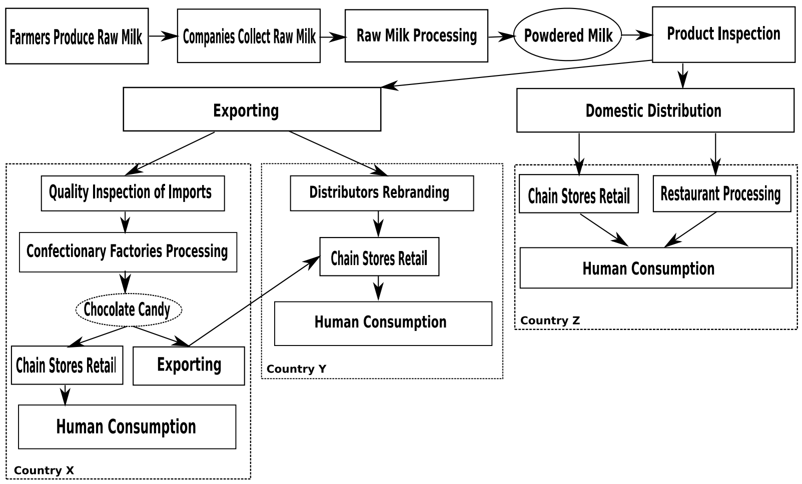


Figure 3.4: Powdered Milk Production And Consumption

Q1: Imagine one day, a brand of powdered milk produced in country Z is found contaminated. What brands of chocolate candy are affected in country X? How much is the total worth of the affected products?

S1: *a*) Find the problematic batch of powdered milk by content based query (by “brand name”, “address” and “production time”). *b*) Get the descendants of the problematic batch by structure based query (the *descendant* operator). *c*) Conduct content based query on the descendants (e.g., “type”=“chocolate candy”, “address”=“country X”) to get affected brands of chocolate candy in country X. *d*) Use aggregation to calculate the total worth.

Q2: How was the problematic batch of powdered milk produced, transported and processed to make an affected brand of chocolate candy? Was the powdered milk inspected both before and after exportation?

S2: *a*) Find the problematic batch of powdered milk and the affected brand of chocolate candy by content based queries. *b*) Relate the problematic batch of powdered milk and the affected brand of chocolate candy by using the in-betweener operator. *c*) In the structure graph obtained from the previous step, use the path matching operator to check whether there exists a path matching pattern $(\mathbf{inspection})X^*(\mathbf{exportation})X^*(\mathbf{inspection})$.

Q3: There are two separate brands of chocolate candy that have received complaints. Do they share some kind of similarity in their production processes?

S3: *a*) Find these two brands by content based queries. *b*) Find the ancestors of each brand separately (the ancestor operator). *c*) Compare the results obtained from the previous step by, e.g., intersection of structure graphs.

Chapter 4

Implementation

Dreams pass into the reality of action. From the actions stems the dream again; and this interdependence produces the highest form of living.

Anais Nin

4.1 Framework of Implementation

The implementation of `Butterfly` is still in the initial stage and is not the focus of this thesis. Figure 4.1 briefly shows the architecture of `Butterfly`.

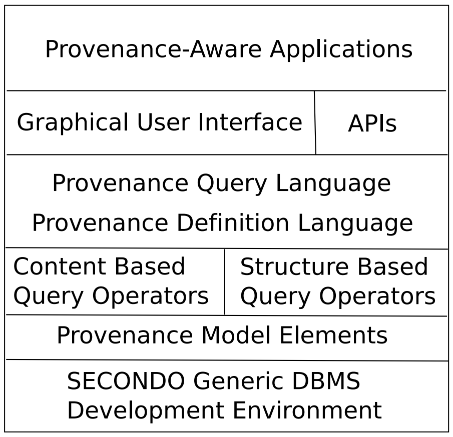

Provenance-Aware Applications
Graphical User Interface
APIs
Provenance Query Language
Provenance Definition Language
Content Based Query Operators
Structure Based Query Operators
Provenance Model Elements
SECONDO Generic DBMS Development Environment

Figure 4.1: `Butterfly` Architectural Stack

We use SECONDO as the development platform for our provenance model. SECONDO [11] is a generic DBMS developing environment for implementing and experimenting a wide range of data models and query languages. SECONDO uses BerkeleyDB as the storage engine. The formal basis of SECONDO is second-order signature (SOS) [16]. SOS uses two coupled signatures to describe a data model and an algebra over that data model. The first signature declares the types and type constructors. The second signature specifies available query operations over the types or terms of type constructors in the first signature. The following is an example of first signature and second signature.

FIRST SIGNATURE

KINDS

TYPE CONSTRUCTORS

→ ProvEnt	StaticEntityRecord, ActivityRecord
→ ProvRel	ProvRelationship
→ ID	Id
→ Val	Int, Real, String, Bool
→ PathPattern	Pat
Any → Set	Set
Any → Store	Store
→ StructGraph	Simple, Clustered, Topological

SECOND SIGNATURE

QUERY OPERATORS

Store(Any) → Set(Any)	**store_to_set**	_ #
Set(Any) → Store(Any)	**set_to_store**	_ #
Store(ProvEnt) × String → Store(ProvEnt)	**type**	_ # _
Store(ProvEnt) × (ProvEnt → Bool) → Store(ProvEnt)	**filter_cond**	_ # [_]
Store(ProvEnt) × (ProvEnt → Val) × (Val × Val → Val) → Val	**agg**	_ # [_] [_]
Store(ProvEnt) × Store(ProvEnt) → Store(ProvEnt)	**union**, **minus**	_ # _
Store(ProvEnt) → Store(ID)	**id**	_ #

$\underline{\text{Store}}(\text{ProvEnt}) \times \underline{\text{Store}}(\text{ID}) \rightarrow \underline{\text{Store}}(\text{ProvEnt})$	**filter_id**	_ # _
$\text{StructGraph} \rightarrow \underline{\text{Set}}(\text{ID})$	**record_id, relation_id**	_ #
$\text{StructGraph} \times \text{StructGraph} \rightarrow \text{StructGraph}$	**union, intersection, minus**	_ # _
$\text{StructGraph} \times \text{ID} \rightarrow \underline{\text{Set}}(\text{ID})$	**in_edges, out_edges**	_ # _
$\text{StructGraph} \times \text{ID} \rightarrow \text{ID}$	**start_node, end_node**	_ # _
$\text{StructGraph} \times (\text{ID} \rightarrow \text{ID}) \times \text{PathPattern} \rightarrow \text{StructGraph}$	**path_pat**	_ # [_] _
$\text{StructGraph} \times \underline{\text{Set}}(\text{ID}) \rightarrow \text{StructGraph}$	**in_between, abstract_id, filter_id, ancestor, descendant**	_ # _

To implement the provenance algebra module, we need to provide implementation to both type constructors and query operators, and then register them to the SECONDO system catalogue. A type constructor is used to validate whether a piece of data is compatible with the specified type. Query operators call type constructors to validate input data types, then perform the actual execution of queries. Note in the first signature, $\underline{\text{Simple}}$, $\underline{\text{Clustered}}$, $\underline{\text{Topological}}$ are three physical representations of structure graph. In the second signature, syntax of query operators can also be specified. For example, "_ # _" specifies two operands are placed around the operator.

4.2 A Uniform Data Structure For Provenance

Provenance model elements vary in form. However, we can use *nested list* as a uniform logical data structure for provenance. For example, what follows is a nested list representation of a static entity record:

((entity-id (S02, empty)) (entity-address (125 Mt. Rd)) (entity-type (Seller))
(annotation ((name (Allen Snow)))) (snapshot-time (11-26-07)) (record-id (R09))).

The advantage of nested list over other data structures is that it is extensible and there are already lots of handy, well-defined operations for nested list, for example, iteration, filtering, and searching.

4.3 Storage of Structure Graph

In this section, the storage issue of structure graph is discussed. Any file organization of structure graph must support the following operations:

- Insertion of nodes and edges.
- Retrieval of nodes and edges.
- Finding the incoming edges and outgoing edges of a given node.
- Finding the starting node and ending node of a given edge.

A Simple File Structure

A simple file structure that meets the requirement is as follows:

Node Table

Node Key	Node Label

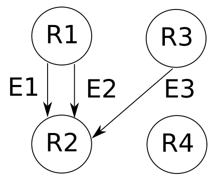
Edge Table

Start Node	End Node	Edge Key	Edge Label

Figure 4.2: A Simple File Structure

Node key or edge key is automatically generated when a node or an edge is inserted. Figure 4.3 shows a structure graph stored using the simple file structure.

Structure Graph



Node Table

1	R1
2	R2
3	R3
4	R4

Edge Table

1	2	1	E1
1	2	2	E2
3	2	3	E3

Figure 4.3: A Structure Graph Stored Using The Simple File Structure

It is easy to verify that the simple file structure satisfies our requirement.

Reducing Disk I/Os By Clustering

Since the structure graph may become so large that in-memory storage and algorithms can become infeasible, we need to use a more realistic I/O model which reflects the cost of accessing the secondary storage. The problem of file organization is studied in [2,18,19,22]. In our scenario, a good file organization must be able to reduce disk I/Os when performing structure graph traversal. Figure 4.4 shows an example. Layout 1, is a random placement of edges in blocks. Layout 2, try to keep all out-going edges of a node in a block. To retrieve the out-going edges of node a, layout 1 needs to read 3 blocks. With layout 2, if there exists a node index that tells which blocks all out-going edges of a given node reside, only one block needs to be read. Since retrieving out-going edges of a given node is a frequent operation in structure graph traversal, layout 2 is better than layout 1.

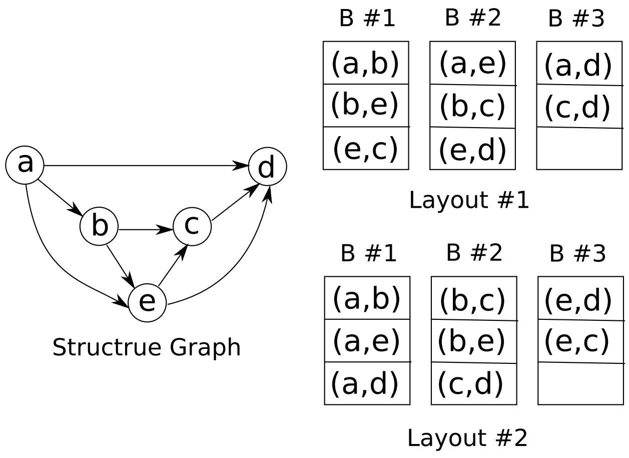


Figure 4.4: Different Storage Layouts

The lesson learned is to keep relevant information of a node stored closely. Since when performing graph traversal, both incoming and out-going edges of a given node are frequently accessed, those information should be clustered together. This gives rise to an alternative file structure discussed below.

An Alternative File Structure

Figure 4.5 shows an alternative file structure. Each entry includes a node, its list of incoming-edges, and list of outgoing-edges. In order to retrieve a node efficiently, a node index is also needed.

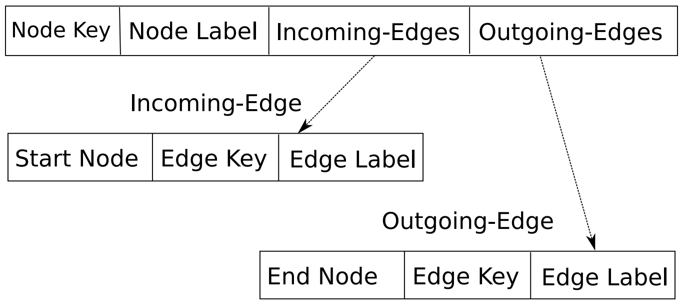


Figure 4.5: An Alternative File Structure

So far, we have not considered the relative order of node entries. The intuition is to store closely node entries that are likely accessed together. We can do this by using any clustering algorithm (e.g. k-mean) to group the node entries. Node entries in the same group are placed into the same block. The problem is how to define a distance function (measure of relevancy) for node entries. One straightforward distance function is to assign 1 for two nodes that are neighbors, and 0 otherwise. Because it may not be feasible to run the clustering algorithm on the complete set of node entries. The clustering process needs to be incremental. The node entry placement problem has been studied in [2,18,19]. We have found our approach very similar to [19]. When a node entry is inserted, a block which contains most of its "neighbors" is chosen. When a block turns full, a minimum cut of the graph induced by the node entries within the block is computed. The two sets created by the cut are put into separate blocks. [19] also uses the caching technique to further reduce access to the secondary storage.

Chapter 5

Conclusion and Future Work

Our enemy is motivated by hatred and will not stop planning more plots against until they are ultimately defeated. Today was an important and necessary victory in the war, but there is a long road ahead. We must remain committed if we are to succeed and protect our liberty.

Timothy Murphy

5.1 Provenance Is Complex, We Provide *The* Solution

Provenance is useful, however it is complex. We find out its complexity reside mostly on three aspects.

1. Provenance is pervasive phenomenon, but rather elusive and hard to be formalized. It is difficult to get a realistic, but still manageable semantic model of provenance.
2. Elements of provenance have very complex content structures. "Flat" data model with limited extensibility and flexibility cannot satisfy growing needs of applications.
3. Complex query on relationships among provenance elements is of great interest. Some of such queries are counterintuitive or impossible to be formulated in n-th order logic. The performance of query evaluation needs to be real life.

To address these problems, we offer a general semantic model of provenance, which abstracts an application system as a world composed of *static* and *dynamic* elements. Those elements interact with each other within an *environment*, playing specific *roles*. There are other provenance semantic models, which are either over-realistic towards exact knowledge representation of provenance at the cost of increasing query complexity and overloading the users, or over-simplied towards expressing simple written queries at the cost of losing interesting provenance information. We strike a balance between these two extremes and propose a semantic model that we think any provenance system should at least support.

We create a query model with clearer semantics by separating it into the *content based query model* and the *structure based query model*. By introducing the concept of *accessible datum* and *store of data*, we offer a flexible and extensible *content model* for provenance elements. We also offer a set of general and expressive operations for the content model.

We use *structure graph* as the major formalism of structure based query. It employs an algebraic query architecture and uses structure graph as the first class member of query processing while still elegantly keeping the expressiveness of FOLTC in its formalism. The advantage of using structure graph as the first class query element is that large set of efficient graph algorithms can be directly used, and the expressiveness of structure based query is not limited by FOLTC. The algebraic approach makes it easy for users to choose different implementations for a graph operation, and to seamlessly integrate graph operations with other query operations.

5.2 Comparison, Evaluation And Contribution

5.2.1 Comparison of Query Capacity With Existing Systems

We compare to a class of existing systems, instead of comparing to individual systems. Generally speaking, any provenance investigation can fall into two categories: *a*) search/transform provenance data based on some specification on values. *b*) query the structure of provenance among the provenance data.

So what are missing from the existing provenance management systems, but get covered in our approach, in terms of query capacity?

1. The first one is "generic annotation", a truly extensible recording mechanism compared with record (recall the definition of datum and store, in contrast to fixed schemas used in most existing systems), and the ability to access and transform annotation in truly general way (recall all those general operations on store and datum, and the ability to express arbitrary conditions by allowing general functions in the formalism). The reason why most existing systems cannot support queries based on complex qualification is that they don't even store/structure such information in a machine readable format (i.e., a class of systems do not support flexible content structure), let along general operations on them.

2. The second one is the ability to express complex structural queries that are not expressible (or hard to express) in first order logic with transitive closure (or more generally speaking, in nth order logic). Basically, existing systems use some kind of transitive closure to find descendants and ancestors. Removing all kinds of distracting covers of those systems, their query models are relational. For this reason, they cannot even express the following three basic classes of structural queries. *a*) Given two provenance entities, return the "shortest" path between these two entities. *b*) Does a provenance graph satisfy a property? For example, does it contain a subgraph that conforms to a pattern (e.g., in its simplest form, a path pattern)? *c*) Abstraction of provenance graphs (e.g., the abstracting-by-record-id operator). Relational systems can only do some simple navigational operations (possiblely in a recursive way). In contrast, our approach keeps the elegancy of first order logic with transitive closure by using the general selection operation (recall that this operation is itself a structure graph operation, and it can wrap first order logic with transitive closure into our structure-graph-operation approach) and is not limited by the expressibility limitation of nth order logic by using structure graph operations

(i.e., structure graph as first class query member). The following examples show the differences:

Ex1 (can be answered by both approaches): Get the descendants of a provenance entity. Even this query can be answered by both approaches (first order logic with transitive closure or structure graph operation). Relational approach can only return descendants themselves. How about their relative order on the paths? By using structure graph operation (the descendant operator), a complete structure graph is returned.

Ex2: Find the "shortest" path between two provenance entities. This query cannot be formulated by the relational approach. In the algebraic structure graph approach, simply call a "shortest path" operator that is well implemented based on some graph traversal algorithm.

Ex3: Find a path that satisfies a pattern. This query cannot be expressed in relational query. In our approach, just call the path pattern operator.

Note that we still maintain the expressibility of existing systems in our approach by using the general selection operation.

5.2.2 Comparison of Query Capacity With Existing Models

Although there may be different data models (e.g., relational, XML, graph) that can be used to address different aspects of the provenance management problem separately, there is no one solution that addresses them all under the application of provenance management, and integrates seamlessly into a query architecture for provenance.

Even separately speaking,

Relational Plus User Defined Functions: It does not support complex structure. Besides, it does not integrate well with other operations (models).

XML: One criterion that distinguishes a data model from another data model is the operations that they can offer. We can say our structure graph is different (more powerful) from (than) XML, by pointing out a few operations that XML does not currently support. What goes beyond is that graph is a more natural (or intuitive, general) abstraction of relationships than XML. And XML mixes the content with structure, so for relationship modeling, graph is better (clear and more expressive).

Graph: *a*) It does not support complex content management. *b*) Graph is a widely used model for representing relationships. What distinguish a graph model from another graph model can be thought of as the core sets of algebra (operations) they define. Our algebra is provenance algebra (i.e. focuses on provenance operations). There can be another graph algebra that focuses on query of graph property, for example, retrieve a planar subgraph of the original graph. That is why we call our graph "structure graph" instead of simply "graph" to stress the difference (connoted by the operations) from other graph models. We believe our classification and abstraction of graph operations for provenance is good and useful. *c*) For the graph operation part, our approach does not miss the fine-grained expressibility of formal logic. As far as we know, other query formalisms that use graph, either use a pure algebraic approach (missing the expressibility of formal logic) or pure logical query approach (being limited by the expressibility of nth order logic).

5.2.3 Main Contribution

Our approach well defines a minimum semantic model of provenance and a core set of operations that any provenance management system must support, no matter whether they adopt Butterfly. As far as we know, no formal work has been done in summarizing and categorizing the operations that must be supported in provenance management systems. We also suggest a query architecture (a content based sub-model and a structure based sub-model; As far as we know, no existing system has separated the structure based query from the content based query) which is not limited by the expressibility constraint of nth order logic (but still contains it), and permits users the right to do their own optimization (e.g., choose their own algorithms and storage schemes) and extend their systems (by introducing ad-hoc operators into the query algebra).

5.3 Future Work

Future work includes the following:

1. Implementation of content based operators and structure based operators.

2. Optimization of structure based operators and structure graph storage.

3. Provide language and graphical user interfaces to the query engine. Display query result in a comprehensible way.

4. Demonstrate novel provenance-driven applications that are based on Butterfly.

Bibliography

[1] Dublin core metadata element set, version 1.1: Reference description, 2003.

[2] J. Banerjee. A dag clustering algorithm for design configurations. In *ACM '87: Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 444–451, 1987.

[3] R. A. Becker and J. M. Chambers. Auditing of data analyses. In *SSDBM'86: Proceedings of the 3rd international workshop on Statistical and scientific database management*, pages 78–80, 1986.

[4] O. Benjelloun, A. D. Sarma, C. Hayworth, and J. Widom. An introduction to uldbs and the trio system. *IEEE Data Engineering Bulletin, Special Issue on Probabilistic Databases*, 29, 2006.

[5] D. Bhagwat, L. Chiticariu, W. Tan, and G. 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, 2004.

[6] R. Bose and J. Frew. Composing lineage metadata with xml for custom satellite-derived data products. In *in SSDBM*, pages 275–284, 2004.

[7] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[8] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 539–550, 2006.

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

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

[11] S. Dieker and R. H. Güting. Plug and play with query algebras: Secondo. A generic DBMS development environment. In *IDEAS '00: Proceedings of the 2000 International Symposium on Database Engineering & Applications*, pages 380–392, 2000.

[12] C. Pancerella et al. Metadata in the collaboratory for multi-scale chemical science. In *DCMI '03: Proceedings of the 2003 international conference on Dublin Core and metadata applications*, pages 1–9. Dublin Core Metadata Initiative, 2003.

[13] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *In Proceedings of the 14th Conference on Scientific and Statistical Database Management*, pages 37–46, 2002.

[14] J. Frew and R. Bose. Earth system science workbench: A data management infrastructure for earth science products. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, pages 180–189, 2001.

[15] P. Groth, S. Jiang, S. Miles, S. Munroe, V. Tan, S. Tsasakou, and L. Moreau. An architecture for provenance systems. Technical report, Technical report, Electronics and Computer Science, University of Southampton, 2006.

[16] R. H. Güting. Second-order signature: a tool for specifying data models, query processing, and optimization. *SIGMOD Rec.*, 22(2):277–286, 1993.

[17] D. P. Lanter. Design of a lineage-based meta-data base for gis. *Cartography and Geographic Information Systems*, 18, 1991.

[18] P. A. Larson and V. Deshpande. A file structure supporting traversal recursion. *SIGMOD Rec.*, 18(2):243–252, 1989.

[19] A. O. Mendelzon and C. G. Mendioroz. Graph clustering and caching. In *Proceedings of the 13th international conference on Computer science 2 : research and applications*, pages 37–46, 1994.

[20] S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of recording and using provenance in e-science experiments. Technical report, Journal of Grid Computing, 2005.

[21] L. Moreau, J. Freire, J. Futrelle, R. McGrath, J. Myers, and P. Paulson. The open provenance model. Technical report, Technical Report 14979, ECS EPrints repository, 2007.

[22] P. Scheuermann, Y. C. Park, and E. Omiecinski. Heuristic reorganization of clustered files. In *3rd International Conference, FODO 1989 on Foundations of Data Organization and Algorithms*, pages 16–30, 1989.

[23] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance techniques. Technical Report Technical Report IUB-CS-TR618, Indiana University, Bloomington IN, 2005.

[24] R. D. Stevens, A. J. Robinson, and C. A. Goble. mygrid: personalised bioinformatics on the information grid. *Bioinformatics*, 19 Suppl 1, 2003.

[25] A. Vahdat and T. Anderson. Transparent result caching. In *In Proceedings of the 1998 USENIX Technical Conference*, 1998.

[26] A. Woodruff, M. Stonebraker, A. Woodruff, and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *In ICDE*, pages 91–102, 1997.

[27] J. Zhao, C. Goble, M. Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-science. In *In Proc. of the Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data*, 2003.