INCORPORATING PROVENANCE IN DATABASE SYSTEMS

by

Adriane P. Chapman

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:

Professor Hosagrahar V. Jagadish, Chair
Professor David J. States
Associate Professor Mark S. Ackerman
Associate Professor Jignesh M. Patel

DEDICATION

To Shaggy Dog

My constant, loyal, loving companion

ACKNOWLEDGEMENTS

David Gammack, I couldn't have done this without you. Thank you. You have been my friend, comfort, support and happiness through the years. I appreciate all the time I've had with you, and look forward to all the adventures we haven't had yet.

Mom and Dad, thanks for everything[1]. In addition to the raising me, putting up with me and loving me bit, there were a few things you taught me that surfaced over and over again during grad school. Dad, thank you for all the logic questions and problems you posed to me over the years. I think you succeeded in teaching me to think rationally. Thank you too for the stubbornness; it served me well in this venue. Mom, thank you for teaching me how to work and learn. The hours you spent checking homework, editing essays and shuttling me to sports, school and the library are very much appreciated. Thank you both for teaching me about compassion, friendship and being a bitch when necessary. None of this would have been worth it without the friends and loved ones those skills provided me.

Of those friends and loved ones, I'd like to start with Jenny. I was in the darkroom at *The Tech* when you returned my call looking for a roommate. I never thought my world could change so much from such a simple event. Thank you for seeing the best and the worst of me, and still choosing to be my best friend. My world is brighter, and so much wonderfully sillier, with you in it. Lita and Jason, thank you for always listening whenever I drop into the world, and not minding when I drop out again. Lita, someday I will be

[1] How do you express all the thanks and love of a lifetime? I love you so much.

brave enough to follow your lead, and follow my dream to become a Park Ranger. You both inspire me to try to live the way I *want* to live, not the way the world says someone with my socio-economic-education check-boxes should live. Thank you both too, for making me feel like I have a home away from home whenever I need it, and keeping up a constant supply of the best chocolate in the world[2]. Chocolate always eases the pain. Col and Lisa, thank you both for putting up with my social absences, and providing an ear whenever I needed it. Mandy, you have provided me with a template for how I would like to behave when the chips are down (dealing with the event with strength, grace and humor). Shanshan, thank you for the laughter, and your example of squeezing every last drop of enjoyment out of a day. Where shall we move next? Jodi, thanks for indulging in my knitting habit with me and years of delicious dinners. Christian, thank you for going through this whole Ph.D. process at the same time as me, bitching with me when times were rough, and celebrating the good things. I actually really came to like Ann Arbor while roaming it with you two. Jen and Chris, thank you for always being up for an adventure, and finding the ludicrousness in every situation. Tim, thank you for always having a happy, quiet space, and not minding an unsocial guest re-grouping on your couch. You were integral in getting me through the final push[3]. Neamat, thank you for helping me to appreciate the joy in the little things like running a race or canoeing down a river. Rachael, you were the first person I met in Ann Arbor. Thank you for being a climbing buddy, a running buddy, and just a buddy. Thank you for all the support and encouragement during my injury and recovery, and for always being a good shoulder.

My entire family is responsible for this thesis. Thank you for putting up with things like me leaving Thanksgiving dinner to run more experiments, or arriving almost unannounced on your doorstep looking for a place to unwind. Great Uncle Bill, thank you for reading

---

[2] La Burdick

[3] And thanks for the last minute data entry.

every single paper I wrote during my Ph.D.; it's nice to know that *someone* has read them. Cousy Lisa, thank you for the constant calls and for nattering away the hours with me. Thank you too for the reminders that the grass always looks greener on the other side. Cousies Jason and Jina, thank you for always making the time to celebrate everything from holidays to weddings with me. Uncle Jon, Aunt Faith and Cousin Dollie, thank you for the guidance and encouragement in starting my own company. Cousins Dollie and Claude, thank you, thank you, thank you for always having your guest room open for me. I would never have finished if I hadn't had a place and understanding friends to pop off to at a moment's notice for relaxation or adventure. Uncle Jon, thank you too for providing me a place to escape to. Little brother, your excellent choice of xmas presents always provided me with the perfect escape without leaving Ann Arbor. Aunt Elaine, thank you for always taking an interest in what was going on in my life, and providing all the fun at family gatherings. Uncle Bob, Aunt Linda, Cousy Dawn, Cousy Todd, Cousy Eddy, Cousy John-Michael, Mary Mendryk, Cousy Allison and Cousy Stephen, just knowing that if I ever needed it, you would move the world for me, has meant so much. I'd also like to remember the two family members I lost during my Ph.D. I'd like to thank my godfather, Uncle Bill, for his quiet encouragement, and my Aunt Ceil who demonstrated a passion for everything she did. Finally, thank you to my two grandmothers. Grandmama, thank you for teaching me how to make apple cobbler (it's always a hit), showing me what true tenacity is, and believing I spent the last seven years in Montreal. Gram, you were gone before I even started this thesis, but it never would have happened without your love, making my first bed at MIT, your guidance and encouragement. Thank you.

My co-workers could have made the last seven years hell. Instead, they were responsible for making them bearable. In my seven years in the Database Group, I never heard a bad interaction between anyone. We sweat together through every deadline, and provided support and encouragement for each other. It's an amazing group of people who can care enough to cut themselves to the quick for their own submission, and instead

of going home to get some much needed rest, will turn around and help someone still struggling. Outside of deadline mode, everyone in the group has always been willing to take the time from their own work to help someone else who is struggling. I count myself lucky to have known such an incredible group of people. Thank you, all of you. Special thanks to Magesh and Cong for not killing me during the endless task of building MiMI. You two are the best. Thanks to You Jung, Yuanyuan, Neamat, Willis, Arnab, and Bin for helping to relieve stress and talk about life outside work during our daily lunch and card game. Willis, thank you for always being willing to get a cup of tea. Glenn, thank you for being a kind and patient teacher in subjects ranging from project management to coding languages. Neamat, Prasad and Nate, thank you for being the cheering squad. Bin, Nuwee, Melanie, Sushant and Shurug, thank you for always being willing to debug a bit of code with me, or teach me how a certain module works. DB Group, it would have been impossible without you. Outside of the DB Group grad students, I'd like to thank Peter Buneman, James Cheney, Kristen LaFevre, Prakash Ramanan, Mark Ackerman, David States, Jignesh Patel and Toby Teorey for their comments, criticisms, assistance and guidance.

I would have cracked years ago without outside activities. First, thank you to the folks at the Spine Program, Andréa Brisson and Naomi White for helping me to walk again. While we're on the subject, a special thank you to all of my friends, family and co-workers who pulled me through that horrible, painful time. I wouldn't be here today without you. Also, thank you Bethany, Rudolpho, Lane 4 and the rest of the Ann Arbor Masters team for years of companionship and swimming fun. Thanks as well to the Tortoise and Hare Tuesday Running Group for accepting me and running with me even when I couldn't keep up.

Finally, thank you Jag. How can I say thank you for everything you've done for me? You have fought in my corner innumerable times. You have spent countless hours teaching me. You have answered every impertinent personal "life" question. You have calmed and

comforted me. You have pushed me to better myself. You have offered me tissues and laughed with me. Thank you for taking me on as a student, and thank you for giving me so much.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER I

INTRODUCTION

Data *provenance* is the history of a piece of data. This includes a description of its origin and the processes by which it was modified throughout its lifetime. For example, in the biological domain of protein interactions there are over a hundred databases. However, only a small number of these are 'source' databases that accept experimental results. The rest are snapshots of these source databases and other secondary sources. Provenance associated with the data in these "snapshot" databases describes the original source database, publications and curation properties. The provenance for a protein interaction reported in Michigan Molecular Interactions (MiMI) [36, 83], a secondary protein interaction database, contains information about the original publication and whether the curation, or the addition of information to the dataset, occurred by hand or via an automated NLP process [118]. Provenance is essential information to assist users in understanding the significance and veracity of a piece of data. Figure 1.1 shows a snapshot of MiMI. Using provenance, a user can determine which underlying source a data item originated from, e.g. the molecule Chk1 came from HPRD. Using this information, a user can assign a confidence value on the Chk1 data item's credibility. Additionally, one can determine what processes occurred to create the dataset. The provenance attached to the Wee1 molecule shows that it underwent a merge process. This information is needed in order to understand why the Wee1 record in MiMI does not match the one found in HPRD.

Currently there are two provenance approaches in the literature: provenance generated

within workflow frameworks, and provenance within a contained relational database. Workflow provenance allows workflow re-execution, and can offer some explanation of results. Meanwhile, within relational databases, knowledge of SQL queries and relational operators is used to express what happened to a tuple. However, there is a disconnect between these two areas of provenance research. Techniques that work in relational databases cannot be applied to workflow systems because of heterogeneous data types and black-box operators. Meanwhile, the real-life utility of workflow systems has not been extended to database provenance. Myriads of systems that need provenance fall in the gap between provenance in workflow systems and databases. For instance, when creating a new dataset, like MiMI [36, 83], using several sources and processes, a database is used, but does not contain the entire set of processes. Likewise, in a system such as MiBlast [85], that generates sequence alignments, a set series of modules are executed, but no workflow system is used. These hybrid systems cannot be mashed into a workflow framework and do not exist solely within a database. This work solves issues that block provenance usage in hybrid systems.

1.1 Contributions

Research in database provenance studies how to express provenance of tuples cleanly and efficiently by relying upon SQL query specification and relational operators. Provenance in workflow systems captures enough information for 'playback'. However, if a hybrid system does not fit inside a relational database or workflow framework, no strategies exist for working with provenance.

The Michigan Molecular Interactions (MiMI) [36, 83] database is one example of a hybrid system. It uses a relational database, but expands well beyond. Moreover, the processes that create MiMI cannot be used within a workflow system. Below, we first describe MiMI to provide a concrete example of a hybrid system. Then we describe the provenance needs in these hybrid systems.

1.1.1 MiMI 1.0

The problems that inspired this work were uncovered while building Michigan Molecular Interactions (MiMI). Examples from MiMI are used throughout this work, and the algorithms and implementations utilize MiMI provenance information. As such, a brief introduction to MiMI, its construction and usage, is provided here.

Both the volume and number of data sources in molecular biology are increasing rapidly. Often multiple resources provide overlapping, partial and polymorphic views of the same data. This data is stored and published in a diverse set of data sources. Each source is distinct with respect to its biological focus (e.g. SNPs, gene promoters, etc), organism (e.g. fly) and format (e.g. tab delimited file, relational database, etc). Even after narrowing the problem down to a subset of biological information, such as protein interaction information, there is a deluge of information. With such a rich variety of sources to choose from, a scientist who wishes to visualize the full picture concerning a particular protein must visit a myriad of sites, learn a plethora of names, aliases and identifiers, compile information from journal papers, and then piece the resulting jigsaw puzzle together. This task becomes even more onerous due to several complicating factors. First, no naming or identification scheme has been agreed upon. Thus, the scientist must painstakingly map her protein of interest to a series of different names and identifiers. Second, many interaction databases, or even lab web pages, place an interaction in the public domain even if it is supported by only one experiment. This forces scientists to search through multiple databases for conflicting or corroborating evidence. Third, heterogeneous sources storing information in their own unique formats force scientists to become programmers in order to trawl through large volumes of data and reorganize it into an understandable format. Finally, once a researcher has gathered data from several sources, sifted through and amalgamated it, there is usually no trail left linking the data to its original sources. At this stage, if the scientist discovers conflicting pieces of information existing in her amalgamated view, she has no way of making an informed

decision about how to correct the data.

The work of [16, 53] minimizes the burden on the user by integrating a large number of disparate sources containing information over a range of attributes such as expression, structure and family. However, while the integration is from a large number of heterogeneous sources, it is a shallow integration. Michigan Molecular Interactions (MiMI) helps scientists search through large quantities of information by integrating all information from participating data sources through the process of deep-merging [37]. As a result of deep merging, redundant data are removed and related data are combined. Moreover, the provenance of each piece of information is tracked throughout the system, allowing scientists to choose which data to trust [21]. Trust is Qi to the usage of information by scientists. MiMI allows users to ask more advanced questions than each of its component databases can answer independently. MiMI attempts to relieve scientists of the burden of tracking down multiple sources, mapping multiple identifiers and merging redundancies. By integrating well-known datasets such as HPRD [114] and BIND [7], MiMI creates a deep-merged repository that is a synergy of all the merged datasets. By such integration, MiMI shows scientists when facts are corroborated by different datasets, and when facts are contradicted among datasets. MiMI's integration is distinctly different from the approach used by the International Molecular Exchange Consortium (IMEx). While several of the integral components of MiMI also belong to IMEx, the tasks are very different. IMEx is attempting to increase the rate of data curation by separating curation tasks among different groups. Once curation is done, the information is shared among all. However, regardless of any cooperation between data curation sites, or partitioning of resources, there will always be some data overlap or redundancy among them. MiMI does not attempt to find new data to curate, but augment known information by highlighting redundancy and contradictions. Additionally IMEx itself is in the very infancy of data exchange, and there is as yet no cohesive, united and deeply merged dataset produced by it. The following is a brief description of the underlying concepts of MiMI, as well as a

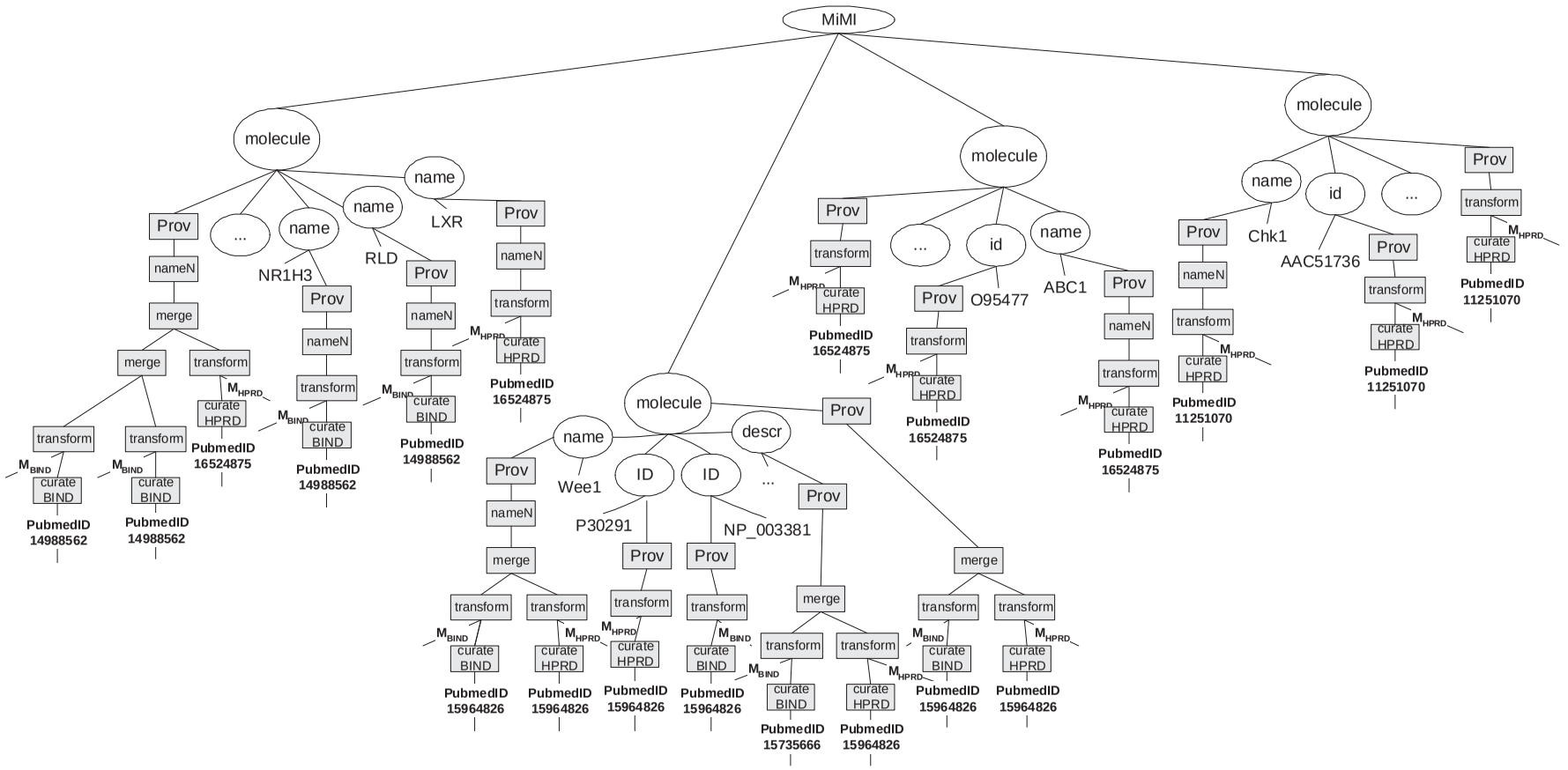


Figure 1.1: A snapshot of MiMI. The information used to create this small version of MiMI can be found in Figures 1.2–1.3. The underlying workflow is shown in Figure 1.4. Note that data nodes are round and provenance nodes are rectangular.

```
<HPRD>
   <protein>
      <name>Wee1</name>
      <ref>P30291</ref>
      <descr>tyrosine kinase</descr>
      <PubMedID>15964826</PubMedID>
   </protein>
   <protein>
      <name>ABC1</name>
      <ref>O95477</ref>
      <descr>ATP binding cassette 1</descr>
      <molFunct>sterol transporter activity</molFunct>
      <PubMedID>16524875</PubMedID>
   </protein>
   <protein>
      <name>LXR</name>
      <ref>Q13133</ref>
      <descr>liver-X-receptor</descr>
      <seq>mslw</seq>
      <PubMedID>16524875</PubMedID>
   </protein>
   <protein>
      <name>Chk1</name>
      <ref>AAC51736</ref>
      <descr>cell cycle checkpoint kinase</descr>
      <PubMedID>11251070</PubMedID>
   </protein>
</HPRD>
```

Figure 1.2: The information from HPRD used to create the snapshot of MiMI.

```
<BIND>
   <molecule>
      <name>WEE1</name>
      <extid>NP_003381</extid>
      <function>protein kinase activity</function>
      <article>15964826</article>
   </molecule>
   <molecule>
      <name>RLD</name>
      <extid>NP_005684</extid>
      <function>nuclear receptor</function>
      <article>15604093</article>
      <seq>mslw</seq>
      <seq>msiw</seq>
   </molecule>
   <molecule>
      <name>NR1H3</name>
      <extid>gi:5031893</extid>
      <function>liver receptor</function>
      <article>15680331</article>
   </molecule>
</BIND>
```

Figure 1.3: The information from BIND used to create the snapshot of MiMI.

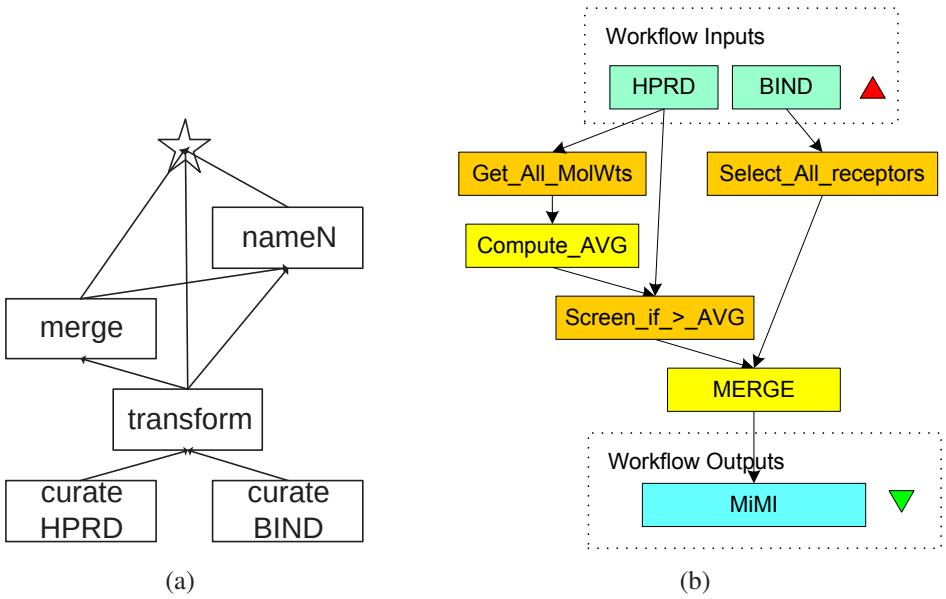


Figure 1.4: (a) The programmatic flow used to create MiMI; a representation of an implicit workflow. (b) A portion of the workflow used to create MiMI; an example of an explicit workflow.

---

detailed list of datasets employed by MiMI.

Database Construction

MiMI 1.0 uses XML as its data model. XML is the current lingua franca of biological data exchange, and gives the MiMI data model the flexibility to change as biological understanding increases. The physical storage of MiMI is built upon Timber [81, 113], a native XML database. MiMI is a component of the NIH's National Center for Integrative Biomedical Informatics (http://www.ncibi.org), and is publicly available at: http://mimi.ncibi.org.

Data Sets MiMI 1.0 has 117,549 molecules and 256,757 interactions, and is the result of integrating BIND [7], DIP [146], BioGRID [128], HPRD [114] and IntAct [78] as well as datasets from Center for Cancer Systems Biology at Harvard [77] and the Max Delbrueck Center [130]. Additionally, supplementary protein information was integrated from: GO [69], InterPro [103], IPI [84], miBLAST [85], OrganelleDB [142], OrthoMCL [39], PFam

[60] and ProtoNet [119].

Identity The issue of identity is determining when two database entries refer to the same real-world object. For instance, to a human, it is obvious that an Hsp10p molecule found in yeast and listed in DIP is the same as the Hsp10p molecule found in yeast and listed in BIND. In the simplest case, identity is defined by the uniqueness of a key attribute (set); in this case, the name. However, in protein identification, no key exists across all datasets, necessitating keyless identity functions.

For example, in BioGRID, there is an entry for HSP10, and an external reference to NCBI's RefSeq NP_014663. In BIND, there is an entry for Hsp10, with an external reference to NCBI's GI 6324594. From a human perspective, it is obvious these are the same proteins with different capitalizations. However, there is no linking identifier in either BioGRID or BIND. Further searching reveals that NCBI's records hold a link between this GI and RefSeq. Given less obviously matching names for this protein, such as CPN10 (BioGRID) and Yor020p (BIND) and different external identifiers, matching identical records is not a trivial task for an individual. Combine this with the thirteen known names for this object in BIND, DIP, BioGRID and IntAct, and the human user is bound to miss incorporating some relevant data. MiMI utilizes keyless identity functions to determine which proteins represent the same real-world object. Once this identity has been determined, the entries are deep-merged.

Deep Merge Data sets frequently have overlapping, and sometimes even contradictory information content. Our goal is to fuse information from multiple sources, even when these sources have overlapping or contradictory information, and present a cohesive result to the user. This process is called deep merging or deep integration. (In contrast, shallow integration performs just the schema translations and groups the datasets together.) To appreciate the issues involved, let us consider an example.

Example 1. *Figure 1.5 shows a brief look at some of the entries for Hsp10. Each database*

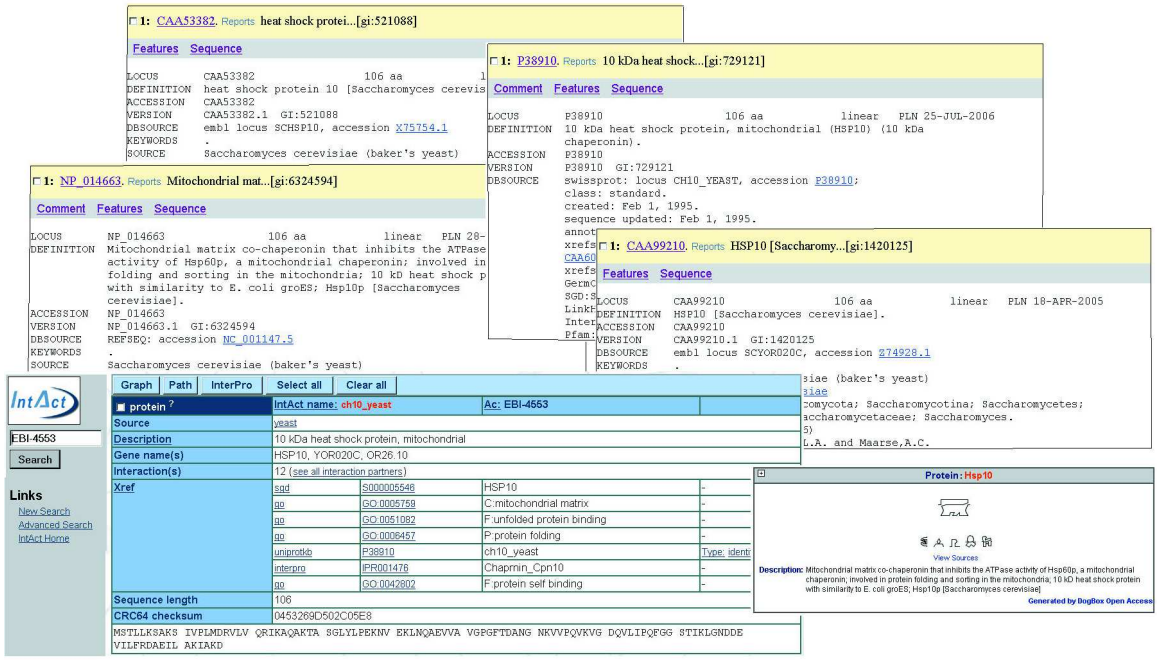


Figure 1.5: Sample protein data for Hsp10 from Intact, NCBI and BIND.

---

has different identifiers and names for the molecule. BIND calls the protein Hsp10, while IntAct calls it ch10_yeast. NCBI itself has at least four versions of this protein with the exact same sequence, and different supportive information. Assuming that an appropriate identity function is found that integrates all six molecules, shallow integration would result in 15 listed interactions. However, there are only 13 non-redundant interactions reported in the datasets. A similar problem occurs for other information on the molecule such as PTMs. Figure 1.6 shows a view of the resulting deep-merging process on Hsp10.

In other words, a deep merge finds redundancies amongst all attributes, instead of just combining all attributes from redundant data items. There is significant redundancy across data sources. Table 1 shows the number of molecules and interactions provided by each source. It also shows the resulting number of molecules and interactions after a deep-merge. For molecules there is a whopping 49% redundancy rate, while 40% of the interactions are redundant across sources.

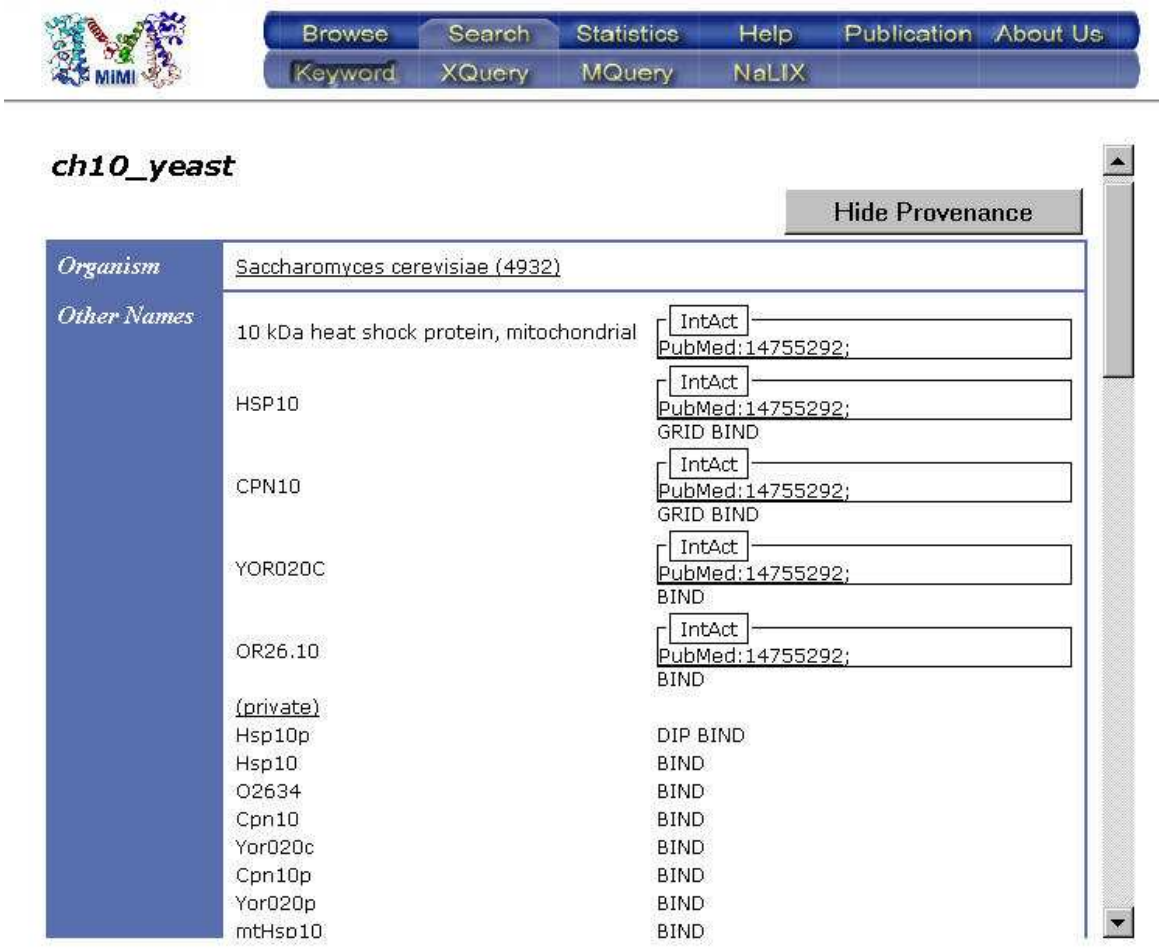


Figure 1.6: The Hsp10 information after a Deep Merge

Source	# Mols.	#Interacts.
BIND	111,394	175,678
IntAct	62,667	67,955
HPRD	18,839	66,723
GRID	15,687	53,378
DIP	19,050	54,511
Center for Cancer Systems Biology dataset	3,134	6,726
Max Delbrueck Center Dataset	1,909	3,269
MiMI	117,549	256,757

Table 1.1: Number of molecules and interactions for each source as well as total deep merged molecules and interactions in MiMI v.1.0.

1.1.2 Provenance

MiMI's complex use of external data sources requires the use of provenance to keep track of where data items came from and what processes occurred. In this work, we focus on the representation, capture, storage and usage of provenance information in hybrid systems. First, we create a basic provenance model. This provenance model is a generalization of several existing provenance structures, and is mappable to the Open Provenance Model [110], that we describe more fully in Chapter II. Provenance in this model is attached to information in MiMI. Using this model as a base, provenance capture, storage and usability is explored.

Many mechanisms [51, 73, 74, 75, 76, 105, 125] exist for capturing provenance from automatic systems such as sensor equipment [145] or workflows [3, 4, 19, 32, 52, 62, 75, 86, 94, 97, 98, 101, 125, 109, 151, 150]. For instance, the Saccharomyces Genome Database [40] uses triggers to store records of updates to the database. Unfortunately, capturing manual alterations to a dataset is not automated in any way. After exploring the set of user actions on data, we describe the minimum set operations that can describe the actions a user can make while curating data. The apparatus for capturing these actions, CPDB, is implemented with several storage strategies. Using CPDB, it is possible to understand what actions a user took, and trace the life of a piece of data. Additionally, we provide several techniques to reduce the size needed to store users actions.

Once provenance information is captured, it can grow to an inordinate size. For instance, MiMI is 270MB; its provenance store is 6GB. For provenance usage to be useful in any hybrid or workflow system, it cannot be so much larger than the data. Even with large, cheap disks, the current size of provenance can be a barrier to provenance use in hybrid systems. As such, we propose a set of reduction techniques. Utilizing properties of the provenance stores themselves, we create two families of algorithms, Factorization and Inheritance. Basic Factorization, Node Factorization, Optional Factorization, and Argument Factorization all reduce size based on repetitions within the provenance store.

Structural Inheritance and Predicate Inheritance reduce size based on properties of the data and provenance. We show that combinations of Factorization and Inheritance are possible, and provide analysis of data and provenance properties that allow system designers to choose the methods appropriate for their needs.

Unfortunately, capturing and storing traditional provenance information does not create a usable provenance store. Current provenance stores, while storing adequate information to automatically recreate a dataset, are often unable to express in a human-understandable way what has happened to the data. They contain both too much and too little information to be valuable to a human. Because the data is manipulated via a series of black boxes, it is often impossible for a human to understand what happened to the data. In this work, we highlight the missing information that can assist user understanding. Unfortunately, provenance information is already very complex and difficult for a user to comprehend, which can be exacerbated by adding the extra information needed for deeper black-box understanding. In order to alleviate this, we develop a model of provenance answers that assists the user by allowing the user to decide on the fly what information should be presented. We show the benefits of this model to users of a real scientific dataset with provenance information. Finally, we show that the structures and information needed for this model are a negligible addition.

A problem that exists in relational, workflow and hybrid provenance systems is understanding why data items are not in the result set [82]. These systems are all set up to explain "why" or "how" a data item ended up in the result set. However, while observing users, we noticed a trend of executing a query, looking at the results, and verbalizing, "Why is the tuple with attribute=<favorite test value>not in this result set"? Now increase the size of your initial dataset, and pretend you are a life-science researcher who has limited knowledge of declarative or programmatic queries. When you do not see your favorite protein after a set of manipulations, what do you do? We introduce the concept of WHY NOT? queries: the ability to ask why data items are not in the result set. We

allow researchers to specify data items they are looking for that are not in the result set, and determine which manipulation was responsible for weeding it out. In this work, we develop a model for answers to WHY NOT? queries, and describe two algorithms for finding the manipulation that discarded the data item of interest. Moreover, we work through two different methods for tracing the discarded data item that can be used with either algorithm. Using our algorithms, it is feasible for users to find the manipulation that excluded the data item of interest, and can eliminate the need for exhausting debugging.

The techniques proposed in this thesis are not limited to provenance in MiMI 1.0. They are also not limited to an XML storage model. Instead, they provide a general framework for using provenance in hybrid systems that developers can pick and choose from based on their data and provenance needs. Additionally, many of the topics discussed can also solve problems in workflow system provenance or relational provenance. For instance, increasing provenance usability by peeking into black-boxes tackles a problem that also occurs in workflow provenance systems.

1.2 Thesis Outline

Chapter II describes the data and provenance models used throughout this work. In Chapter III, a framework for tracking manually curated data is expounded. It is implemented as CPDB with several different storage mechanisms. Utilizing CPDB, it is possible to track user actions as they manually copy information from one source to another. Chapter IV focuses on reducing the size of any provenance store. Two families of algorithms are presented: Factorization and Inheritance. Three Factorization strategies and two Inheritance mechanisms are applied to several provenance stores. Additionally, it is shown how Factorization and Inheritance algorithms can be used in conjunction. In Chapter V, the focus shifts to the usability of provenance information, by creating a notion of provenance answers. Chapter VI, continues exploring the usability of provenance information by extending the provenance model to allow users to ask “Why is this NOT

in the result set?" Related work, conclusions and future work are presented in Chapters VII–VIII.

CHAPTER II

FOUNDATIONS

There is currently no standard for representing provenance, although an initial attempt is the Open Provenance Model [100, 110]. The Open Provenance Model is an attempt to have a standardized model so that provenance and tools that operate on provenance can be shared across systems in a "technology-agnostic" manner. The Open Provenance Model is does not specify a representation for provenance or syntax for machine-readable provenance. Nor does it specify protocols for storing or querying provenance. Instead, it defines a set of primary entities: Artifact, Process and Agent. An artifact is an immutable object, a process is an action or set of actions performed on artifacts that create new artifacts, while an agent can act as a catalyst on a process. However, this work is in its infancy, and has no actual representation.

Influencing the creation of the Open Provenance Model, several provenance capture systems exist [51, 73, 74, 75, 76, 105, 125], each with their own focus (actor vs data provenance), form (XML vs relational) and model. Also, several core systems have actively been used in the scientific process: Chimera [62], myGRID [74, 75], ESSW [66] and CMCS [111]. Groups from almost every scientific domain have begun to specify and collect metadata specific to their domain, such as QIS-XML [79] for Quantum Information Science. In addition, several workflow systems actively generate provenance [3, 4, 19, 32, 52, 62, 75, 86, 94, 97, 98, 101, 125, 109, 151, 150]. Others [38, 147] use a provenance model similar to the Open Provenance Model to discuss issues surrounding provenance in workflows and relational systems. From a high-level perspective, all

provenance systems have characteristics similar to the generic model we construct below.

Throughout this work, we call the basic logical data unit a *data item*. Data items may be tuples or attributes in a relational table, elements or attributes in XML, objects of arbitrary granularity in an OODB, etc. One data item may completely include, overlap with, or be totally disjoint from another data item. For example, in Figure 1.1, we show six data items: two molecule items, and their name and ID sub-items. A *dataset* is comprised of a set of data items. Datasets are often manipulated via workflows, whether explicit or implicit. A workflow is defined by an input description, output description and transformation rules. An explicit workflow is one generated by any number of workflow engines [11, 19, 101, 121]; an implicit workflow is executed by a user with a specific goal in mind, but without recording the executed processes. For example, MiMI [83] is created via an implicit workflow. A series of steps are executed, but they are neither executed within a formal workflow system, nor even fully documented. A workflow is modeled as a directed graph, where each node represents a manipulation (see Figure 1.4).

Definition 1. $Manipulation:$
A $manipulation$ *takes one or more datasets as input and produces a dataset as output.*

Thus, a manipulation is a discrete component of a workflow. An arc (m_1, m_2) in a workflow graph indicates that the output of manipulation m_1 is fed as an input to manipulation m_2. We intentionally leave the granularity of a manipulation unspecified. Depending on the user's needs and the workflow system, this can be anything from a simple function to a whole program. A query can be a manipulation or a tree of manipulations within a workflow. The workflow in Figure 1.4(a) consists of five manipulations. A few common manipulations and examples follow:

Manipulation 1. $Selection$
From an input dataset, selects a subset of data, based on some selection condition.

Example 2. *In the SDSS experiment [6], the first step is called fieldPrep. This manipulation extracts measurements of the galaxies of interest from the full dataset.*

Manipulation 2. $Translation$
Transforms the input dataset I *based on a mapping* M *and outputs dataset* I'.

Example 3. *In Figure 1.1, the input I to the translation manipulation is the HPRD dataset and M_{HPRD}, a mapping from HPRD's schema to the researcher's own. The output I' is the transformed HPRD dataset.*

The route a data item takes through a workflow is a DAG, but can be represented by a tree. If a manipulation's output is an input to two different manipulations, this route can be represented by a tree with repeated nodes, similar to query evaluation plans. When an output data item d results from an aggregation of two different input data items, its provenance record is a tree whose root element describes the aggregation step, and the two subtrees are the provenance structures associated with the two input data items. This tree is the provenance of the item d, and is shown in the "prov" subtrees in Figure 1.1. Note this is a tree-ified version of the provenance model described in the Open Provenance Model [110].

Definition 2. $Provenance\ Record$*:*
The record of input, and the manipulations applied to that input, to produce a new data item.

Definition 3. $Provenance\ Node$
A single manipulation, its input and parameters, that comprise a part of the provenance record for a data item.

Definition 4. $Provenance\ Node\ Component$
A single manipulation, input, or parameter that forms a part of a provenance node.

A provenance record is a tree of *provenance nodes*. Each node in the tree corresponds to one manipulation, and has *components* that are inputs to the manipulation. For example, in Figure 1.1, the provenance record for the ABC1 molecule is a tree of two nodes. The transform node has the component M_{HPRD}. The curate node has the parameter PubMedID 16524875. Provenance contains a record of the manipulations used, and relates processes with input and output data. The provenance model we present is generic so that it can be applied to a variety of real-world provenance stores.

It would be incorrect to substitute the original workflow for information in the provenance store. This is because the provenance record for each data item is very specific, giving the exact path that data item took through the workflow: the original source data

item it is based on, the exact parameters used in its manipulations, etc. For instance, did the transform manipulation for that data item use M_{HPRD} or M_{BIND}? The workflow is much more general, applying to all data items.

As stated above, a data item may completely include another data item e.g. a tuple can contain an attribute. Each data item may have an associated a provenance record. In Figure 1.1, the ABC1 molecule data item has a provenance record, as does the O95477 ID data item contained within it.

Definition 5. $Instance\text{-}level\ Provenance$
The provenance record associated with a particular data item in the dataset.

On the other hand, if a query was used to create the entire dataset, the query could be recorded as dataset-level provenance.

Definition 6. $Dataset\text{-}level\ Provenance$
The provenance record associated with an entire collection of data items.

Definition 7. $Provenance\ Store$
The repository of all provenance records relating to a dataset and all data items in it.

Throughout this book, we let D denote the original data store that contains N data items (which may overlap), along with their provenance records (e.g. Figure 1.1); let $size(D)$ denote the space used to store D. Each data item in D has a provenance record associated with it; so the number of provenance records is also N. Each provenance record is a tree consisting of several provenance nodes. We let n denote the total number of provenance nodes in D, where $n \geq N$.

CHAPTER III

CAPTURING PROVENANCE

Provenance does not just spring into existence, it must be captured as processes act upon data. If the appropriate provenance is not captured during execution, then it is lost forever, and can never be fully re-created. In workflow systems, the framework that organizes and executes each module automatically populates the provenance store with the necessary details. Likewise, in relational databases, the combination of query semantics and database logs can be used as provenance. However, we again encounter a hole between workflows, relational databases and hybrid database systems. In a hybrid database system, there is possibly a relational database being used. There are usually custom programs being executed and there are often manual user 'tweaks' to the data. Capturing provenance in the relational database component can utilize existing relational provenance techniques, and the capture of provenance from custom programs can be adapted from the workflow systems. However, the capture of provenance about the manual 'tweaks' is impossible in current systems.

In this chapter we study the problem of tracking provenance of scientific data in *curated databases*, databases constructed by the "sweat of the brow" of scientists who manually assimilate information from several sources. Notice that these systems, while built within relational databases, have external components, references and actions that invalidate the use of traditional database provenance, and move them into the realm of hybrid systems. First, it is important to understand the working practices and values of the scientists who maintain and use such databases.

3.0.1 Curated Databases

There are several hundred public-domain databases in the field of molecular biology [54]. Few contain raw experimental data; most represent an investment of a substantial amount of effort by individuals who have organized, interpreted or re-interpreted, and annotated data from other sources. The Uniprot [136] consortium lists upwards of seventy scientists, variously called curators or annotators, whose job it is to add to or correct the reference databases published by the consortium. At the other end of the scale there are relatively small databases managed by a single individual, such as the Nuclear Protein Database [57]. These databases are highly valued and have, in some cases, replaced paper publication as the medium of communication. Such databases are not confined to biology; they are also being developed in areas like astronomy or geology. Reference manuals, dictionaries and gazetteers that have recently moved from paper publication to electronic dissemination are also examples of curated databases.

One of the characteristics of curated databases is that much of their content has been derived or copied from other sources, often other curated databases. Most curators believe that additional record keeping is needed to record where the data comes from – its provenance. However, there are few established guidelines for what provenance information should be retained for curated databases, and little support is given by databases or surrounding technology for capturing provenance information. There has been some examination [15, 49, 88, 131] of provenance issues in data warehouses; that is, views of some underlying collection of data. But curated databases are not warehouses: they are manually constructed by highly skilled scientists, not computed automatically from existing data sets.

Example 4. *A molecular biologist is interested in how age and cholesterol efflux affect cholesterol levels and coronary artery disease. She keeps a simple database of proteins which may play a role in these systems; this database could be anything from a flat text or XML file to a full RDBMS. One day, while browsing abstracts of recent publications, she discovers some interesting proteins on SwissProt, and copies the records from a SwissProt web page into her database (Figure 3.1(a)). She then (Figure 3.1(b)) fixes the new entries*

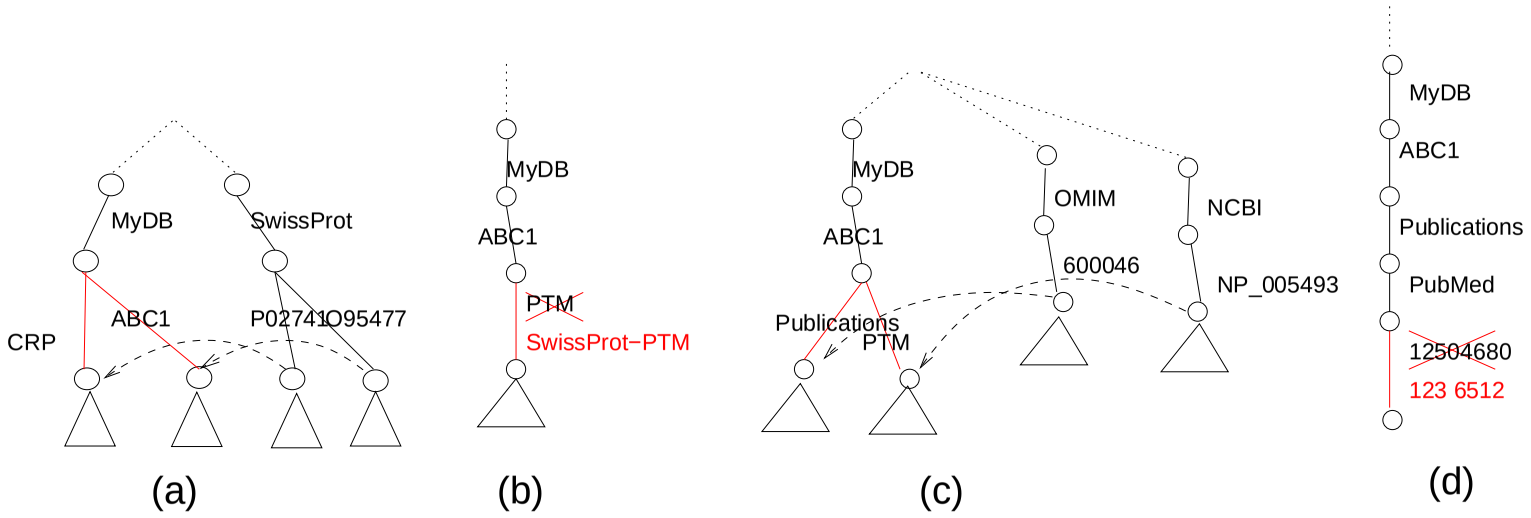


Figure 3.1: A biological database curation example. Dashed lines represent provenance links.

so that the PTM (post-translational modification) found in SwissProt is not confused with PTMs in her database found from other sites. She also (Figure 3.1(c)) copies some publication details from Online Mendelian Inheritance in Man (OMIM) and some other related data from NCBI. Finally (Figure 3.1(d)), she notices a mistake in a PubMed publication number and corrects it. This manual curation process is repeated many times as the researcher conducts her investigation.

One year later, when reviewing her information, she finds a discrepancy between two PTMs and the conditions under which they are found. Unfortunately, she cannot remember where the anomalous data came from, so cannot trace it to the source to resolve the conflict. Moreover, the databases from which the data was copied have changed; searching for the same data no longer gives the same results. The biologist may have no choice but to discard all of the anomalous data or spend a few hours tracking down the correct values. This would be especially embarrassing if the researcher had already published an article or version of her database based on the now-suspect data.

In some respects, the researcher was better off in the days of paper publication and record keeping, where there are well-defined standards for citation and some confidence that the cited data will not change. To recover these advantages for curated databases, it is necessary to retain provenance information describing the source and version history of the data.

The current approach to managing provenance in curated databases is for the database designer to augment the schema with fields to contain provenance data [7, 83] and require curators to add and maintain the provenance information themselves. Such manual bookkeeping is time consuming and seldom performed. It should not be necessary. We believe it is imperative to find ways of automating the process.

3.0.2 The problem

The term "provenance" has been used in a variety of senses in database and scientific computation research. One form of provenance is "workflow" or "coarse-grained" provenance: information describing how derived data has been calculated from raw observations [18, 62, 74, 124]. Workflow provenance is important in scientific computation, but is not a major concern in curated databases. Instead, we focus on "fine-grained" or "dataflow" provenance, which describes how data has moved through a

network of databases.

Specifically, we consider the problem of tracking and managing provenance describing the user actions involved in constructing a curated database. This includes recording both local modifications to the database (inserting, deleting, and updating data) and global operations such as copying data from external sources. Because of the large number and variety of scientific databases, a realistic solution to this problem is subject to several constraints. The databases are all maintained independently, so it is (in the short term) unrealistic to expect all of them to adopt a standard for storing and exchanging provenance. A wide variety of data models are in use, and databases have widely varying practices for identifying or locating data. While the databases are not actively uncooperative, they may change silently and past versions may not be archived. Curators employ a wide variety of application programs, computing platforms, etc., including proprietary software that cannot be changed.

In light of these considerations, we believe it is reasonable to restrict attention to a subproblem that is simple enough that some progress can be made, yet which we believe provides benefits for a common realistic situation faced by database curators. Specifically, we will address the issue of how to track the provenance of data as it enters a curated database via inserts and copies, and how it changes as a result of local updates.

3.0.3 Our approach

In this chapter, we propose and evaluate a practical approach to provenance tracking for data copied manually among databases. In our approach, we assume that the user's actions are captured as a sequence of insert, delete, copy, and paste actions by a provenance-aware application for browsing and editing databases. As the user copies, inserts, or deletes data in her local database T, provenance links are stored in an auxiliary provenance store P. These links relate data locations in T with locations in previous versions of T or in external source databases S. They can be used after the fact to review the process used to

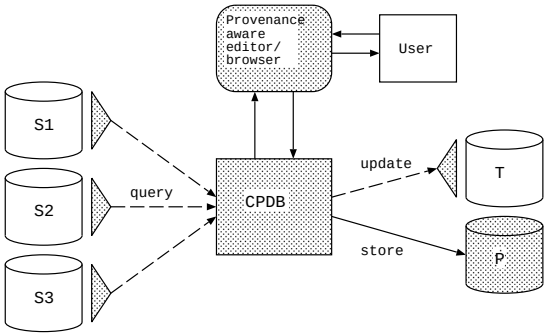


Figure 3.2: Provenance architecture

---

construct the data in T; in addition, if T is also being archived, the provenance links can provide further detail about how each version of T relates to the next.

The architecture is summarized in Figure 3.2. The new components are shaded, and the existing (and unchangeable) components are unshaded. The shaded triangles indicate *wrappers* mapping $S_1, \ldots, S_n, T$ to an XML view; the database P stores provenance information describing the updates performed by the editor. Alternatively, provenance information could be stored as annotations alongside data in T; however, this would require changing the structure of T. The only requirement we make is that there is a canonical location for every data element. We shall describe this in more detail shortly.

When provenance information is tracked manually or by a custom-built system, the user or designer typically decides what provenance information to record on a case-by-case basis. In contrast, our system records everything. The obvious concern is that the processing and storage costs for doing this could be unacceptably high. The main contribution of this chapter is to show how such fine-grained user provenance information can be tracked, stored, and queried efficiently.

We have implemented our approach and experimented with a number of ways of storing and querying provenance information, including a naïve approach and several more sophisticated techniques. Our results demonstrate that the processing overhead of the naïve approach is fairly high; it can increase the time to process each update by 28%, and the

amount of provenance information stored is proportional to the size of the changed data. In addition, we also investigated the impact of two optimization techniques: *transactional* and *hierarchical* provenance management. Together, these optimizations typically reduce the added processing cost of provenance tracking to less than 5–10% per operation and reduce the storage cost by a factor of 5–7 relative to the naïve approach; moreover, the storage overhead is bounded by the lesser of the number of update operations and the amount of data touched. In addition, typical provenance queries can be executed more efficiently on such provenance records. We believe that these results make a compelling argument for the feasibility of our approach to provenance management.

The structure of the rest of this chapter is as follows. Section 3.1 presents the conceptual foundation of our approach to provenance tracking. Section 3.2 presents the implementation of CPDB, an instance of our approach that uses the Timber XML database [81]. In Section 3.3, we present and analyze the experimental results. We conclude in Section 3.4.

3.1 Provenance and User Updates

In order to discuss provenance we need to be able to describe *where a piece of data comes from*; that is, we need to have a means for describing the location of any data element. We make two assumptions about the data, which are already used in file synchronization [65] and database archiving [24] and appear to hold for a wide variety of scientific and other databases. The first is that the database can be viewed as a tree; the second is that the edges of that tree can be labeled in such a way that a given sequence of labels occurs on at most one path from the root and therefore identifies at most one data element. Traditional hierarchical file systems are a well-known example of this kind of structure. Relational databases also can be described hierarchically. For instance, and the data values in a relational database can be addressed using four-level paths where $DB/R/tid/F$ addresses the field value F in the tuple with identifier or key tid in table R of database

DB. Scientific databases already use paths such as SwissProt/Release{20}/Q01780 to identify a specific entry, and this can be concatenated with a path such as Citation{3}/Title to identify a data element. XML data can be addressed by adding key information [24]. Note that this is a general assumption that is orthogonal to the data models in use by the various databases.

Formally, we let Σ be a set of labels, and consider *paths* $p \in \Sigma^*$ as addresses of data in trees. The trees t we consider are unordered and store data values from some domain D only at the leaves. Such trees are written as $\{a_1 : v_1, \ldots, a_n : v_n\}$, where v_i is either a subtree or data value. We write $t.p$ for the subtree of t rooted at location p.

We next describe a basic update language that captures the user's actions, and the semantics of such updates. The atomic update operations are of the form where

$$U \quad ::= \quad \mathsf{ins}\ \{a : v\}\ \mathsf{into}\ p \mid \mathsf{del}\ a\ \mathsf{from}\ p \mid \mathsf{copy}\ q\ \mathsf{into}\ p$$

The insert operation inserts an edge labeled a with value v into the subtree at p; v can be either the empty tree or a data value. The delete operation deletes an edge and its subtree. The copy operation replaces the subtree at p with a copy of the subtree at location q. We write sequences of atomic updates as $U_1; \ldots; U_n$. We write $[\![U]\!]$ for the function on trees induced by the update sequence U. The precise semantics of the operations is as follows.

$$\begin{aligned}
[\![\mathsf{ins}\ \{a : v\}\ \mathsf{into}\ p]\!](t) &= t[p := t.p \uplus \{a : v\}] \\
[\![\mathsf{del}\ a\ \mathsf{from}\ p]\!](t) &= t[p := t.p - a] \\
[\![\mathsf{copy}\ q\ \mathsf{into}\ p]\!](t) &= t[p := t.q] \\
[\![U; U']\!](t) &= [\![U']\!]([\![U]\!](t))
\end{aligned}$$

Here, $t \uplus u$ denotes the tree t with subtree u added; this fails if there are any shared edge names in t and u; $t - a$ denotes the result of deleting the node labeled a, failing if no such node exists; and $t[p := u]$ denotes the result of replacing the subtree of t at p by u, failing if path p is not present in t. Insertions, copies, and deletes can only be performed in a subtree of the target database T.

```
copy    S1/a2/y    into T/c1/y;
insert  {c2 : {}}  into T;
copy    S1/a2      into T/c2;
insert  {y : 12}   into T/c2;
insert  {c3 : {}}  into T;
copy    S1/a3      into T/c3;
copy    S2/b3/y    into T/c3;
insert  {c4 : {}}  into T;
copy    S2/b2      into T/c4;
insert  {y : 13}   into T/c4;
```

Figure 3.3: An example copy-paste update operation.

As an example, consider the update operations in Figure 3.3. These operations copy some records from S_1 and S_2, then modify some of the field values. The result of executing this update operation on database T with source databases S_1, S_2 is shown in Figure 3.4. The initial version of the target database is labeled T, while the version after the transaction is labeled T'.

3.1.1 Provenance tracking

Figure 3.4 depicts *provenance links* (dashed lines) that connect copied data in the target database with source data. Of course, these links are not visible in the actual result of the update. In our approach, these links are stored "on the side" in an auxiliary table $\mathsf{Prov}(Tid, Op, To, From)$, where Tid is a sequence number for the transaction that made the corresponding change; Op is one of I (insert), C (copy), or D (delete); $From$ is the old location (for a copy or delete), and To is the location of the new data (for an insert or copy). The To fields of deletes and $From$ fields of inserts are ignored; we assume their values are null ($\perp$). Additional information about the transaction, such as commit time and user identity can be stored in a separate table.

We shall now describe several ways of storing provenance information.

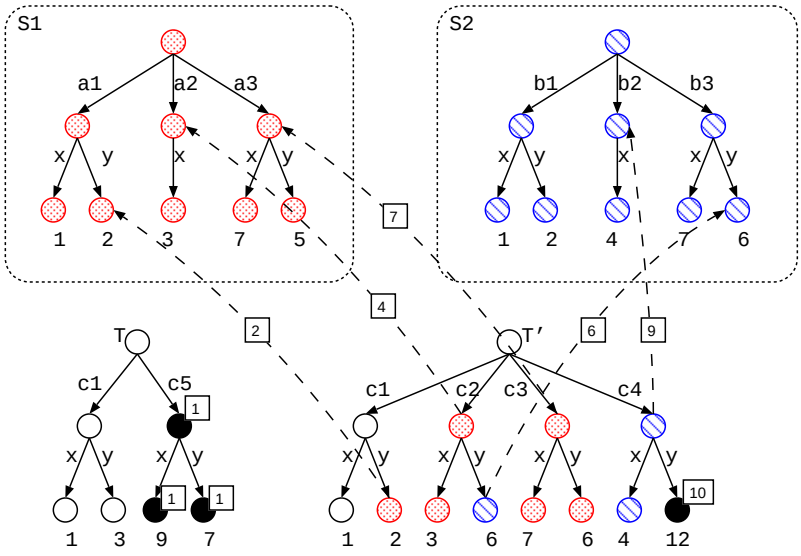


Figure 3.4: An example of executing the update in Figure 3.3. The upper two trees S_1, S_2 are XML views of source databases; the bottom trees T, T' are XML views of part of the target database at the beginning and end of the transaction. White nodes are nodes already in the target database; black nodes represent inserted nodes; other shadings indicate whether the node came from S_1 or S_2. Dashed lines indicate provenance links. Additional provenance links can be inferred from context.

Naïve provenance

The most straightforward method is to store one provenance record for each copied, inserted, or deleted node. In addition, each update operation is treated as a separate transaction. This technique may be wasteful in terms of space, because it introduces one provenance record for every node inserted, deleted, or copied throughout the update. However, it retains the maximum possible information about the user's actions. In fact, the exact update operation describing the user's sequence of actions can be recovered from the provenance table.

Transactional provenance

The second method is to assume the updated actions are grouped into transactions larger than a single operation, and to store only provenance links describing the net changes resulting from a transaction. For example, if the user copies data from S_1, then on further

(a) Prov

Tid		To	$From$
121	C	$T/c_1/y$	$S_1/a_1/y$
122	I	T/c_2	$\perp$
123	C	T/c_2	S_1/a_2
123	C	$T/c_2/x$	$S_1/a_2/x$
124	I	$T/c_2/y$	$\perp$
125	I	T/c_3	$\perp$
126	C	T/c_3	S_1/a_3
126	C	$T/c_3/x$	$S_1/a_3/x$
126	C	$T/c_3/y$	$S_1/a_3/y$
127	C	$T/c_3/y$	$S_2/b_3/y$
128	I	T/c_4	$\perp$
129	C	T/c_4	S_2/b_2
129	C	$T/c_4/x$	$S_2/b_2/x$
130	I	$T/c_4/y$	$\perp$

(b) Prov

Tid		To	$From$
121	C	$T/c_1/y$	$S_1/a_1/y$
121	C	T/c_2	S_1/a_2
121	C	$T/c_2/x$	$S_1/a_2/x$
121	I	$T/c_2/y$	$\perp$
121	C	T/c_3	S_1/a_3
121	C	$T/c_3/x$	$S_1/a_3/x$
121	C	$T/c_3/y$	$S_2/b_3/y$
121	C	T/c_4	S_2/b_2
121	C	$T/c_4/x$	$S_2/b_2/x$
121	I	$T/c_4/y$	$\perp$

(c) HProv

Tid		To	$From$
121	C	$T/c_1/y$	$S_1/a_1/y$
122	I	T/c_2	$\perp$
123	C	T/c_2	S_1/a_2
124	I	$T/c_2/y$	$\perp$
125	I	T/c_3	$\perp$
126	C	T/c_3	S_1/a_3
127	C	$T/c_3/y$	$S_2/b_3/y$
128	I	T/c_4	$\perp$
129	C	T/c_4	S_2/b_2
130	I	$T/c_4/y$	$\perp$

(d) HProv

Tid		To	$From$
121	C	$T/c_1/y$	$S_1/a_1/y$
121	C	T/c_2	S_1/a_2
121	I	$T/c_2/y$	$\perp$
121	C	$T/c_3/y$	$S_2/b_3/y$
121	C	T/c_4	S_2/b_2
121	I	$T/c_4/y$	$\perp$

Figure 3.5: The provenance tables for the update operation of Figure 3.3. (a) One transaction per line. (b) Entire update as one transaction. (c) Hierarchical version of (a). (d) Hierarchical version of (b).

reflection deletes it and uses data from S_2 instead, and finally commits, this has the same effect on provenance as if the user had only copied the data from S_2. Thus, details about intermediate states or temporary data storage in between consistent official database versions are not retained. Transactional provenance may be less precise than the naïve approach, because information about intermediate states of the database is discarded, but the user has control over what provenance information is retained, so can use shorter transactions as necessary to describe the exact construction process.

The storage cost for the provenance of a transaction is the number of nodes touched in

the input and output of the transaction. That is, the number of transactional provenance records produced by an update transaction t is $i + d + c$, where i is the number of inserted nodes in the output, d is the number of nodes deleted from the input, and c is the number of copied nodes in the output.

Hierarchical provenance

Whether or not transactional provenance is used, much of the provenance information tends to be redundant (see Figure 3.5(a,b)), since in many cases the annotation of a child node can be inferred from its parent's annotation. Accordingly, we consider a second technique, called *hierarchical provenance*. The key observation is that we do not need to store all of the provenance links explicitly, because the provenance of a child of a copied node can often be inferred from its parent's provenance using a simple rule. Thus, in hierarchical provenance we store only the provenance links that cannot be so inferred. These non-inferable links correspond to the provenance links shown in Figure 3.4. Insertions and deletions are treated as for naïve provenance, while a copy-paste operation copy p into q results in adding only a single record $\mathsf{HProv}(t, \mathsf{C}, q, p)$. Figure 3.5(c) shows the hierarchical provenance table HProv corresponding to the naïve version of Prov. In this case, the reduced table is about 25% smaller than Prov, but much larger savings are possible when entire records or subtrees are copied with little change.

Unlike transactional provenance, hereditary provenance does not lose any information and does not require any user interaction. We can define the full provenance table as a view of the hierarchical table as follows. If the provenance is specified in HProv, then it is just copied into Prov. Otherwise, the provenance of every target path p/a *not* mentioned in HProv is q/a, provided p was copied from q. If p was inserted, then we assume that p/a was also inserted; that is, children of inserted nodes are assumed to also have been inserted, unless there is a record in HProv indicating otherwise. Formally, the full provenance table

Prov can be defined in terms of HProv as the following recursive query:

$$
\begin{aligned}
\mathsf{Infer}(t,p) &\leftarrow \neg(\exists t,x,q.\mathsf{HProv}(t,x,p,q))\\
\mathsf{Prov}(t,op,p,q) &\leftarrow \mathsf{HProv}(t,op,p,q).\\
\mathsf{Prov}(t,\mathsf{C},p/a,q/a) &\leftarrow \mathsf{Prov}(t,\mathsf{C},p,q),\mathsf{Infer}(t,p).\\
\mathsf{Prov}(t,\mathsf{I},p/a,\perp) &\leftarrow \mathsf{Prov}(t,\mathsf{I},p,\perp),\mathsf{Infer}(t,p).
\end{aligned}
$$

We have to use an auxiliary table Infer to identify the nodes that have no explicit provenance in HProv, to ensure that only the provenance of the closest ancestor is used. In our implementation, Prov is calculated from HProv as necessary for paths in T, so this check is unnecessary. It is not difficult to show that an update sequence U can be described by a hierarchical provenance table with $|U|$ entries.

Transactional-hierarchical provenance

Finally, we considered the combination of transactional and hierarchical provenance techniques; there is little difficulty in combining them. Figure 3.5(d) shows the transactional-hierarchical provenance of the transaction in Figure 3.3.

It is also easy to show that the storage of transactional-hierarchical provenance is $i+d+C$, where i and d are defined as in the discussion of transactional provenance and C is the number of *roots* of copied subtrees that appear in the output. This is bounded above by both $|U|$ and $i+d+c$, so transactional-hierarchical provenance may be more concise than either approach alone.

3.1.2 Provenance queries

How can we use the machinery developed in the previous section to answer some practical questions about data? Consider some simple questions:

Src What transaction first created a node? This is particularly useful in the case of leaf data; e.g., who entered your telephone number incorrectly?

Hist What is the sequence of all transactions that copied a node to its current position?

Mod What transactions were responsible for the creation or modification of the subtree under a node? For example, you would like the complete history of some entry in a database.

Hist and Mod provide very different information. A subtree may be copied many times without being modified.

We first define some convenient views of the raw Prov table (which, of course, may also be a view derived from HProv). We define the views $\mathsf{Unch}(t,p)$, $\mathsf{Ins}(t,p)$, $\mathsf{Del}(t,p)$, and $\mathsf{Copy}(t,p,q)$, which intuitively mean "p was unchanged, inserted, deleted, or copied from q during transaction t," respectively.

$$\begin{aligned}
\mathsf{Unch}(t,p) &\leftarrow \neg(\exists x,q.\mathsf{Prov}(t,x,p,q)).\\
\mathsf{Ins}(t,p) &\leftarrow \mathsf{Prov}(t,\mathsf{I},p,\bot)\\
\mathsf{Del}(t,p) &\leftarrow \mathsf{Prov}(t,\mathsf{D},\bot,p)\\
\mathsf{Copy}(t,p,q) &\leftarrow \mathsf{Prov}(t,\mathsf{C},p,q)
\end{aligned}$$

We also consider a node p to "come from" q during transaction t (table $\mathsf{From}(t,p,q)$) provided it was either unchanged (and $p=q$) or p was copied from q.

$$\begin{aligned}
\mathsf{From}(t,p,q) &\leftarrow \mathsf{Copy}(t,p,q)\\
\mathsf{From}(t,p,p) &\leftarrow \mathsf{Unch}(t,p)
\end{aligned}$$

Next, we define a $\mathsf{Trace}(p,t,q,u)$, which says that the data at location p at the end of transaction t "came from" the data at location q at the end of transaction u.

$$\begin{aligned}
&\mathsf{Trace}(p,t,p,t).\\
\mathsf{Trace}(p,t,q,u) &\leftarrow \mathsf{Trace}(p,t,r,s),\mathsf{Trace}(r,s,q,u).\\
\mathsf{Trace}(p,t,q,t-1) &\leftarrow \mathsf{From}(t,p,q).
\end{aligned}$$

Note that Trace is essentially the reflexive, transitive closure of From. Now to define the queries mentioned at the beginning of the section, suppose that t_{now} is the last transaction

number in Prov, and define

$$\begin{aligned}
\mathsf{Src}(p) &= \{u \mid \exists q.\mathsf{Trace}(p, t_{now}, q, u), \mathsf{Ins}(u, q)\} \\
\mathsf{Hist}(p) &= \{u \mid \exists q.\mathsf{Trace}(p, t_{now}, q, u), \mathsf{Copy}(u, q)\} \\
\mathsf{Mod}(p) &= \{u \mid \exists q.p \leq q, \mathsf{Trace}(q, t_{now}, r, u), \neg\mathsf{Unch}(u, r)\}
\end{aligned}$$

That is, $\mathsf{Src}(p)$ returns the number of the transaction that inserted the node now at p, while $\mathsf{Hist}(p)$ returns all transaction numbers that were involved in copying the data now at p. Finally, $\mathsf{Mod}(p)$ returns all transaction numbers that modified some data under p. This set could then be combined with additional information about transactions to identify all users that modified the subtree at p. Here, $p \leq q$ means p is a prefix of q. Despite the fact that there may be infinitely many paths q extending p, the answer $\mathsf{Mod}(p)$ is still finite, since there are only finitely many transaction identifiers in Prov.

The point of this discussion is to show that provenance mappings relating a sequence of versions of a database can be used to answer a wide variety of queries about the evolution of the data, even without cooperation from source databases. However, if only the target database tracks provenance, the information is necessarily partial. For example, the Src query above cannot tell us anything about data that was copied from elsewhere. Similarly, the Hist and Mod queries stop following the chain of provenance of a piece of data when it exits T. If we do not assume that all the databases involved track provenance and publish it in a consistent form, many queries only have incomplete answers.

Of course, if source databases also store provenance, we can provide more complete answers by combining the provenance information of all of the databases. In addition, there are queries which only make sense if several databases track provenance, such as:

Own What is the history of "ownership" of a piece of data? That is, what sequence of databases contained the previous copies of a node?

It would be extremely useful to be able to provide answers to such queries to scientists who wish to evaluate the quality of data found in scientific databases.

3.2 Implementation

We have implemented a "copy-paste database", CPDB, that tracks the provenance of data copied from external sources to the target database. In order to demonstrate the flexibility of our approach, our system connects several different publically downloadable databases. We have chosen to use MiMI [83], a biological database of curated datasets, as our target database (T in Figure 3.2). MiMI is a protein interaction database that runs on Timber [81], a native XML database. We used OrganelleDB [142], a database of protein localization information built on MySQL, as an example of a source database. Since the target database interacts with only one source database at a time, we only experimented with one source database.

3.2.1 Overview

CPDB permits the user to connect to the external databases, copy source data into the target database, and modify the data to fit the target database's structure. The user's actions are intercepted and the resulting provenance information is recorded in a *provenance store*. Currently, CPDB provides a minimal Web interface for testing purposes. Providing a more user-friendly browsing/editing interface is important, but orthogonal to the data management issues that are our primary concern.

In order to allow the user to select pertinent information from the source and target databases, each database must be wrapped in a way that allows CPDB to extract the appropriate information. This wrapping is essentially the same as a "fully-keyed" XML view of the underlying data. In addition, the target database must also expose particular methods to allow for easy updating. Figure 3.6 describes the necessary functions that the source and target databases must implement. Essentially, the source and target databases must provide methods that map tree paths to the database's native data; in addition, the target database must be able to translate updates to the tree to updates to its internal data.

This approach does *not* require that any of the source or target databases represent

SourceDB	
treeFromDB()	Returns a tree, with unique identifiers, populated from the data. The SourceDB is responsible for determining how the data fits in the tree, e.g. mapping a relational database to tree format.
copyNode()	Returns a list of nodes that a user has copied. If the user copies a leaf node, the list is size 1. Otherwise, each node in the subtree of the selected node is contained in the list. Each node contains the identifying path and data value.
TargetDB	
addNode (String nodename)	Inserts a new, empty node with name=nodename in the target db according to the database's mapping from a tree to native format.
deleteNode()	Deletes the specified node from the target database.
pasteNode(Node X)	Insert node X as a child of the specified node according to the tree to database schema mapping.

Figure 3.6: Wrappers for Source and Target Databases

data internally as XML. Any underlying data model for which path addresses make sense can be used. Also, the databases need not expose all of their data. Instead, it is up to the databases' administrators how much data to expose for copying or updating. In many cases, the data in scientific databases consists of a "catalog" relation that contains all the raw data, together with supporting cross-reference tables. Typically, it is only this catalog that would need to be made available by a source database.

3.2.2 Implementation of provenance tracking

Given wrapped source and target databases, CPDB maintains a provenance store that allows us to track any changes made to the target database incorporating data from the

sources. To this end, during a copy-paste transaction, we write the data values to the target database, and write the provenance information to the provenance store. A user may specify any of the storage operations discussed in the previous section. In this section, we discuss how the implementations of provenance tracking and the Src, Hist, and Mod provenance queries differ from the idealized forms presented in Section 3.1.

Naïve provenance

The implementation of the naïve approach is a straightforward process of recording target and source information for every transaction that affects the target database. Whenever an insert, delete, or copy operation is performed, the corresponding function $trackInsert, trackDelete, trackPaste$ is called with the transaction identifier and applicable source and target paths. These operations simply add the corresponding records to the provenance store. Note that for a paste operation, we add one record per node in the copied subtree.

Transactional provenance

In transactional provenance, the user decides how to segment the sequence of update operations into transactions. When the user decides to end a transaction and commit its changes, CPDB stores the provenance links connecting the current version with its predecessor, and the current version becomes the next reference copy of the database, to which future provenance links will refer. Only provenance links of data actually present in the output of a transaction are stored; no links corresponding to temporary data deleted or overwritten by the transaction are stored.

To support this behavior, the transactional provenance implementation maintains an active list, $provlist$, of provenance links that will be added to the provenance store when the user commits. When an atomic update is performed, the provenance store is unaffected, but any resulting provenance links are added to the list. Conversely, in the case of a copy or delete, any provenance links on the list corresponding to overwritten or

deleted data are removed. At the time of the commit, the $commit()$ function is called, which writes the provenance of all items in the active list to the provenance store.

Hierarchical Provenance

In the hierarchical provenance storage method, we store at most one record per operation, and in particular, for a copy, we only store the record connecting the root of the copied tree to the root of the source. In addition, we check whether each link is redundant given the current provenance store, and if so, we do not add it.

Hierarchical Transactional Provenance

Combining the hierarchical and transactional provenance is straightforward; all we need to do is to maintain hierarchical provenance instead of naïve provenance records in $provlist$.

3.2.3 Provenance Queries

We implemented the provenance queries Src, Mod, and Hist as programs that issue several basic queries, due to lack of support for the kind of recursion needed by the Trace query. For naïve and transactional provenance, we can directly query the provenance store. For hierarchical provenance, the provenance store corresponds to the HProv relation. Instead of building a view containing the full provenance relation, we query the provenance store directly and compute the appropriate provenance links on-the-fly. All versions of the queries are implemented as stored procedures written in Java running in MySQL.

	Upd. Length	Trans. Length	Update Pattern	Prov. Method	Measured	Figures
1	3500	5	add, delete, copy, ac-mix, mix	N, H, T, HT	space	3.7
2	14000	5	mix, real	N, H, T, HT	space, time	3.8, 3.9, 3.10
3	14000	5	del-random, del-add, del-mix, del-copy, del-real	N, H, T, HT	space	3.11
4	3500	7, 100, 500, 1000	real	HT	time	3.12
5	14000	5	real	N, H, T, HT	query time	3.13

Table 3.1: Summary of capture experiments

3.3 Evaluation

3.3.1 Experimental setup

The evaluation of CPDB was performed on a Dell workstation with Pentium 4 CPU at 2GHz with 640MB RAM and 74.4GB disk space running Windows XP. As noted above, the target database was a 27.3MB copy of MiMI stored in Timber, and the source database was 6MB of data from OrganelleDB stored in MySQL. The provenance information was stored separately in MySQL. We used Timber version 1.1 and MySQL version 4.1.12a-nt via TCP/IP. CPDB was implemented as a Java application that communicates with MySQL via JDBC and Timber using SOAP.

We performed five sets of experiments to measure the relative performance of the naïve (N), transactional (T), hierarchical (H), and hierarchical-transactional (HT) provenance storage methods. Table 3.1 summarizes the experiments we report, including a description of the fixed and varying parameters, and listing the figures summarizing the results. We used six patterns of update operations, summarized in Table 3.2. The first five are random sequences of adds, deletes, and copies in various proportions. The copies were all of subtrees of size four (a parent with three children) from OrganelleDB to MiMI. The real update consisted of a regular pattern of copies, deletes, and inserts simulating the effect of a bulk update on MiMI that could be performed via a standard XQuery statement using XPath. It repeatedly copies a subtree into the target, then inserts three elements under the subtree root and deletes three existing subtree elements. We also used variations of the mix dataset that exhibited different deletion patterns, shown in Table 3.3.

In the first set of experiments we ran 3500-step updates on each of the first five update patterns using each storage method. For the transactional approaches, commits were performed after every five updates. In each case, we measured the amount of time needed for provenance manipulation, interaction with the target database, and interaction with the provenance database. We also measured the total size of the provenance store and target

database (both in number of rows and in real storage terms) at the end of the transaction. Efficiency considerations precluded measuring the size of the provenance store or target database after each operation.

In the second experiment, we ran 14,000-step versions of the real and mix updates using all four provenance methods, with the same experimental methodology as for the 3500-step updates. These experiments were intended to elucidate how our techniques scale as larger numbers of realistic user actions are performed, so we did not run the less realistic add, delete, or copy update patterns of this length.

Figure 3.7 shows the total provenance storage in rows needed for each method and each run for the 3500-step updates. The real storage sizes in bytes display the same trends (each row requires between 100 and 200 bytes), so we omit this data. Figure 3.8 shows the total provenance storage in rows needed for each of the 14,000-step runs. Numbers at the top of each bar show the physical sizes of the tables. Figure 3.9 shows the average time for target database interaction, and average time per add, delete, copy, or commit operation for the 14,000-mix run. These results accurately reflect observed provenance processing times in all the other experiments, so we omit this data. In order to determine how expensive provenance tracking is per add, delete, or copy operation, we also calculated the average time for dataset manipulation by operation type; Figure 3.10 shows the overhead of provenance tracking for each operation as a percentage of base dataset manipulation time.

In the third experiment, we measured the effects of deletes on provenance storage. We performed five different versions of the 14,000-mix update with varying deletion patterns. These deletion patterns may not be representative of common user behavior, but demonstrate the storage performance of the various methods under different conditions. Figure 3.11 shows the results of this experiment. We plot two columns per provenance method, one (labeled “ac”) showing the provenance table size when only the adds and copies are performed, the other (labeled “acd”) showing the size when the deletes are also performed.

add	All random adds
delete	All random deletes
copy	All random copies
ac-mix	Equal mix of random adds and copies
mix	Equal mix of random adds, deletes, copies
real	Copy one subtree, add 3 nodes, delete 3 nodes

Table 3.2: Update patterns

del-random	Paths deleted at random
del-add	All added paths deleted
del-copy	Only copies deleted
del-mix	50–50 mix of adds and copies deleted
del-real	3 nodes from copied subtree deleted

Table 3.3: Deletion patterns

The fourth experiment measured the effect of transaction length on provenance processing time. It consisted of running the 3500-real update for the hierarchical-transactional method with transaction lengths 7, 100, 500, and 1000. We measured the processing time required for each operation. Figure 3.12 summarizes the results of this experiment; it shows the average time needed for each add, delete, copy, and commit for each run. Also, the "amortized" data series shows the average time per operation with commit time amortized over all operations.

Finally, the fifth experiment measured the cost of answering some typical provenance queries. For each storage method, we measured the average query processing time for $getSrc$, $getMod$, $getHist$ queries of random locations run at the end of a 14,000-real run. Figure 3.13 shows the results. Error bars indicate the typical ranges of response times. No indexing was performed on the provenance relation, so these query times represent worst-case behavior.

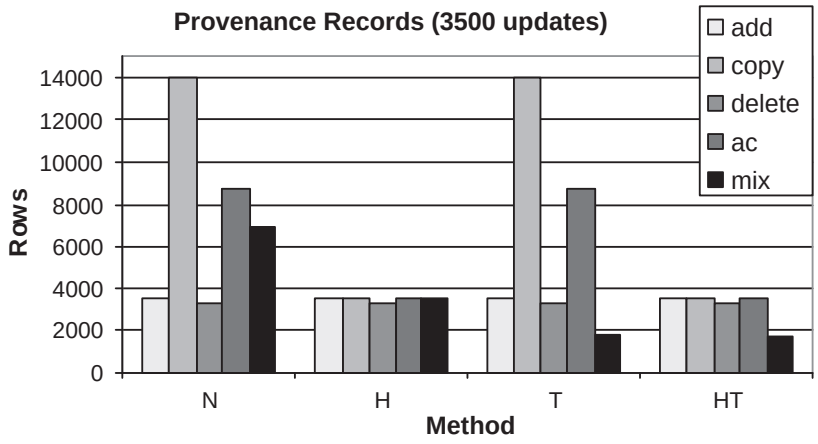


Figure 3.7: Number of entries in the provenance store after a variety of update patterns of length 3500.

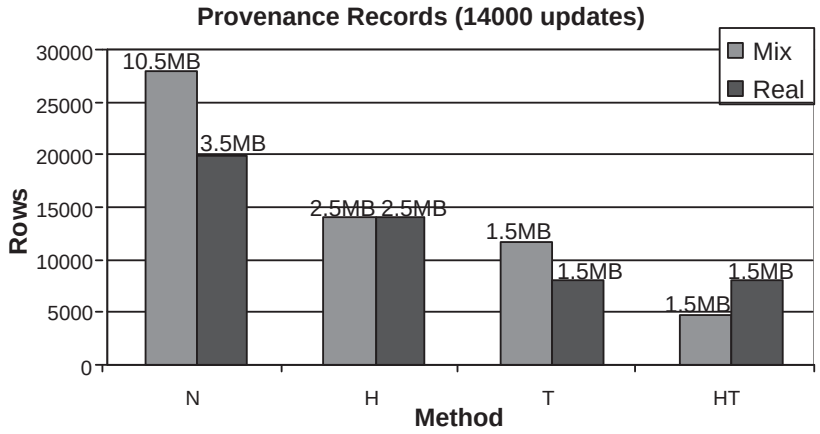


Figure 3.8: Number of entries in the provenance store after mix and real update patterns of length 14,000. The number at the top of each bar shows the physical size of the table.

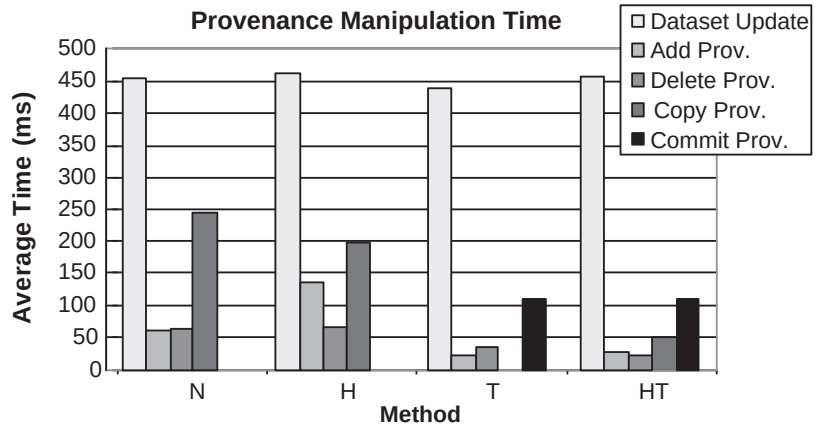


Figure 3.9: The average amount of time for target database processing and for add, delete, copy, and commit operations on the provenance store after a 14000-mix update.

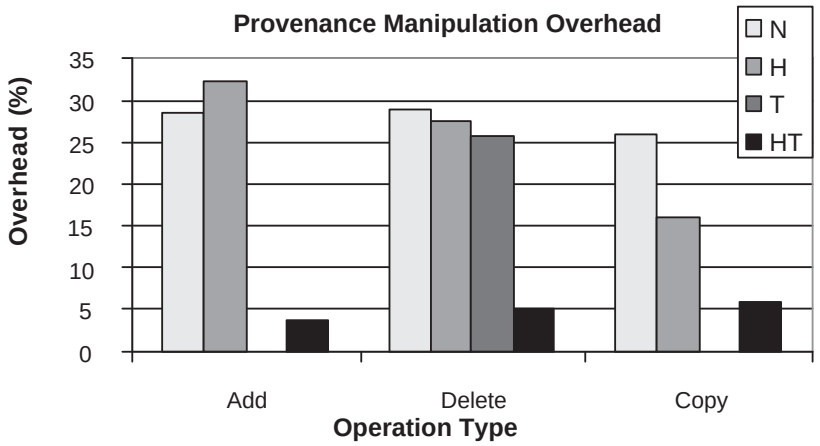


Figure 3.10: The overhead of provenance tracking per operation, as a percentage of the time to perform each basic operation.

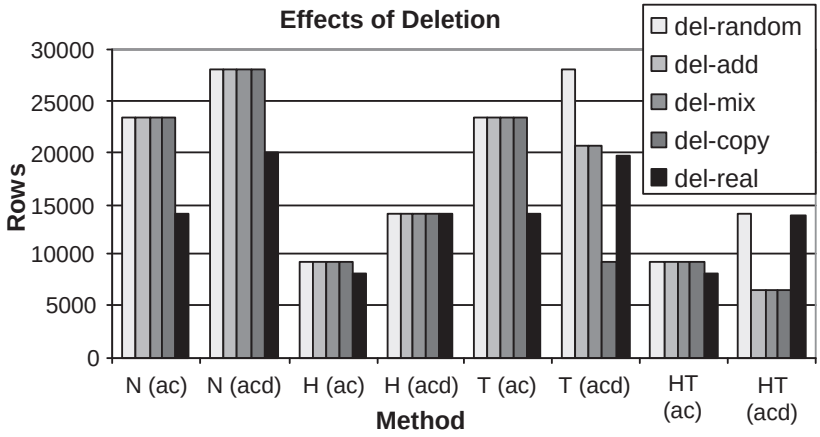


Figure 3.11: The effect of deletion on the provenance store. The notation (ac) indicates provenance table size when only add and copy operations are performed while (acd) includes deletes.

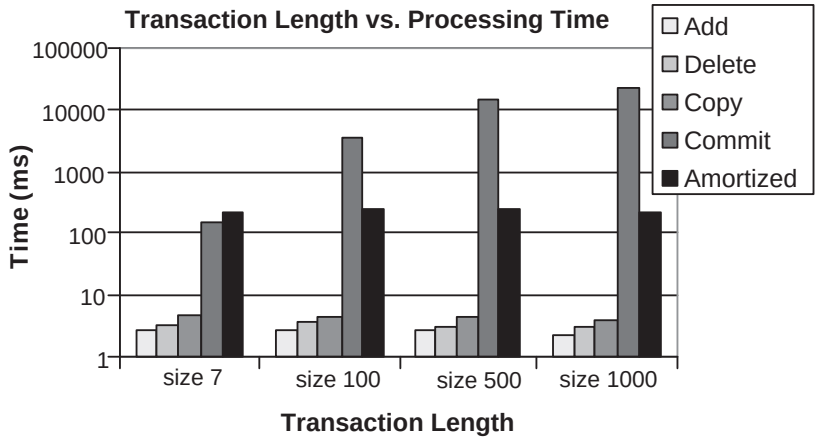


Figure 3.12: The effect of transaction size on provenance processing time.

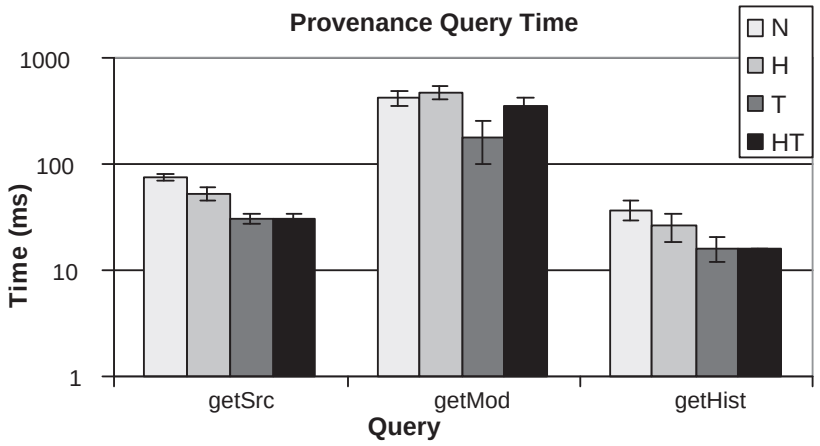


Figure 3.13: The time needed to perform basic provenance queries.

3.3.2 Analysis

As can be seen in Figures 3.7 and 3.8, either a hierarchical or transactional strategy can provide substantial space savings. Figure 3.7 shows how the storage methods perform for different types of actions. Perhaps unsurprisingly, inserts and deletes are handled essentially the same by all methods. Only copy operations really stress the system. The naïve and transactional approaches store four provenance records per copy (recall that all copies are of subtrees of size four), whereas the hierarchical techniques store only one such record per copy. The hierarchical-transactional technique provides the most efficient storage overall. The results in Figure 3.8 confirm these trends for longer sequences of updates.

Figure 3.9 shows the time spent on storing provenance information for all the techniques. For comparison, the average dataset processing time and average commit times are shown as well. Figure 3.10 depicts the average overhead of provenance processing per individual add, delete, or copy operation. For naïve storage, the add, delete and copy operations require less than 30% of the processing time needed for interaction with the target database. Although hierarchical provenance is much faster for copies, it requires more time to process inserts. (Deletes are unaffected because hierarchical

provenance treats deletes exactly as naïve provenance does.) More time is needed because we must first query the provenance database to determine whether to add the provenance record. Transactional provenance, on the other hand, is much more responsive. Inserts and copies run essentially instantaneously, because no interaction with the target database or provenance store is needed. Moreover, commits require about 25% of the average time for database interaction time, but only occur once every five steps. The savings seem to be due to the reduced number of round-trips to the provenance database. For hierarchical-transactional storage, more time is needed for copies and inserts, but all the basic operations take at most 6% of the total time. Commits take the same amount of time on average as for hierarchical provenance.

The effects of deletion are shown in Figure 3.11. For naïve and hierarchical provenance, deletion simply adds provenance records. For transactional provenance, some deletion patterns result in fewer overall records being stored, because some data is inserted and deleted in the same transaction. However, hierarchical-transactional provenance displays the most stable behavior, and stores the fewest records among the approaches for each update pattern.

The effect of transaction length on processing time is shown in Figure 3.12. Processing time per basic operation does not vary much with transaction size, while the amount of time needed to process a commit grows approximately linearly with transaction length. The average overall time per operation remains about the same. These results reflect the expected behavior, and illustrate that our approach works at interactive speeds (at most one or two seconds) for transactions of up to 100 operations. Committing the corresponding changes to the target database is likely to take as long or longer. More sophisticated techniques that minimize network round trips during commits could further reduce the overall processing time.

Finally, Figure 3.13 displays the time needed to query the various forms of provenance using the $getSrc$, $getMod$, and $getHist$ queries. In general, it is expected that $getHist$

will outperform $getSrc$, and both will do better than $getMod$ based on the provenance store access patterns and data manipulation inherent in each. The $getSrc$ and $getHist$ queries run slightly (15%) faster for hierarchical provenance, but interestingly, the $getMod$ query is about 20% slower; there is no benefit over the naïve version since each query must process all the descendants of a node, including ones not listed in the provenance store. The queries ran fastest for transactional provenance; for all three queries, we observed a speedup of roughly a factor of 2.5 relative to the naïve approach. This makes sense because transactional provenance stores only about 25–35% as many records as the naïve approach. Of course, this is because transactional provenance is less descriptive than the naïve approach; however, this seems like a reasonable tradeoff, especially since the user can decide which versions of the database to commit. Finally, hierarchical-transactional provenance benefits from the reduced number of records inherent in the transactional method, so both $getSrc$ and $getHist$ perform as well as for the transactional approach, but $getMod$ runs only slightly faster than for the naïve approach.

3.4 Conclusions

Provenance information is essential for assessing the integrity and value of data, especially in scientific databases. Because managing provenance metadata alongside ordinary data adds to the already-high cost of database curation, it is of particular concern in scientific databases that are "curated", or constructed by hand by expert users who either enter raw data or copy existing data from other sources. Therefore, automatic techniques for collecting and managing provenance in such situations would be very beneficial. However, this is a challenging problem because it requires tracking data as it is copied between databases or modified by curators.

In this chapter, we have proposed a realistic architecture for automatic provenance tracking in curated databases. We have implemented our approach and conducted an experimental evaluation of several methods of storing and managing provenance. The

most naïve approach we investigated has relatively high storage cost (storage overhead is proportional to the amount of data touched by an update), moderate processing cost (overhead of up to 30% of update processing time), and even simple provenance queries are fairly expensive to answer. However, the hierarchical-transactional technique reduced the storage overhead in our experiments reduced by around a factor of 5, while decreasing the processing overhead per update operation to at most 6% and providing improved performance on provenance queries.

These experimental results affirm that provenance can be tracked and managed efficiently using our approach. We believe that this is a promising first step towards providing powerful, general-purpose tools that will make life easier for scientific data curators and increase the reliability and transparency of the scientific record.

CHAPTER IV

EFFICIENT PROVENANCE STORAGE

Once provenance has been captured, it must be stored efficiently. Recently, several scientific endeavors have been coupled with provenance management studies. Chimera [62] has been used with physics and astronomy data; myGRID [74] with biological data; Collaboratory for Multi-Scale Chemical Science (CMCS) [111] with chemistry data; Earth System Science Workbench (ESSW) [66] with earth science data. These experiments can involve $\sim$10TB of actual base data [6]. Unfortunately, the provenance information can grow to be many times larger than the base data [6, 44, 74, 111]. Thus far, only workflow systems have created provenance stores with scientific endeavors, but hybrid systems such as MiMI have the ability to create large volumes of provenance information. This is particularly true if the provenance is fine-grained, particularly rich, or a large number of operations have been performed on each piece of data.

For instance, in a recent provenance use study [74], provenance was attached to an experiment to determine the structure of protein sequences using GRID technology [63]. Starting with sets of protein sequences, a workflow containing about 12 steps was run. The base data was about 100Kb; the provenance size was approximately 1MB, which is ten times the data size [74]. Other scientific experiments run in conjunction with provenance storage produce similar results. MiMI [83], an online protein interaction database is 270MB; its provenance store is 6GB. We also have anecdotal evidence of a real deployed scientific data system where provenance information was partially removed to reduce the storage overhead [123].

To gain an appreciation of where the enormous size of provenance comes from, consider the following small example:

Example 5. *There are many large protein interaction datasets, including HPRD [114] and BIND [7, 8, 9, 135]. Figures 1.2–1.3 show a small extract from each. A biologist may wish to integrate information from these two sources. To do this, she must first create a unified schema and transform the individual datasets into it. Then, she merges the datasets such that overlapping entries from different sources are combined. Finally, she runs each protein through a name normalizing script.*

Figure 1.4(a) depicts the workflow described above. Notice that a piece of data starts at the bottom of the workflow, and can follow any path through it depending on the data itself. Figure 1.1 depicts the resulting dataset, along with the provenance associated with each data item. Even using a small provenance record and minimal manipulations, the size of the provenance already outweighs the size of the dataset.

In this chapter, we study how to reduce the space required to store provenance. Utilizing a generic provenance model, we describe two classes of space-saving algorithms. The first is a family of algorithms that reduce the size by removing duplicate provenance records and nodes. In any series of data manipulations, patterns can be found in the provenance data. A brief glance at Figure 1.1 can elucidate this even in our small example. We propose a series of *provenance factorization* techniques that find common subtrees and manipulate them to reduce the provenance size. Finding common subtrees is a known hard problem, studied in the context of eliminating common subexpressions [43, 59, 72]. We use this work to define a "Basic Factorization" algorithm. We then develop several enhancements crucial for good size reduction in our context. We additionally develop a second set of algorithms through *provenance inheritance*. There are two distinct algorithms in this set: one based on Structural Inheritance, and one based on Predicate Inheritance. Finally, we show how both types of Inheritance can be combined with Factorization to achieve maximum savings.

We require that provenance be queriable with the base data, so that queries such as "give me all molecules that came from HPRD" can be quickly answered. Strategies such as XMill [90] lead to a representation that is not queriable. Meanwhile, XML compressors such as XGRIND [134] that facilitate querying of compressed stores do not support a

rich enough query language. In our context, it is imperative that users are able to specify relationships and joins between data and provenance information. Our methods meet this requirement, and can also be used in tandem with other XML compressors for maximum size reduction.

In Chapter II, we laid the conceptual foundations needed to describe our methods. In Sections 4.1–4.2 we outline the algorithms used to reduce provenance size using Factorization and Inheritance respectively. In Section 4.3 we discuss how Factorization and Inheritance can be combined, as well as the queriability of the compressed provenance stores. We test our reduction methods on real provenance stores, generated and stored via three distinct methods. The results of this evaluation are presented in Section 4.4. In Sections 4.3.3–4.5 we discuss incremental maintenance and our conclusions.

4.1 Provenance Factorization

Many items in a large data store may have similar or even identical provenance. If we could factor out common "sub-expressions" in the provenance of different items, these common portions could be stored just once for the whole data set rather than once for each item. We call this *provenance factorization*.

We consider Factorization at three different levels: factorization of identical provenance records, factorization of identical provenance nodes, and factorization of nodes that are identical except for their parameters.

After Factorization, the provenance records and nodes are stored in a provenance store that is separate from the data store. From each data item, there are one or more pointers to the provenance store, and in some cases, these pointers have some associated annotation.

4.1.1 Basic Factorization

Basic Factorization removes common provenance records; only one copy is stored. Each data item uses a provenance pointer to point to its provenance record. For example, in

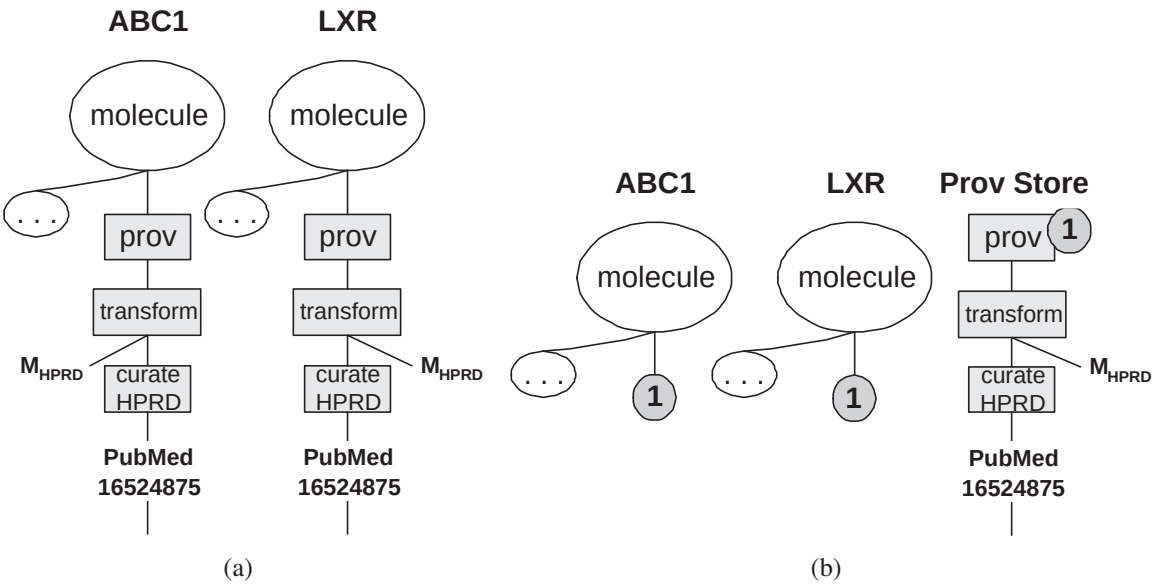


Figure 4.1: Example of Basic Factorization. (a) ABC1 and LXR molecule data items. (b) The same data items after Basic Factorization.

Figure 4.1(a), the ABC1 and LXR molecule data items are shown with their provenance. The Factorization algorithm discovers that the two provenance records are identical, and replaces each with a pointer to the record, now written separately in the provenance store, as shown in Figure 4.1(b).

The Basic Factorization Algorithm makes one pass over D, and separates the provenance records from the data items; its runtime is $O(size(D))$. When a provenance record is encountered, it is converted to a (possibly long) string; this string represents all the information in the record. The main data structure used is a hashtable on these strings; it is used to identify common provenance records. If the current provenance record R is not found in the hashtable, a copy of it is stored in the new provenance store. In the data store, the provenance record R is replaced by a pointer to its copy in the provenance store.

Since there is one provenance record per data item, the number of (not necessarily distinct) provenance records is N. Let N_1 be the number of distinct provenance records. Let S be the average size of a provenance record. The space used for storing provenance,

Reduction Technique	Estimated Provenance Size
No Reduction	$N * S$ or $n * s$
Basic Factorization	$N * p + N_1 * S$
Node Factorization	$n * p + n_1 * s$
Argument Factorization	$n * p + A * a + n_2 * s'$
Structural Inheritance	$N_2 * S$
Predicate Based Inheritance	$N * S/T$

Variables Used

N	total number of provenance records
N_1	number of distinct provenance records; $N_1 \leq N$
N_2	number of data items whose provenance record is different from that of their parent data item; $N_2 \leq N$
n	number of provenance nodes; $n \geq N$
n_1	number of distinct provenance nodes; $n_1 \leq n$
n_2	number of distinct provenance nodes, after removing the arguments; $n_2 \leq n_1 \leq n$
S	average size of a provenance record
s	average size of a provenance node
s'	average size of a provenance node without arguments; $s' \leq s$
p	size of a pointer from the data store to the provenance store
A	average size of an argument
a	number of argument annotations
T	number of data items that satisfy a predicate, and have common provenance records

Table 4.1: Estimated provenance size for each reduction technique.

before and after Basic Factorization, is shown in Table 4.1.

4.1.2 Node Factorization

Often, two data items will have distinct provenance records, but these provenance records will have many nodes in common. Node Factorization removes common provenance nodes. Only one copy of each node is stored in a separate provenance store. Provenance pointers are stored with data items to refer to these nodes.

Consider the workflow in Figure 1.4(a). Two distinct, but similar processes exist, curateHPRD and curateBIND. Consider two provenance records that contain different curation manipulations, but are otherwise identical. For instance, for provenance records

$P_0 = A \rightarrow B \rightarrow C$, and $P_1 = A \rightarrow X \rightarrow C$, the provenance store after Basic Factorization will have one record for each of P_0 and P_1. Obviously, we can do better by factoring common nodes. This amounts to combining P_0 and P_1 as $A \rightarrow (B\ OR\ X) \rightarrow C$. The pointer from the data items to the provenance store is used to indicate which of B or X is present, i.e., which of P_0 or P_1 is the correct provenance record for that data item.

In order to accomplish this reduction, we must be able to determine that a) the A nodes in P_0 and P_1 are equal, b) node B in P_0 is similar to node X in P_1, and c) the C nodes in P_0 and P_1 are equal. Provenance Node Equality and Similarity are defined as follows.

Definition 8. $Provenance\ Node\ Equality$*:*
Two provenance nodes a and b are equal, denoted $a \stackrel{P}{=} b$, iff
i. they refer to the same manipulation,
ii. all parameters and input types to the manipulation are identical.

Definition 9. $Provenance\ Node\ Specific\ Similarity$*:*
Two provenance nodes a and b are specifically similar, with respect to a similarity function S_x, if S_x(a,b) = TRUE.

Notice that similarity function values are dependent on the provenance nodes. For instance, we can define a similarity function

$S_1(a, b) = \{a.name\ like\ \text{`}curate\text{'}\ and\ b.name\ like\ \text{`}curate\text{'}\}$. In this case $S_1(curateHPRD, curateBIND) = TRUE$, but $S_1(curateHPRD, transform) = FALSE$. We write $\mathcal{S}$ for the set of acceptable similarity functions, as defined by a provenance expert familiar with the provenance store in question.

Definition 10. $Provenance\ Node\ Similarity$*:*
Two provenance nodes a and b are similar, if they are specifically similar with respect to some similarity function $S_x() \in \mathcal{S}$.

Provenance node similarity, as defined above, is a binary relation on the provenance nodes. We assume that the set $\mathcal{S}$ of similarity functions is such that this relation has the following properties.

- *Reflexive*: Each provenance node is similar to itself.
- *Symmetric*: If node a is similar to node b, then b is similar to a.

- *Transitive*: If a is similar to b, and b is similar to c, then a is similar to c.

So, provenance node similarity is an equivalence relation. It divides the set of all provenance nodes in D into equivalence classes, such that two nodes are similar iff they are in the same equivalence class. For example, consider the workflow shown in Figure 1c; there are five different kinds of manipulations. If we assume that all provenance nodes that pertain to each kind of manipulation are similar, then the similarity relation has five equivalence classes. If we further assume that all $curate_{HPRD}$ and $curate_{BIND}$ nodes are similar to each other, then the similarity relation has only four equivalence classes.

Using the above definitions, we can combine P_0 and P_1 as $A \rightarrow (B\ OR\ X) \rightarrow C$. But what happens if we change our provenance records slightly to: $P_3 = J \rightarrow K \rightarrow L \rightarrow M$ and $P_4 = J \rightarrow N \rightarrow O \rightarrow M$; we would like to combine them as $J \rightarrow (K\ OR\ N) \rightarrow (L\ OR\ O) \rightarrow M$. In other words, two provenance records could contain a long chain of similar provenance nodes. We can apply Node Factorization to such records using the following definitions.

Definition 11. $Common\ Ancestor\ Node$*:*
Two provenance nodes a *and* b *have a common ancestor node if*
i. $a.parent \stackrel{P}{=} b.parent$*, or*
ii. $a.parent$ *and* $b.parent$ *are similar, and also have a Common Ancestor Node.*

Definition 12. $Common\ Descendant\ Node$*:*
Two provenance nodes a *and* b *have a common descendant node if, for some children* c *and* d *of* a *and* b*, respectively, we have*
i. $c \stackrel{P}{=} d$*, or*
ii.c *and* d *are similar, and also have a Common Descendant Node.*

Definition 13. $Similar\ Chains$*:*
Two equal length chains C *and* C' *of provenance nodes are similar if*
i. The topmost nodes in C *and* C' *are equal,*
ii. The bottommost nodes in C *and* C' *are equal, and*
iii. the i^{th} *node in* C *and* C' *are similar,* $\forall i \neq top$ *and* $i \neq bottom$*.*

Utilizing these definitions, our Node Factorization algorithm produces a smaller provenance store. When two nodes are determined to be Similar nodes in Similar Chains, they can be merged in the Provenance Store. The equivalence class that they belonged to

and the Provenance Store now have one larger node. Moreover, because of the property of Similar Chains, the parents of these two merged nodes can also be merged and treated as one large node.

Algorithm 1: The Node Factorization with Similarity Algorithm.

```
   Input: Dataset D with Provenance Records
   Input: Similarity Functions S
   Output: Dataset with Provenance Store of Factorized Nodes
 1 Hashtable H;
 2 forall DataItems d ∈ Dataset D do
 3 |   ProvenanceRecord r = d.provenance;
 4 |   for ProvenanceNode n ← r.nextNode() do
 5 |   |   if ! H.contains( n ) then
 6 |   |   |   H.put( n, pointer );
 7 |   |   end
 8 |   |   pointer = H.get( n );
 9 |   |   writePointerInDataset( pointer );
10 |   end
11 end
12 forall ProvenanceNode n ∈ Hashtable H do
13 |   E= groupIntoEquivalenceClasses(S, H);
14 end
15 forall EquivalenceClasses E ∈ E do
16 |   forall ProvenanceNodes n ∈ E do
17 |   |   mergeAllSetsOfSimilarChains();
18 |   |   writeInProvenanceStore();
19 |   end
20 end
```

Node Factorization makes one pass over D, and runs in time $O(size(D) + e^2h)$, where e is the number of provenance nodes in an equivalence class and h is the height of the provenance trees. In our experience, $size(D)$ greatly outweighs e^2h. Algorithm 1 contains the related pseudocode. The main data structure used is a hashtable on the provenance nodes. As each provenance node is encountered in the input data file, we search for it in the hashtable. When all provenance nodes have been seen, we find similar nodes in the provenance store. If a node X is equal or similar to a node B in the provenance store, and has a common ancestor and common a descendant with B, then X and B are unioned (i.e.,

OR-ed) in the provenance store; see the example of (P_0, P_1) or (P_3, P_4) given above. We further assume the similarity functions are coarse enough such that the following holds: the number of equivalence classes is some constant determined by $\mathcal{S}$, independent of $size(D)$.

We must expand the provenance pointer to include more information. The provenance pointer used in Basic Factorization is merely a pointer to the root of a particular tree in the reduced provenance store that corresponds to the provenance record of a data item. In our example, if only the base of the branch, A, were recorded for a data item's provenance, does the provenance contain B or X? To remedy this, our provenance pointer must note which provenance nodes are being referenced.

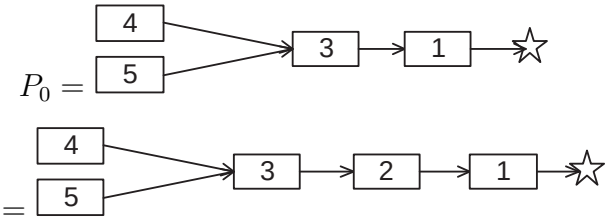
We have the following result about the content of the provenance store.

Theorem 4.1.1. *Order Invariance:*
Suppose that the set $\mathcal{S}$ of similarity functions is such that the provenance node similarity is an equivalence relation. Given a set of provenance records, the order in which they are merged into the provenance store by our Node Factorization algorithm does not affect the content of the provenance store.
Proof: *Follows from the fact that the provenance nodes are divided into equivalence classes.*

Recall from Chapter II that n denotes the number of provenance nodes in D; let s be their average size. Let n_1 be the number of distinct provenance nodes. The space used for provenance records, after Node Factorization, is shown in Table 4.1.

Factorization of Optional Nodes

Consider the two provenance records:

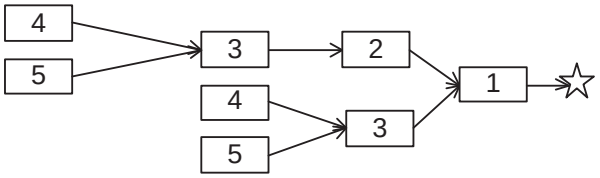


(4.1)

(4.2)

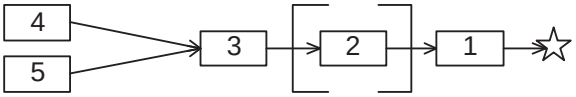
Node factorization will not combine them, because Manipulation_3 has different

parents in P_0 and P_1. This will lead to the following provenance store:



(4.3)

This provenance store is larger than it could be. Instead, we would like a much smaller provenance store as:



(4.4)

where the square brackets indicate that [Manipulation_2] is optional.

We can achieve this result by using the provenance pointer to indicate whether the optional part applies in each instance. Once this machinery is in place, we can even merge two independent paths into one longer sequence. For example, $A \rightarrow B \rightarrow C \rightarrow D$ and $A \rightarrow E \rightarrow D$ can be merged as $A \rightarrow [(B \rightarrow C)\ OR\ E] \rightarrow D$, with the provenance pointer to indicate which of $(B \rightarrow C)$ and E is present. Note that we no longer require similarity of merged nodes. In other words, $(B \rightarrow C)$ need not be similar to E.

Our algorithm for Node Factorization can be modified to also factor optional nodes; it will retain a single pass, $O(size(D))$ run time.

Unfortunately, we no longer have order invariance. Because the algorithm adds 'optional nodes' based on the parental ordering of the incoming provenance tree, and attaches them to the bottom of any other pre-existing optional nodes, the resulting provenance tree will be directly affected by the order in which we encounter the sequence of provenance nodes (e.g. $A \rightarrow [B \rightarrow C] \rightarrow [E] \rightarrow D$ is different from $A \rightarrow [E] \rightarrow [B \rightarrow C] \rightarrow D$).

4.1.3 Argument Factorization

We find that minor differences across provenance nodes can limit the utility of the Factorization algorithms discussed so far. For example, PubMedID, an input to the curateHPRD manipulation, can be different in otherwise identical provenance nodes. Because this one item is different, we no longer have a common provenance node to factor out. In Figure 4.2(a), the curateHPRD provenance nodes for the ABC1 and Chk1 molecules are identical except for the PubMedID, leading to no Basic or Node Factorization.

To permit maximum factorization of provenance under such circumstances, we consider provenance node components. We explicitly identify "arguments", and maintain them as part of the instance provenance pointer (from a data item to the provenance store) while factoring out the rest of the node. This begs the question, "What is an argument?" While the case is clear for the PubMedID in the example above, how about a parameter to a process that completely alters its execution? Rather than attempt to define the semantics of what is an argument, we say that a component is an argument if it exists in the provenance store less often than a user-specified threshold. The choice of this threshold is discussed in Section 4.4.7.

Argument Factorization involves two passes over D. The first pass uses a hashtable of provenance components; it is used to identify the arguments, by counting the number of times each component occurs. Using the provenance records in Figure 4.2(a) for example, we do a traversal of each provenance node component in each provenance record. The first component seen in this case would be PubMedID 16524875. It is placed in the hashtable. The next provenance component seen is the curateHPRD manipulation; it too is placed in the hashtable. This process continues until curateHPRD is seen again from the provenance record of Chk1. At this point, it is noted that curateHPRD, is already in the hashtable. As we continue through the rest of the the provenance nodes, we add new provenance components, and count those seen multiple times. Then, the components seen

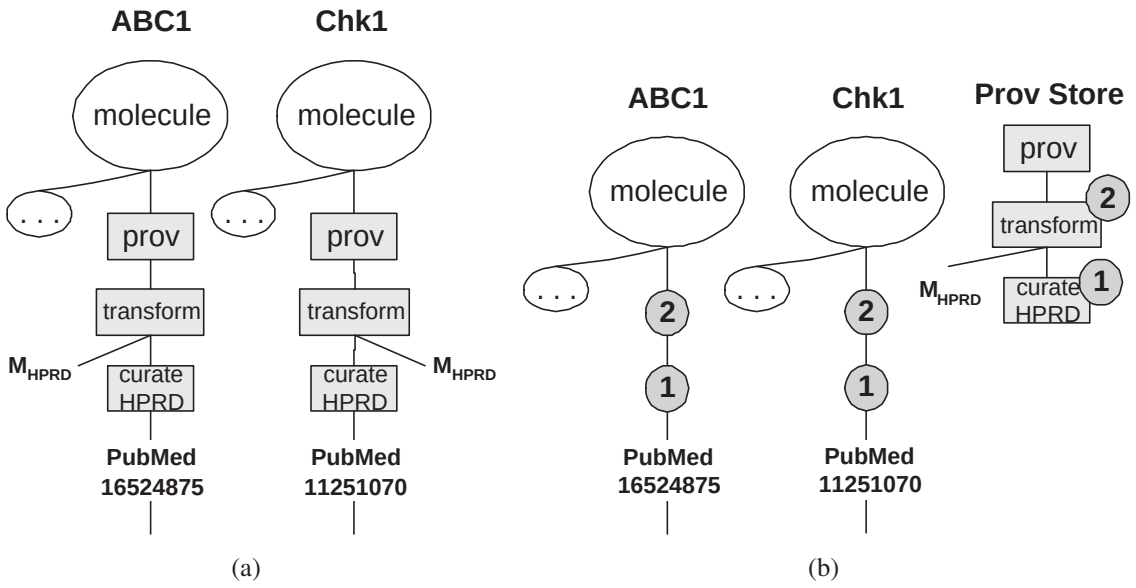


Figure 4.2: Example of Argument Factorization. (a) ABC1 and Chk1 molecule data items. (b) The same data items with provenance pointers, after Argument Factorization.

---

less often than the threshold (one in this example) are identified as arguments. The second pass is used to generate the new provenance store consisting of one copy of each distinct node sans its arguments; this process is similar to Node Factorization (Section 4.1.2). The result of these operations is shown in the provenance store of Figure 4.2(b).

Algorithm 2 contains the pseudocode for Argument Factorization. Argument Factorization makes two passes over D: one pass to place all the components into the hashtable (for determining the arguments), and one pass to factor the nodes sans their arguments. Each pass takes $O(size(D))$ time. Argument Factorization can use the same set of provenance pointers described in Section 4.1.2. The arguments are then attached to the provenance pointer. Additionally, we can make the following statements about Argument Factorization:

Theorem 4.1.2. *$Arg.\ Factorization\ Order\ Invariance$:*
The order in which provenance records are added to the provenance store using Argument Factorization does not affect the final version of the provenance store.
Proof: *Proof is straightforward since factorization depends only on the count.*

Recall that n is the number of original provenance nodes, and n_1 is the number of

Algorithm 2: The Argument Factorization Algorithm.

```
Input: Dataset D with Provenance Records
Input: Arg_Threshold
Output: Dataset with Provenance Store of Argument Factorized Nodes
Hashtable H;
forall DataItems d ∈ Dataset D do
    ProvenanceRecord r = d.provenance;
    for ProvenanceNode n ← r.nextNode() do
        for ProvenanceComponent c ← n.nextComponent() do
            if H.contains( c ) then
                H.put( c, c.getCount++ );
            else
                H.put( c, 1 );
            end
        end
    end
end
forall DataItems d ∈ Dataset D do
    ProvenanceRecord r = d.provenance;
    for ProvenanceNode n ← r.nextNode() do
        for ProvenanceComponent c ← n.nextComponent() do
            int h = H.getCount( c );
            if h ¿ Arg_Threshold then
                writePointerInDatasetToComponent;
                writeComponentInProvStore;
            else
                writeArgumentInDataset;
            end
        end
    end
end
```

distinct provenance nodes; s is their average size. Now, let n_2 be the number of distinct provenance nodes, after removing the arguments; so $n_2 \leq n_1 \leq n$. Let $s' \leq s$ be the average size of a node without arguments. Let A be the average size of an argument, and let a be the total number of argument annotations used on the pointers from the data store to the provenance store. The space used for provenance records, after Argument Factorization, is shown in Table 4.1.

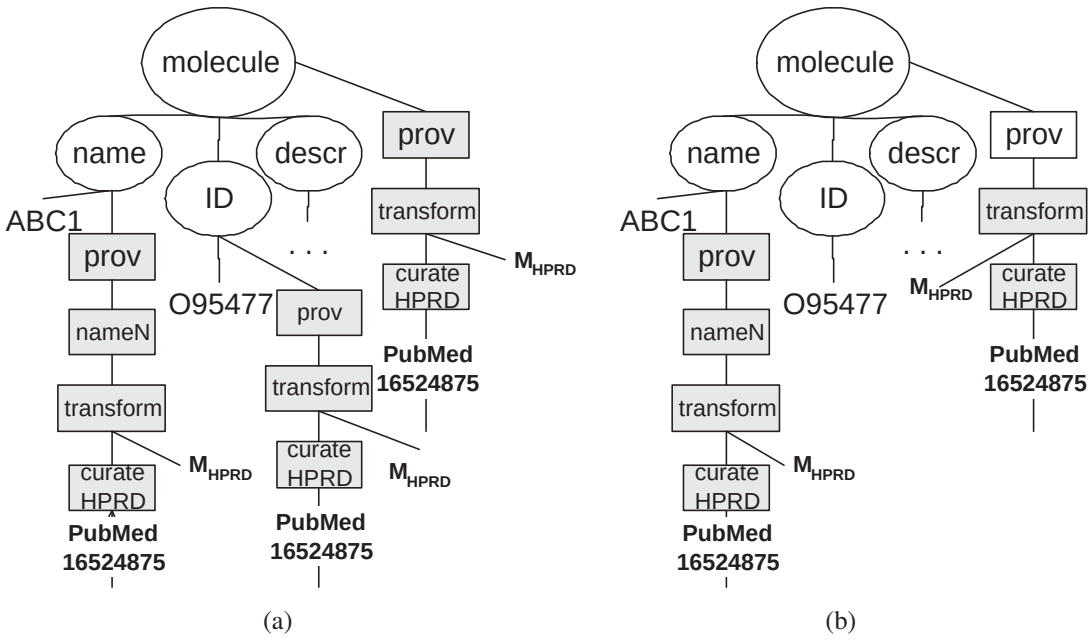


Figure 4.3: Example of Structural Inheritance. (a) The ABC1 molecule data item. (b) The same data item, after applying Structural Inheritance.

4.2 Provenance Inheritance

Provenance Factorization, discussed above, finds similarities between the steps used to derive arbitrary data items. An orthogonal optimization finds similarities in a local portion of the data tree (Structural Inheritance) or between the provenance associated with data items of a particular type (Predicate Inheritance). When provenance is inherited by an item, there is no need to record any provenance with that item; the inheritance mechanism will correctly instantiate what is required.

4.2.1 Structural Inheritance

There is often a repetition of provenance information at a fine-grained level because the same provenance is shared by data items that have a structural (parent-child or ancestor-descendant) relationship. Recall that data items can include other data items. For example, in Figure 4.3(a), the molecule data item contains the ID data item, which could

in turn contain an idType data item. The provenance is the same for both the molecule and ID data items; however, both provenance records are recorded in a full provenance store. If, instead, we only record provenance for an item when it is different from that of its parent, we can reduce the space used. On the other hand, the name data item does not have the same provenance as the molecule data item, and so cannot inherit from its parent. Figure 4.3(b) depicts the provenance records using structural inheritance.

We use a single-pass, stack-based algorithm to determine ancestor-descendant relationships and inheritance patterns; Algorithm 3 contains the pseudocode. Whenever we encounter a new data item, we compare its provenance with the provenance on top of the stack. If the two provenance records are not the same, write the provenance for the data item, otherwise write nothing. Push the provenance onto the stack. When we reach the end of a data item, pop a provenance from the stack. This one-pass algorithm takes $O(size(D))$ time.

Recall from Section 4.1.1 that the number of (not necessarily distinct) provenance records is N, and is the same as the number of data items; S is the average size of a provenance record. Let N_2 be the number of data items whose provenance record is different from that of their parent data item. The space used for provenance records, with Structural Inheritance, is listed in Table 4.1.

4.2.2 Predicate Based Inheritance

Some provenance may apply to the dataset as a whole, or to items of a certain type within it. For instance, a query can be used to create an entire dataset; then, all data items in that set would have the same provenance. If every data item in a dataset contains the same provenance record, that record can be moved from the instance-level provenance to the dataset-level provenance. For instance, in Example 2, every data item was the result of the same selection process.

More frequently, it is the case that only some of the data in the dataset is created using

Algorithm 3: The Structural Inheritance Algorithm.

Input: Root DataItem, $d \in$ Dataset D
Input: Stack S
Output: Data Item with Structurally Inherited Provenance

```
/* Note that this works through a dataset in tree form.
   If given a relational database, this method can still
   be used by mapping each data item to
   Database/table/tuple/row/ or
   Database/table/tuple/row/attribute etc.  and building
   the tree in this manner.                              */
1  ProvenanceRecord r = d.provenance;
2  ProvenanceRecord t = S.peek();
3  S.push( r );
4  if r ≠ t then
5  |  storeProvenanceWithDataItem;
6  end
7  for d ← d.nextChild() do
8  |  structInherit( d, S );
9  end
10 S.pop();
```

a global operation. For instance, for each molecule, we may introduce a new attribute molecular weight computed based on its known sequence information. We would like to store the provenance once for all of the molecular weight items in the dataset, rather than storing it once for every data item. To accomplish this, we partition the data based on the satisfaction of a boolean predicate. An example of a valid predicate would an XPath expression such as $document(\text{“}dataset\text{”})//molecule$. If the associated provenance, or a subset of the provenance, is the same for all data items that satisfy some predicate, then the common provenance can be pulled out of each data instance. It can be stored at the dataset level, together with the boolean predicate that specifies the data items to which the provenance applies.

In general, there is a tradeoff between boolean predicate complexity and the efficiency of predicate-inherited provenance. It is possible to specify a boolean predicate that specifically targets just one data item within the dataset. In this case, it would be more efficient to merely store the provenance at the instance level. On the other hand, if

the boolean predicate is not specific enough it will return too many data items and the likelihood of having a similar provenance among them is small. However, using some knowledge of the dataset, it is possible to find a set of boolean predicates that allow Predicate Based Inheritance on a large portion of the dataset. In our experiments, we use element type as the predicate. Thus, if all elements of the same name in our dataset contain nearly the same provenance, then the provenance, or subset of provenance components, can be stored at the dataset level, as shown in Figure 4.4. Note that we are agnostic about the actual schema used to represent the data set.

The Predicate Based Inheritance algorithm makes two passes over D; pseudocode can be found in Algorithms 4–5. In the first pass, we identify those provenance components that are common to all data items which satisfy a predicate; this is done for each predicate in a set of user-defined boolean predicates. If a data item d satisfies the predicate P, and no provenance information yet exists for P in the dataset-level provenance store, we create a new entry for P: It contains all the provenance components for d. If there already exists a predicate-provenance pair for P, we remove from it those components that are not in the provenance record for d. Once this first pass is completed, the provenance store will have a set of predicate-provenance pairs. A pair is present only if every data item that satisfies the predicate contains the same nonempty subset of provenance node components. A second pass over the entire dataset is then needed to write the remaining provenance that is not predicate-inherited.

Consider the runtime of our Predicate Based Inheritance algorithm. Let $Pred$ be a set of user-defined predicates that are $disjoint$ in the sense that no element can satisfy more than one predicate. Suppose that, for each element, it takes $O(t)$ time to determine which (if any) predicate in $Pred$ that element satisfies. Then the first pass takes time $O(Nt + size(D))$. The $O(size(D))$ part comes from the following: For each element $d \in D$ that satisfies a predicate $P \in Pred$, we either create a new predicate-provenance pair for P (if d is the first element seen that satisfies P), or modify the previously existing

Algorithm 4: The Predicate Inheritance Algorithm, Part I. Note: Parts I & II are separated purely for readability.

```
   Input: Dataset D with Provenance Records
   Input: Predicate List Pred
   Output: Dataset with Predicate Inherited Provenance
 1 Hashtable H;
 2 forall DataItems d ∈ Dataset D do
 3     if d satisfies P ∈ Pred then
 4         ProvenanceRecord r = d.provenance;
 5         if H.get( P ) = null then
 6             List M;
 7             for ProvenanceNode n ← r.nextNode() do
 8                 for ProvenanceComponent c ← n.nextComponent() do
 9                     M.add( c );
10                     H.put( P, M );
11                 end
12             end
13         else
14             List M = H.get( P );
15             List N;
16             for ProvenanceNode n ← r.nextNode() do
17                 for ProvenanceComponent c ← n.nextComponent() do
18                     N.add( c );
19                 end
20             end
21             forall m ∈ M ∉ N do
22                 M.remove( m );
23             end
24         end
25     end
26 end
27 goto: Algorithm 5
```

predicate-provenance pair for P. This takes time proportional to the size of the provenance record for d; over all $d \in D$, the total time is $O(size(D))$. The second pass involves, for each $d \in D$ satisfying predicate P, leaving out those components in the provenance record of d that are in the dataset level predicate-provenance pair for P. This too takes time $\sum_d O(|provrecord(d)|) = O(size(D))$.

Recall from Section 4.1.1 that N is the number of provenance records, and S is their

Algorithm 5: The Predicate Inheritance Algorithm, Part II. Note: Parts I & II are separated purely for readability.

```
   Input: Dataset D with Provenance Records
   Input: Predicate List Pred
   Input: Hashtable H
   Output: Dataset with Predicate Inherited Provenance
 1 forall DataItems d ∈ Dataset D do
 2     ProvenanceRecord r = d.provenance;
 3     if d satisfies P ∈ Pred then
 4         List M = H.get( P );
 5         if M = null then
 6             writeProvForDataItem( r );
 7         else
 8             for ProvenanceNode n ← r.nextNode() do
 9                 for ProvenanceComponent c ← n.nextComponent() do
10                     if c ∈ List M then
11                         r.remove( c );
12                     end
13                 end
14                 if !r.isEmpty() then
15                     writeProvForDataItem( r );
16                 end
17             end
18         end
19     else
20         writeProvForDataItem( r );
21     end
22 end
23 forall M ∈ Hashtable H do
24     writePredicateProv();
25 end
```

average size. Let T be the average number of provenance records that satisfy a predicate, and have the same provenance record. The space used for provenance records, using Predicate Inheritance, is shown in Table 4.1.

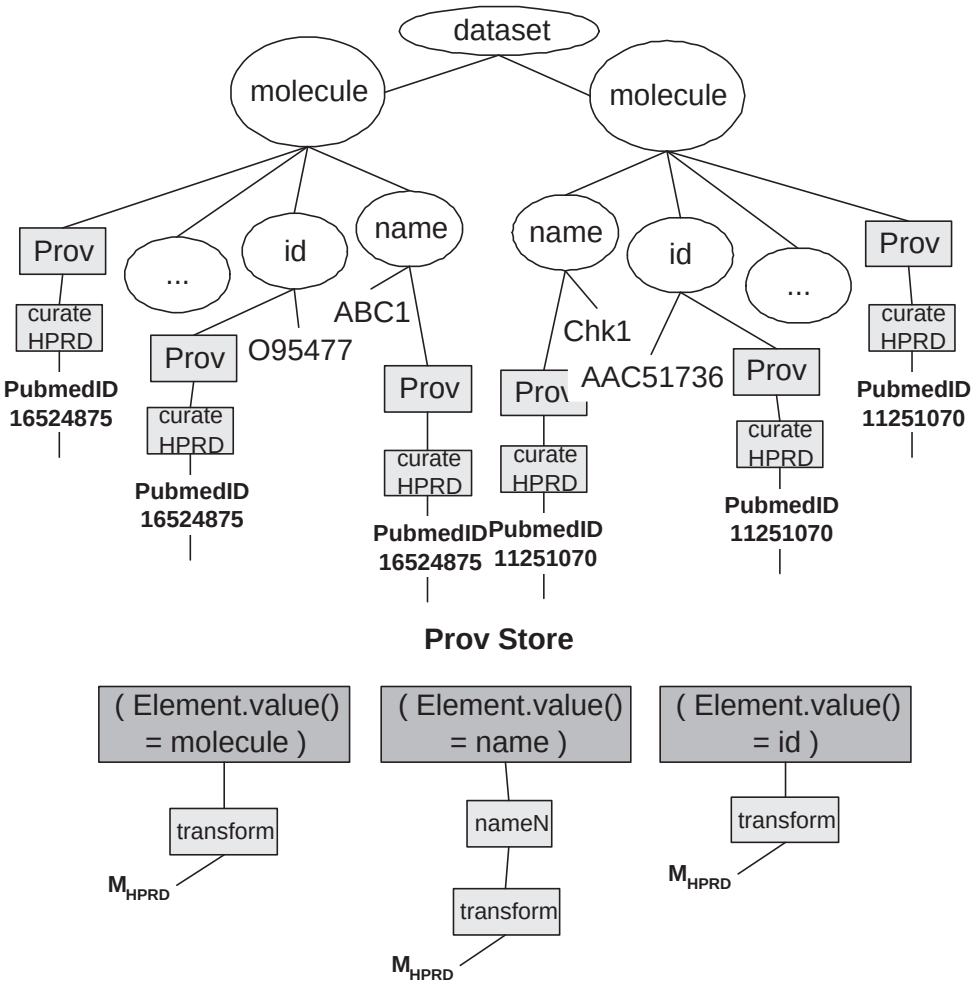


Figure 4.4: The data and provenance after applying Predicate Inheritance to ABC1 and Chk1.

4.3 Discussion

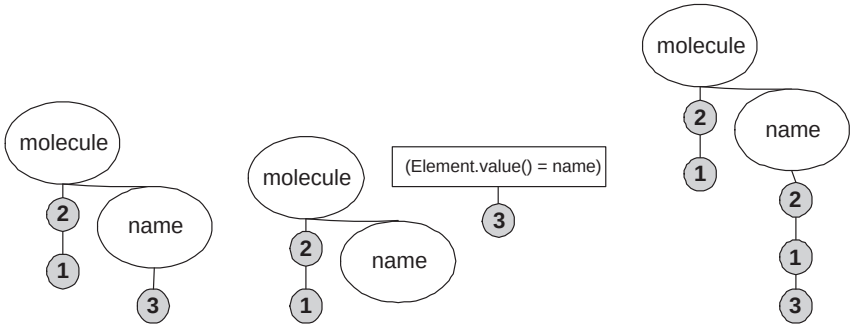
4.3.1 Combining Reduction Techniques

Any member of the Factorization Family (Basic, Node, Optional and Argument) can be applied independently to any dataset. Any member of the Factorization Family can also be used with Inheritance. Structural and Predicate Inheritance can also be combined. To apply such combinations, certain properties must be taken into account.

Using either Inheritance with any Factorization is straightforward, with two caveats: order and arguments. First, Inheritance should be performed before Factorization, since there will be fewer records to factor. Although the same correct results will occur regardless of ordering, the algorithms will run faster with Inheritance performed before Factorization. Second, provenance is not structurally inherited between data items that have the same set of manipulations but different arguments; only completely identical provenance records can be structurally inherited.

While both Structural and Predicate Inheritance can be applied individually to a dataset regardless of any Factorization usage, they can also be applied to a dataset jointly. Their conjunction is straightforward, with just a few details that should be noted. Structural Inheritance must be applied before Predicate Inheritance, as shown in Algorithm 6. Otherwise, reconstructing the provenance of a data item is potentially ambiguous. Consider the scenario:

Example 6.



6a *6b* *6c*

Consider the molecule *and* name *data items shown in 6a (grey circles are provenance nodes). If Predicate, then Structural Inheritance is applied to it, the reduced provenance will look like in 6b (assuming the provenance for the* name *data item gets moved to*

the dataset-level provenance store due to Predicate Inheritance). To re-instantiate the provenance, we would first look for Structural then Predicate Inheritance for the name *data item and produce 6c; this is clearly incorrect. Because Structural Inheritance has the requirement that the entire provenance record is either inherited or not, this situation cannot occur if Structural Inheritance is performed before Predicate Inheritance.*

Algorithm 6: The Structural and Predicate Inheritance Algorithm.

```
Input: Root DataItem, d ∈ Dataset D
Input: Predicate List Pred
Output: Dataset with Structural and Predicate Inherited Provenance
1  ProvenanceRecord r = d.provenance;
2  ProvenanceRecord t = S.peek();
3  S.push( r );
4  if r ≠ t then
5  |  runPredicateInheritance( d, Pred );
6  end
7  for d ← d.nextChild() do
8  |  structAndPredInherit( d, S );
9  end
10 S.pop();
```

Figure 4.5 shows ABC1 and Chk1 with Structural then Predicate Inheritance applied to the entire dataset. The provenance for the ABC1 name data item is found at the dataset-level (predicate based) provenance, and in the reduced provenance pointer. The provenance store size estimation formulas in Table 1 can be modified to reflect combinations of techniques.

4.3.2 Querying Provenance

There are several classes of queries that utilize provenance. Table 4.2 describes some classes, and provides a sample query for each class from the MiMI query logs. Class 1 asks for the provenance of an individual data item. Class 2 seeks the provenance for all data items of a given type. In Classes 3–4, provenance is used as a selection condition for a data item, with low and high selectivity, respectively. Finally, Class 5 performs data item joins based on provenance information. These query classes were chosen from an analysis

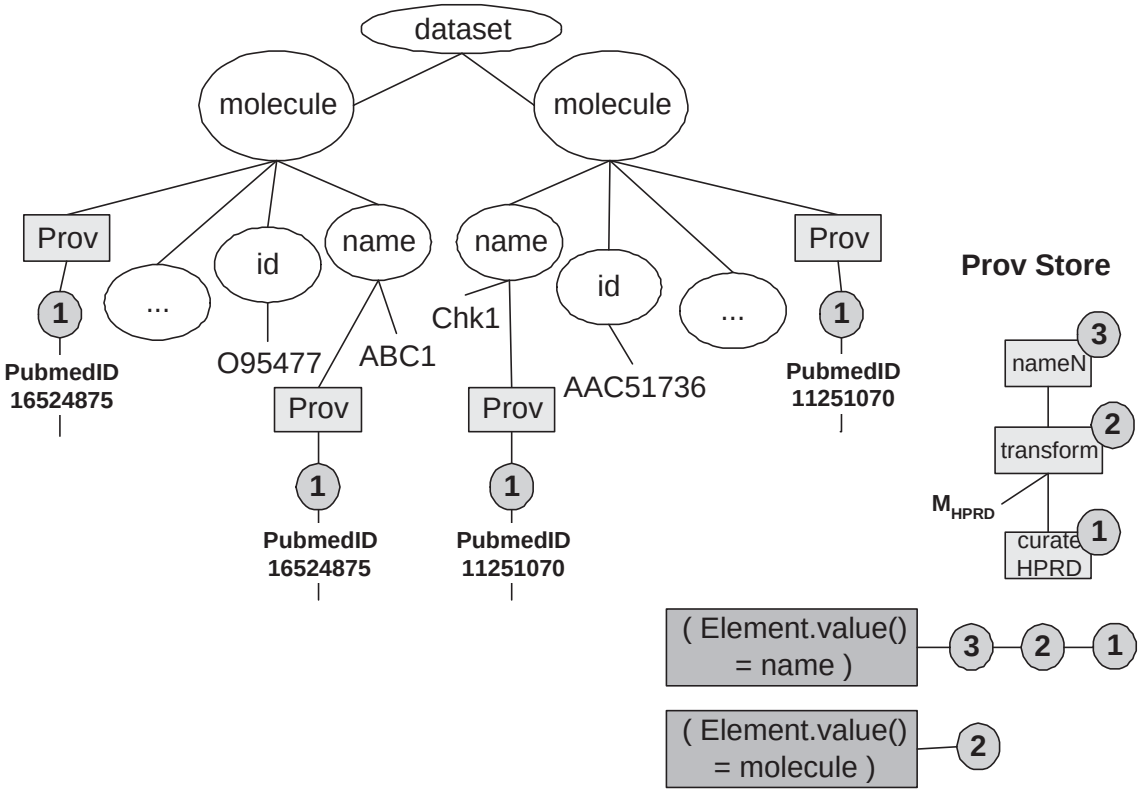


Figure 4.5: The ABC1 and Chk1 records from Figure 1.1 after Structural and Predicate Inheritance and Argument Factorization.

Class	Description	Example
1	retrieve provenance for specific data	for $b in document("MiMI")/molecule where $b/name = "ABC1" return prov($b)
2	retrieve provenance of all items of type X	for $b in document("MiMI")/molecule return prov($b)
3	use provenance as a condition (low selectivity)	for $b in document("MiMI")/molecule where prov($b) = "HPRD" return $b
4	use provenance as a condition (high selectivity)	for $b in document("MiMI")/molecule where prov($b) = "PubMedID_15964826" return $b
5	join using provenance	for $b in document("MiMI")/molecule for $n in document("MiMI")/name where prov($b) = prov($n) return $b

Table 4.2: Sample provenance queries classed by complexity.

of MiMI's query logs, and represent a mixture of interest in the data item, based on its provenance, and the provenance itself.

4.3.3 Incremental Maintenance

We have described above how to reduce the cost of storing provenance, through Factorization and Inheritance, for a static data set with static provenance. We now consider what to do if changes are made to a data set and/or its associated provenance. How does the factorized and/or inherited provenance change? Can we manage these changes using incremental algorithms, without having to analyze the entire data set, and yet achieve the same small storage space as if the static algorithm had been run? Our answer is, for the most part, positive.

There are three different types of updates that we wish to consider. The first is deletion of data. This is simple – the only case needing any attention is a possible impact if the deleted item d had children that structurally inherited provenance from it. In this case, we need to locally adjust the provenance for all children that inherited provenance from d.

The second type of update is insertion of data. For the entire family of Factorizations, the provenance associated with the new data is merged into the provenance store; only the new data and its provenance pointer(s) are written to the data store. If Structural Inheritance is used, the task is again simple – first consider the automatically inherited provenance at the newly inserted item d, and see if this is appropriate. If it is, we are done. If it is not, then we have to record the provenance with d. If d has children, then the impact of the insertion on their structurally inherited provenance must also be considered. If this has changed, then the provenance recorded at these child items has to be modified accordingly. We can encounter a slightly more complicated problem when there is a data insertion while using Predicate Inheritance. Let the new data item d satisfy a boolean predicate P that has dataset-level provenance. If the dataset-level provenance for P is a subset of d's provenance, then this is easy: we store with d only those provenance components that are not stored with P at the dataset level. However, if the dataset-level provenance for P is not a subset of d's provenance, then we must do the following: Remove from the dataset-level provenance for P those components that are not in d's provenance, and re-insert those components as a provenance pointer at every data item (except d) that satisfies P.

The third case is where there is no change to the data, but we change the provenance associated with some data item (perhaps it had been recorded incorrectly). For this, the exact same steps occur as if the data item itself changed. Additionally, the provenance store can be added to, without making any changes to the instance-level provenance pointers.

4.4 Experimental Evaluation

4.4.1 The Setup

Currently, few provenance stores exist along with datasets. Most are either destroyed after the dataset is created, never created. We were able to gain access to two very distinct styles

	Provenance Store
U	Unreduced Provenance Store
S	Structural Inheritance
P	Predicate Inheritance
SP	Structural & Predicate Inheritance
B	Basic Factorization
BS	Basic Factorization with Structural Inheritance
BP	Basic Factorization with Predicate Inheritance
BSP	Basic Factorization with Structural & Predicate Inheritance
N	Node Factorization
NS	Node Factorization with Structural Inheritance
NP	Node Factorization with Predicate Inheritance
NSP	Node Factorization with Structural & Predicate Inheritance
O	Optional Factorization
OS	Optional Factorization with Structural Inheritance
OP	Optional Factorization with Predicate Inheritance
OSP	Optional Factorization with Structural & Predicate Inheritance
A	Argument Factorization
AS	Argument Factorization with Structural Inheritance
AP	Argument Factorization with Predicate Inheritance
ASP	Argument Factorization with Structural & Predicate Inheritance

Table 4.3: Combinations of reduction techniques used in our experiments.

of provenance stores. The first style is a complex workflow used to create a synthetic data set, involving 10 processes each consuming and producing 10 data items. Provenance storage for this workflow has been studied carefully, and in fact two different provenance storage structures have been used: Karma [125] and PReServ [75]. Even though both stores represent the same base provenance, the Karma provenance store is about 300MB while PReServ is about 500MB. The second style of provenance store is from an actual large data set, MiMI [83]. The implicit workflow to create each data item comprises only a few (2-4) steps, but with a very fine-grained approach. The base data in MiMI is 270MB, while the provenance store is 6GB.

We applied various combinations of our provenance reduction techniques, as shown in Table 4.3, to each provenance store. All experiments were run on a Dell Windows XP workstation with Celeron(R) CPU at 3.06GHz with 1.96GB RAM and 122GB disk space.

The algorithms were implemented in Java, as a utility for reducing provenance storage after creation.

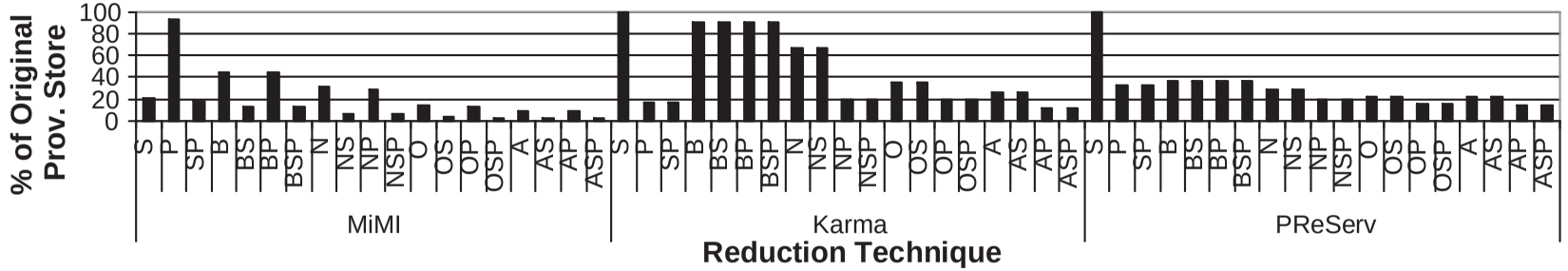
4.4.2 Storage Space

Figure 4.6(a) shows the space needed to store the provenance, according to each method; most techniques significantly reduce the size. As expected, Argument Factorization (A) does the same or better than Node (N) and Optional (O) for all the datasets. Whether Structural or Predicate Inheritance is better depends on the makeup of the dataset. MiMI has a very nested structure in which Structural Inheritance does very well. On the other hand, Karma and PReServ have flatter data unsuitable for Structural Inheritance, but use complex workflows that work well with Predicate Inheritance.

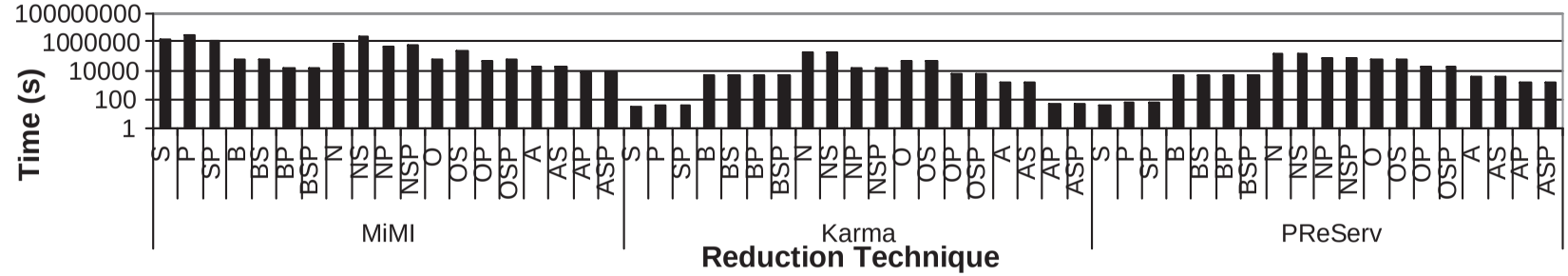
Inheritance combined with Factorization results in greater reduction for all data sets. Regardless of the Inheritance used, Argument Factorization is the clear winner. Using Argument Factorization with Structural Inheritance (AS), we produce a MiMI provenance store that is 5% the original size. Meanwhile, using Argument Factorization with Predicate Inheritance (AP) we can reduce the PReServ and Karma provenance stores to about 15% and 12%, respectively.

Because our reduction techniques are highly dependent on the data store and provenance store characteristics, we also created several artificial datasets to demonstrate each reduction technique's efficacy, based on the data and provenance characteristics; the results are shown in Figure 4.7. In Figure 4.7(a), the provenance store contained different amounts of provenance records, nodes and arguments, while the dataset and provenance store allowed contained different Structural and Predicate Inheritance characteristics. It is clear that the Factorization techniques are highly dependent on the provenance store's distribution while the Inheritance techniques vary based on the dataset and provenance store.

Using a representative sample of the more interesting techniques, as the size of the

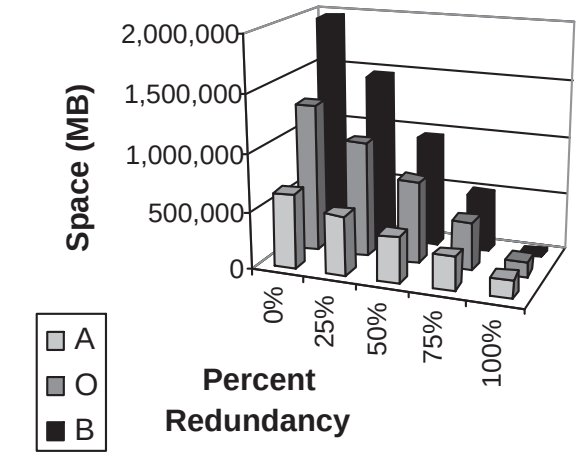


(a)

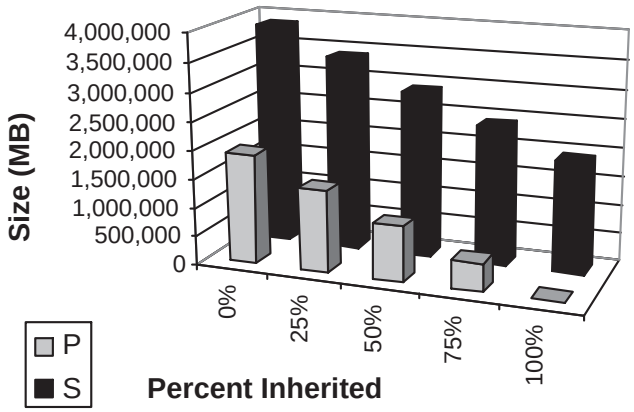


(b)

Figure 4.6: (a) Provenance storage space and (b) reduction time, for each method. See Table 4.3 for the key to letter codes.

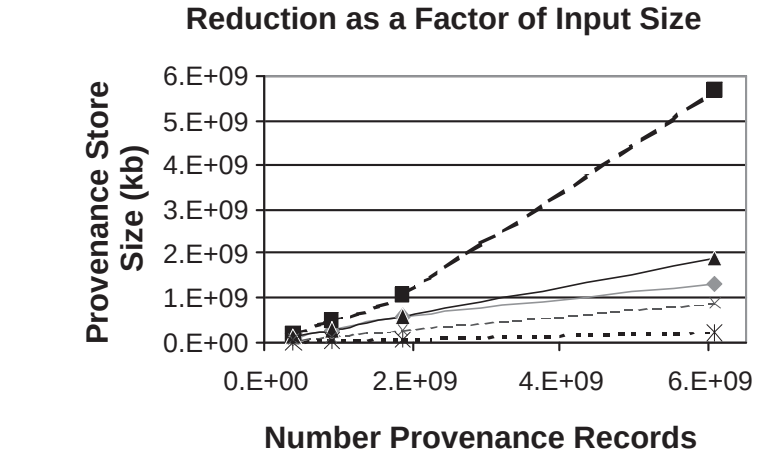


(a)

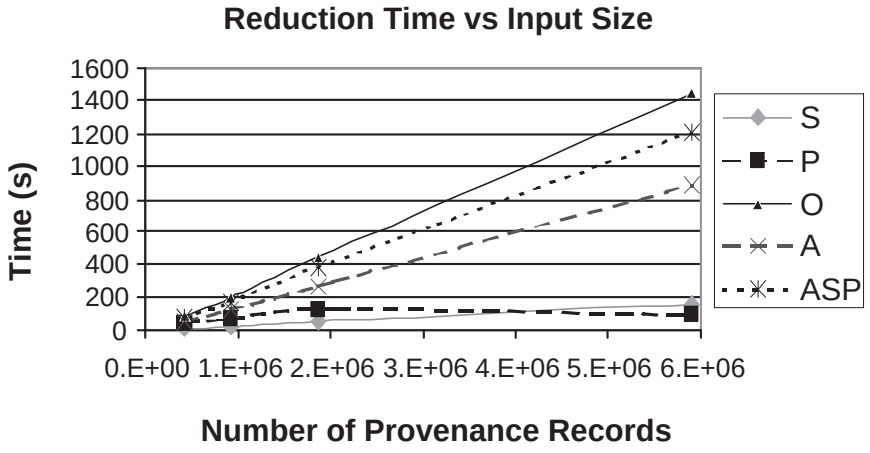


(b)

Figure 4.7: Provenance store size based on reduction technique, data and provenance characteristics. (a) Basic, Node and Argument Factorization. (b) Structural and Predicate Inheritance.



(a)



(b)

Figure 4.8: How the reduction algorithms scale based on input size in (a) space and (b) time.

provenance store grows, all our reduction algorithms remain $O(N)$, as shown in Figure 4.8.

4.4.3 Reduction Time

Figure 4.6(b) shows the reduction time for each technique. As can be seen in Figure 4.6(b), the techniques perform differently on each provenance store. Reduction time is the worst for Node Factorization; Argument Factorization and Basic Factorization are not

so bad. The reason for this is that Node Factorization maintains parental information, and will repeat the same node if it occurs in different places in the workflow, making the underlying data structures large and unwieldy. Argument Factorization has a large in memory structure to keep track of the arguments. However, because these arguments are not written, there are fewer round trips to the provenance store, thus keeping the time cost down. Karma and PReServ reduction is fast through all Factorization techniques. At first glance, it could be expected that the time to run Structural Inheritance should be less than the time to run both Structural Inheritance and Basic Factorization. However, we do not perform global Structural Inheritance then global Factorization which would make S <BS. Instead, for each data item, we test for Structural Inheritance, then immediately, reduce it via Factorization. The overall data structures are therefore smaller for BS than S, and this is reflected in the time. The reduction times presented were generated using an unoptimized implementation. Instead of reading provenance for a local tree, applying the reduction and writing it out, once the provenance structure is read in, it does not get written until the final provenance store build. In other words, as implemented, we have a large memory overhead which can be reduced by a more storage-intensive implementation. In this work, we are more concerned with the relative times between techniques.

4.4.4 Query Time

The time it takes to reduce the provenance store, and the space used to store it, are only part of the overall needs of a functioning provenance system. It is imperative that the provenance remain queriable with the data itself. Because MiMI is queriable online, we were able to obtain the query logs, and use real queries generated by biologists. In Table 4.2, we describe five classes of queries from these real queries. Each query was run five times on a cold cache and the average of the three median times is reported. The only indexes built were element tag indexes. In order to accommodate Structural Inheritance, a new iterator was created. We obtained and modified Timber [81] such that it will find the

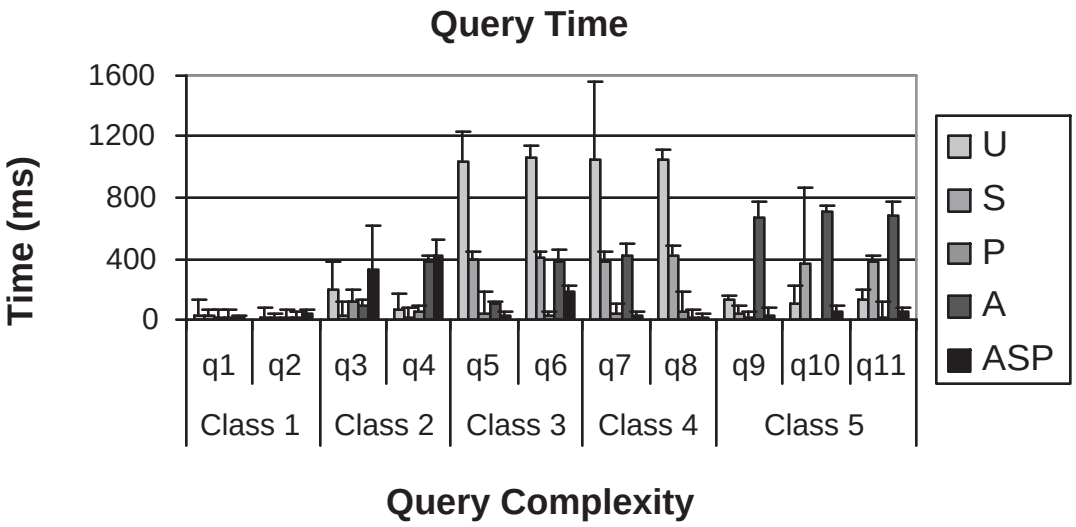


Figure 4.9: Query time for each query class on MiMI, for different reduction techniques.

provenance of a node even if it inherits from an ancestor. If the provenance is not found at a given node, the iterator returns the provenance of the parent node. Thus, this new iterator is at worst $O(h)$ time, where h is the height of the data tree. Figure 4.9 shows the query execution time for queries in different classes.

Although our reduction techniques may make the provenance representation less straightforward, they not only save space, they can also reduce query time. A look at Figure 4.9 shows some interesting trends. For Classes 1, 3 and 4, in which queries have selectivity, queries on reduced stores perform on par, or better than the original store. In particular, Classes 3 and 4, using provenance as a condition in a low and high selectivity query respectively, show how the reduced stores can out-perform the original, based on size differences. Unfortunately, Class 2 queries perform worse on the reduced store. This is because every such query requires at least one join in the reduced stores. Finally, Class 5 query times on reduced stores are mixed compared to the original store. These queries require multiple joins, and it is impossible to push provenance instantiation higher in the query plan. This leads to poor performance in some cases, although Predicate Inheritance

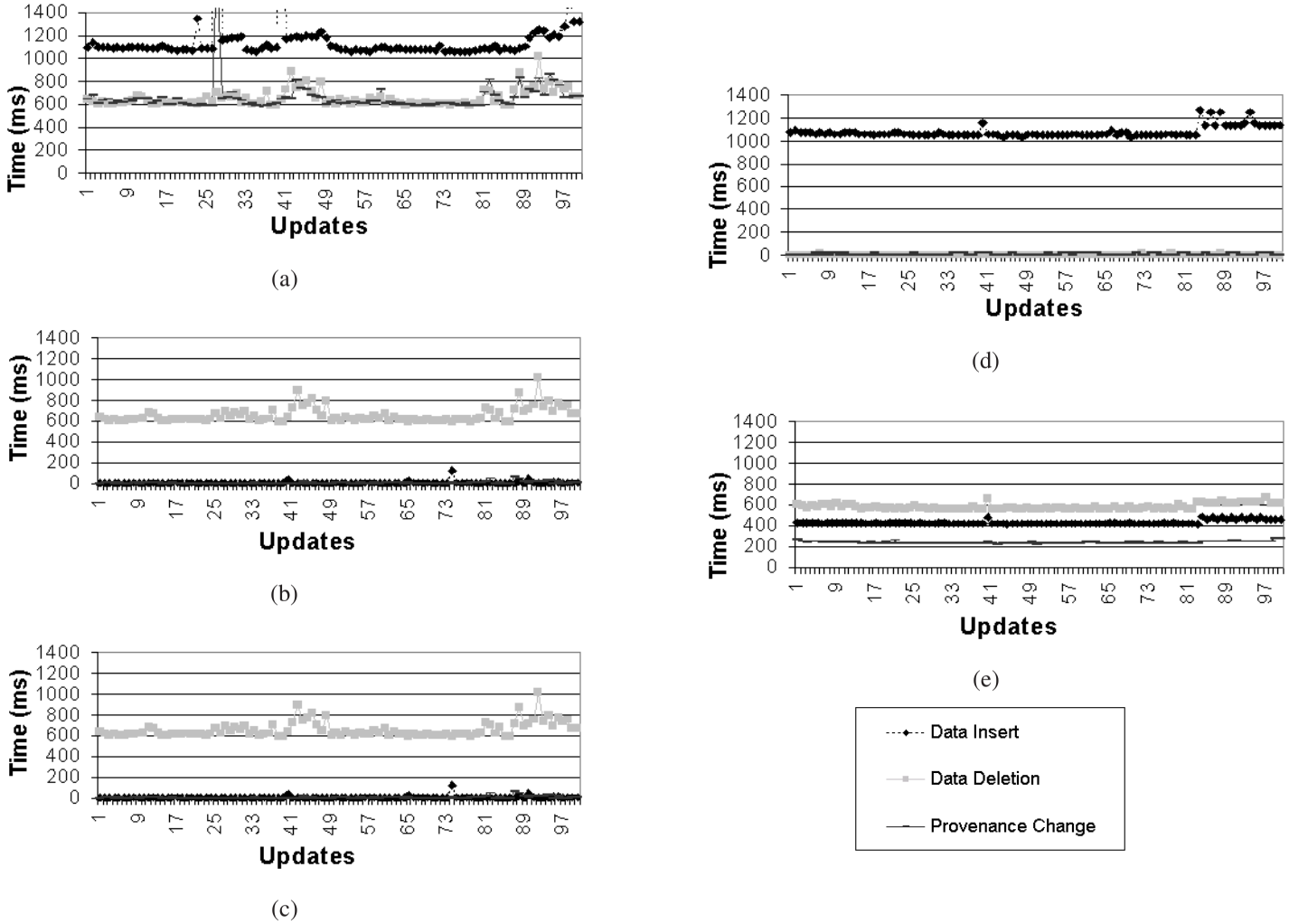


Figure 4.10: Incremental Maintenance on provenance stores with (a) Structural Inheritance, (b) Predicate Inheritance, (c) Optional Factorization, (d) Argument Factorization, and (e) Argument Factorization with Structural and Predicate Inheritance.

(P) and Argument Factorization with Structural and Predicate Inheritance (ASP) both do better than unreduced.

Structural Inheritance performs well across the board. This is due to a combination of reduced space and an $O(h)$ iterator. First, the provenance store is so reduced that the entire database is distinctly smaller. Second, no join needs to be performed, and the ancestor-lookup iterator is relatively fast. Predicate Inheritance appears all over the map in these queries. In some cases it does well, while in others it is almost the worst. Even within the same query class, it has wildly varying performance. A closer inspection of the provenance store itself contains the answer. In the case where there is a predicate-inherited item (e.g. type='name') in the provenance store, the method does very well. However, if no predicate inheritance exists for a certain element type, then the query performs poorly.

4.4.5 Incremental Maintenance

As discussed in Section 4.3.3, the reduction technique used can affect the complexity of incremental maintenance. Figure 4.10 shows how each store performs, for data insertion, data deletion and provenance changes. A random sequence of data inserts, deletions and provenance changes were performed, in equal measure, regardless of the reduction technique. For a provenance store with Structural Inheritance, Figures 4.10(a) and 4.10(e), the following inserts, deletes and provenance changes were performed: 1. insert a data item that Structurally Inherits provenance (from its parent); 2. insert a data item that does not Structurally Inherit provenance; 3. delete a data item with children that Structurally Inherit provenance from it; 4. delete a data item with no such children; 5. change provenance for a data item; with children that Structurally Inherit provenance from it; 6. change provenance for a data item with no such children. For a provenance store with Predicate Inheritance, Figures 4.10(b) and 4.10(e), the following inserts, deletes and provenance changes were performed: 1. insert a data item that Predicate Inherits provenance; 2. insert a data item that does not Predicate Inherit; 3. insert a data item that Predicate Inherits

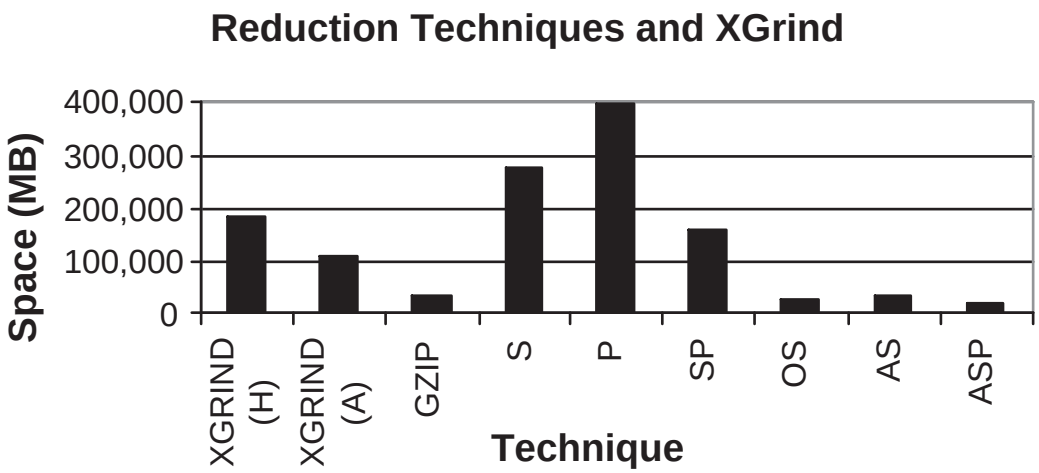


Figure 4.11: XGRIND, GZIP and a sample of Reduction Techniques applied to the MiMI Provenance Store.

provenance, but breaks the inheritance pattern for all elements of that type; 4. delete a data item; 5. change provenance for a data item that Predicate Inherits provenance; 6. change provenance for a data item that does not Predicate Inherit; 7. change provenance for a data item that Predicate Inherits, but breaks the inheritance pattern for all elements of that type. For a provenance store with just Factorization, Figures 4.10(c) and 4.10(d), the following inserts, deletes and provenance changes were performed: 1. insert a data item; 2. delete a data item; 3. change provenance for a data item.

As shown in Figure 4.10, no matter what provenance reduction technique is used, updates are easy to perform. We would like to note that using Predicate Inheritance lowers the average time for a data insert. If the data item and provenance satisfies a predicate, then there is no need to manipulate the provenance store, thus saving time. Additionally, in provenance stores using straight Factorization, deletes and provenance changes are relatively cheap since there is no need to check inheritance dependencies. The take away point here is that incremental maintenance on a reduced provenance store is cheap.

4.4.6 Interaction with Other Compressors

As previously noted, traditional XML compression techniques are not suitable for our purposes because they do not result in a provenance store that is queriable along with base data. Even techniques such as XGRIND, which support keyword and path queries [134], do not have the full associative power needed to support joins between provenance and data. However, we have applied XGRIND using Huffman (H) and Arithmetic (A) encoding to the original provenance store, and compared the compressed size with our reduced stores. Additionally, although a gzipped document is not queriable, we included the gzipped provenance store as a well known comparison point. As shown in Figure 4.11, XGRIND on the un-reduced provenance store creates a reduced store smaller than any of the Inheritance methods on their own. However, using combinations of reduction techniques it is possible to compress the provenance store smaller than XGRIND and still maintain the ability to query data and provenance together. Additionally, it is possible to combine our reduction techniques with any classic XML compressor, such as XMill [90], to get an extremely small store, as shown in Figure 4.12. While our reduction techniques do not compress as well as an XML compressor like XMill, the data in our compressed stores can be queried while the information compressed by XMill cannot. However, should the ultimate storage size be a primary concern, combining our reduction techniques with an XML compressor gives the best results.

4.4.7 Other Parameters

Figure 4.13 shows the relationship between Argument Threshold, the time to produce the reduced provenance store, and the size of the reduced store. In the case of MiMI, there is a drastic drop in the runtime between an Argument Threshold of 5 and 50. This drop is explained by the makeup of the provenance store. With a threshold of 5, there are very few arguments; everything gets moved from the base store to the provenance store, with associated pointers needed. When the threshold moves to 50, however, a substantial part

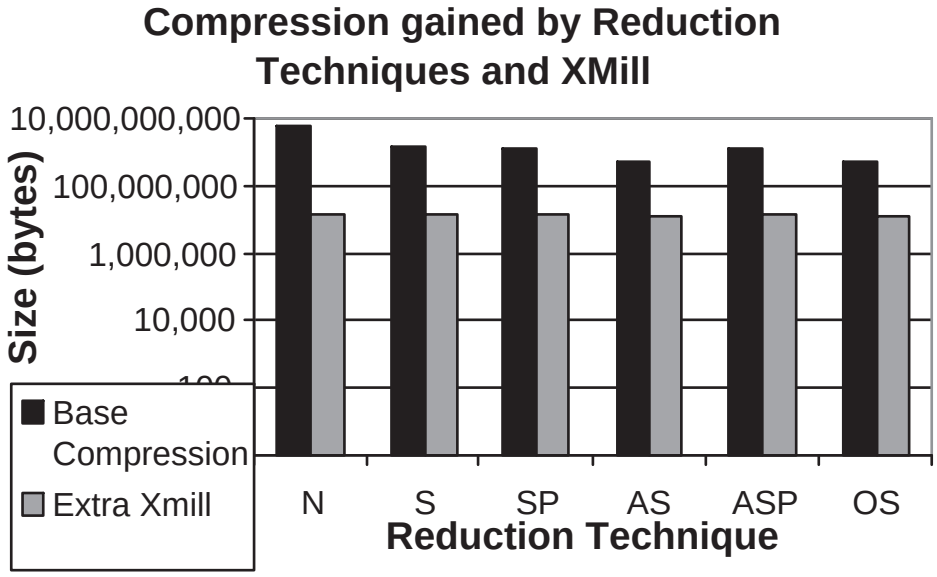
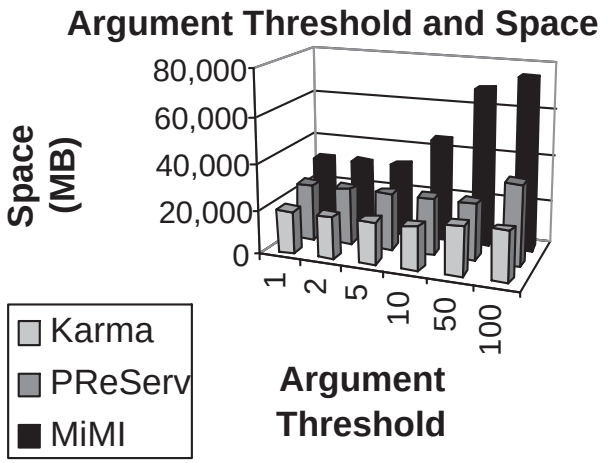


Figure 4.12: XMill applied to the reduced provenance stores.

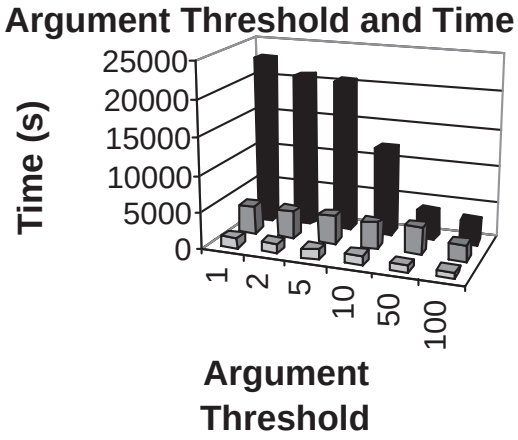
---

of provenance records get treated as arguments, and are left untouched in the provenance store. Unfortunately, as the threshold gets larger, there is again a disadvantage since fewer items qualify to be moved from the general store to the provenance store. Depending on the dataset, the Argument Threshold affects runtime and reducibility differently. Reduction of the Karma and PReServ stores performs best in time and space with an Argument threshold of 10. This is a reflection of how often common sources or manipulations are used in each. MiMI utilizes the same sources and manipulations over and over, while the processes used to generate Karma and PReServ do not.

Between 5-50 seems to be a robust range of values for the Argument Threshold, likely not to be too far from optimum. This is a good range in which to set the default value for this parameter, as a rule of thumb, although as shown with MiMI, some twiddling of this knob may be required.



(a)



(b)

Figure 4.13: Argument Factorization efficiency dependence on Argument Threshold.

4.4.8 Practitioner's Guide

If only one type of reduction were to be used, we would recommend Argument Factorization. We have already seen that it results in better reduction than the other Factorization techniques. Argument Factorization is the hands down winner for the following reasons:

- It has smaller reduction times due to a reduced number of writes.
- It is order invariant and does not depend upon whether user functions are reflexive, symmetric and transitive.

If further reduction is desired, we suggest the following setups based on data and usage criteria:

For Best Storage Reductions

Data Characteristics	**Recommended Tech.**
All	Structural Inheritance
Most data types have specific process e.g. every name element gets normalized	Predicate Inheritance

For Best Query Times

Query Characteristics	**Recommended Tech.**
All	Structural Inheritance [1]
Uses provenance as a condition (high or low selectivity)	Argument Factorization
Uses provenance as a condition and data has type-specific processes	Predicate Inheritance

Note that if the data or query contains several characteristics listed above, our techniques can be combined. The combination is synergistic, and will do more together than either alone.

4.5 Conclusion

Provenance storage is becoming essential for scientific research, but the size of provenance can overwhelm the size of data, in most cases. In this chapter we presented a strategy to reduce provenance storage size. Specifically, we developed a family of Factorization algorithms, as well as algorithms that exploit Predicate and Structural Inheritance. We described how to apply all three techniques in tandem to the same data set.

Our experimental assessment showed that our strategy can reduce the size of provenance by up to a factor of 20. The reduction algorithm scales linearly with provenance store size. Provenance remains queriable, even after reduction using our strategy. In fact, some classes of queries run faster on the reduced store. Also, our reduction strategy is orthogonal to traditional text or XML compression: both can be applied in tandem to get additional reduction, if queriability is not a requirement.

Our work has assumed a generic enough provenance model that many existing systems could easily be mapped to. We are currently in conversations with owners of large scientific data sets to have them adopt our provenance reduction techniques on their production data. We create a suq of possible choices for the user, allowing different compression techniques to be selected and utilized according to the unique characteristics of the data and provenance stores. Code and data used in this chapter are available at:

http://www.sigmod.org/codearchive/sigmod2008/.

---

[1]Requires availability of an iterator to trace ancestors.

CHAPTER V

UNDERSTANDING PROVENANCE BLACK BOXES

In the well-prescribed, white-box world of databases, it is possible to collect very fine-grained provenance on every table, tuple and cell and know exactly where the information came from and why it is in the result set [23, 29, 30, 48, 49, 50, 46, 64, 71, 88, 144]. On the flip side of the coin are workflow systems that allow any number of user-defined and implemented processes to manipulate the data [3, 4, 19, 32, 52, 62, 75, 86, 94, 97, 98, 101, 125, 109, 151, 150]. Unfortunately, the provenance collected for these black-box processes is limited in scope and granularity. The provenance is enough for an automated system to determine the processes and parameters, but fails to allow human users understanding of what happened to a data item within the black-box.

Consider the following query and what happens in a database system that tracks provenance:

Query 1.

```
SELECT *
FROM HPRD
WHERE molecularWt <
   SELECT AVERAGE molecularWt
   FROM HPRD
```

Given the well-defined relational algebra, when a user sees the tuple for LXR in the result set, the provenance can describe *where*, i.e. the exact cell the information came from and *why*, i.e. the series of transformations and input tuples that influenced the fate of the tuple of interest [29, 64, 48]. The provenance answers are precise and easily human understandable, but are only possible because of the well-characterized relational algebra

used to produce them.

Now consider what happens within a workflow system that strings together a series of processes. If you take the query evaluation plan used by a relational system for Query 1 (Figure 5.1(a)) and create processes for each step, you get the workflow described in Figure 5.1(b). Using a Workflow Management System (WfMS) with this workflow allows large amounts of provenance detail to be captured such as software modules used, environmental run-time conditions and parameters. However, given that these processes are now user-defined and written, we have lost the ability to provide fine-grained *where* and *why* information to the user. Real problems can arise that need in-depth provenance to provide an adequate answer. For instance, in 1999, for only a few hours, if a user searched for *A Guide to Programming in C++* [47] in Amazon.com, the top result was *A Hand in the Bush, the Fine Art of Vaginal Fisting*[1]. Why was this the top result? Was there a problem in the underlying books database? Was there a bug in a post-query processing step, or perhaps a malicious hack? Provenance should provide a means for determining the answer.

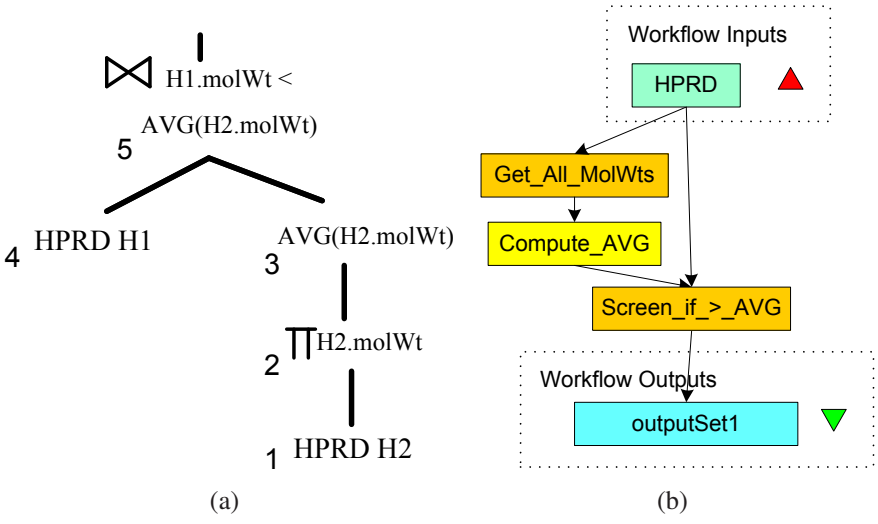


Figure 5.1: (a) The Query Evaluation Plan used for Query 1. (b) The workflow that mimics the Query Evaluation Plan in (a) and can be used in a WfMS.

5.0.1 The Problem

As illustrated above, provenance from WfMSs can only satisfy a portion of the scientist's need, since the granularity of the answer is at the black-box level. It contains both too much and too little information to be useful. It contains too little information to allow human users to fully understand what happened. No WfMS model today can point, for example, to the specific attribute value that resulted in a particular function being applied. If a user doubts the veracity of information coming from this function, knowing further details is essential. However, it can tell her the set of manipulations applied, the set of parameters and the source data on which the answer depends, which can be too much information to process by a human. Furthermore the manipulations are typically defined in terms of software modules/functions invoked, and the source data on which a result depends can easily become a large set. In consequence, the scientist is overwhelmed with irrelevant detail. Finally, we note that what constitutes a good explanation really depends on what the scientist already knows, and what her own mental model of the domain is.

For instance, when a scientist using a WfMS looks at why a data item is in the result set after the workflow in Figure 5.1(b), the provenance store will bombard her with the set of manipulations that acted upon the data, all of their respective parameters and the overwhelming set of input data items that led to the data's inclusion in the result set. Instead, consider how a human would answer the question. She would say, "Because the molecular weight of LXR is 51,102Da, and the average molecular weight for proteins in HPRD is 100,000Da." From this step, based on her personal map of the world, she could be surprised either that LXR weighs 51,102Da, or that the average in HPRD is 100,000Da. In other words, current techniques are not only missing information (51,102Da $<$100,000Da), but providing too much information. Therefore users struggle to understand data in the result set.

We seek to create a model of "provenance answers" to address the above challenges. Our basic idea is to maintain provenance information at a very fine grain, but to develop

a "roll up" model so that the scientist can be presented with a succinct but comprehensive explanation to begin with. From here, the scientist can "drill down" to get recursively more detailed explanations of aspects of interest. In addition, we describe a class of information hereto unexplored by provenance systems that is essential for good user understanding. Even with this extra information, each individual explanation should still be kept succinct, by suitably limiting scope and granularity.

Solutions to assist users in understanding large provenance stores are being explored by [41, 45, 121]. While these systems assist users in mining and visually exploring the data, they fall short of providing "provenance answers" since the provenance information they contain is limited as discussed above. Moreover, while they help organize provenance information presented to the user, they do not allow the user to explore the provenance store according to their own view of the world. While solutions such as [45, 121] allow the user to choose which parts of the provenance store are important and abstract away provenance information that is not needed, these approaches require users to interact with the provenance store or workflow and decide what pieces can be abstracted away. This approach is less effective for end users who may not be familiar with the processes run.

5.0.2 Chapter Outline

Thus, the provenance information as it exists today contains both too much and too little information. It contains information that is imperative for re-running, but which swamps a human user with irrelevant details i.e. too much information. On the other hand, when a user is stymied about what happened to a data item, being given a set of manipulations and their parameters is not helpful i.e. too little information.

In this chapter, we first highlight the need for better provenance explanations in Section 5.1, and present an overview of our proposed solution. Sections 5.2–5.3 describe expand the foundations set down in Chapter II, and the expansion of provenance needed and a new way to access provenance information respectively. The usability of this new provenance

information is evaluated in Section 5.4. The structures needed to store the new provenance information for result explanations are described in Section 5.5 and evaluated in Section 5.6. In Sections 5.7–5.8, we discuss adding our work into traditional workflow systems and conclude.

5.1 Preliminary Investigations

Many people understand what will happen to the data given a SELECT function, but what should be expected from bcgCoalesce [6]? To explore this disconnect between information contained in the provenance store and what a user needs for understanding, a small set of user interviews were performed.

For this experiment, it was imperative for the user to feel they understood what *should* happen to the data from the query and input dataset. Thus, when the actual results are produced, and it does not conform to their conception of the world, a constrained set of provenance questions would be asked. In order to accomplish this, the users were provided with the following information:

1. A small books database. They could see all of the books in it.

2. The application setup, i.e. that they would type their query into the web interface of a database-backed website, which may have several server-side scripts acting upon the data before it was displayed.

3. The set of queries. Table 5.1 contains the set of queries the users were asked to enter into the system.

The users were asked to issue provenance queries verbally to understand the results presented for each result set. We presented provenance in two different flavors: Process [4, 11, 12, 21, 29, 32, 61, 62, 67, 73, 75, 76, 104, 122, 138, 150] and Lineage [13, 14, 17, 29, 30, 48, 49, 50, 106, 140, 148]. The Process provenance stores explicit records of the input and manipulations used while Lineage provenance provides the

Query 1: Select all books whose price is less than the average book price.
Results:

Author	Title	Price
Euripides	Medea	$16
Hrotsvit	Basilius	$20
	Agamemnon	$5,000

Reason: The query evaluation plan used returns all books less than average, and all books with no author.

Query 2: Select all 'Greek epic' books.
Results:

Author	Title	Price
Euripides	Medea	$16
Handford	Where's Waldo	$92
Homer	Odyssey	$49
Sophocles	Antigone	$61

Reason: The server-side script that determines the "type" of book uses book attributes to determine 'Greek epics': if (author=Euripides) or (author=Sophocles) or (author.contains('H' and 'O') and title.contains('O' and 'D')).

Query 3: Select all books written by women.
Results:

Author	Title	Price
Jane Austen	Emma	$50
Charlotte Bronte	Jane Eyre	$16
Emily Bronte	Wuthering Heights	$49
Edith Wharton	The Reef	$20
Anna Karena	Leo Tolstoy	$70

Reason: The data in the database is incorrect. The title and author are reversed for (Leo Tolstoy, *Anna Karena*).

Query 4: Select all books where author = null.
Results:

Author	Title	Price
	Agamemnon	$5,000
Null	Healthy Cooking	$10

Reason: The server-side script runs two queries, author==NULL and author=="Null", and returns the merged results.

Query 5: Select all books written by women.
Results:

Author	Title	Price
	Epic of Gilgamesh	$50
J.K. Rowling	Harry Potter and the Goblet of Fire	$10

Reason: The query created by the application selects on year, but ignores BC and AD.

Table 5.1: The Queries the Users were asked to type into the books website.

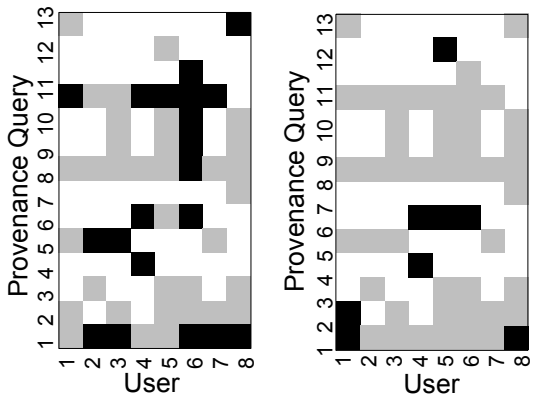


Figure 5.2: Users asked a series of provenance queries to explain the results in Table 5.1. They declared whether or not the provenance sufficiently answered their concern. Black areas indicate an adequate provenance answer; grey areas show inadequate provenance answers. The provenance question was not asked in the white regions. Users were shown both (a) Process Provenance and (b) Lineage Provenance.

---

series of transformation and set of input tuples. Both Process and Lineage provenance information were presented to the users in answer to whatever question they asked. The users were then asked if they understood what happened to the data. Figure 5.2 shows the users satisfaction with the answers provided.

While solutions such as [41] and [44] help the user navigate through the provenance information itself, more needs to be done to make provenance information more usable. Indeed, for the results found in Figure 5.2, provenance was presented in the manner found in [41]. Despite using a user-friendly method to browse and reduce the size of provenance viewed by a user, the provenance was deemed confusing and unhelpful.

5.1.1 Our Solution

After watching the users attempt to understand the result sets and the processes used to produce them, we identified two major stumbling blocks: a dearth of database-style why and where provenance, and occlusion of important information by human-irrelevant data.

In an attempt to assist human users in understanding these provenance black-boxes, we identify a minimal amount of extra information needed to bring us closer to database-style

why and where provenance within the WfMS framework. Moreover, since adding this information increases the amount of material (which was already overwhelming) that a human-user needs to view in order to understand the provenance of a data item, necessitating the creation of a "roll up" then "drill down" querying format. Combined, we call this new information and user query style: Provenance Answers.

5.2 Extended Foundations

In Chapter II, we define a basic logical data unit a *data item*. In this chapter, we utilize this definition, but extend it so that a data item may contain a set of *features*. A data item that is a tuple contains features that are attributes; a data item that is an XML element contains features that are child elements or attributes. Each feature is associated with a data value. Features can be single or multi-valued. A *dataset* is comprised of a finite set of data items.

Running Example Information is selected from HPRD and BIND in Figures 1.2–1.3 according to Query 1 and Example 7, respectively. The resulting data items are then run through a MERGE process until no further merges take place. The workflow is shown in Figure 1.4(b). We will follow three data items shown in Figure 1.2–1.3: LXR from HPRD and RLD and NR1H3 from BIND. The final data item is shown in Figure 1.1.

We also must enrich the concept of Manipulation begun in Chapter II. A MANIPULATION is a basic unit of processing in a workflow. Each MANIPULATION takes one or more data sets as input and produces a dataset as output. We write $M(D^{I_1}, D^{I_2}, ...) = D^O$ to indicate that MANIPULATION M takes datasets D^{I_1}, D^{I_2}, etc as input to generate data set D^O as output.

For example, the MANIPULATION MERGE, shown in Figure 1.4(b), applied to a subset of HPRD and BIND produces an output set comprising entries from the input sets that have been merged based on biological equivalence. The MERGE MANIPULATION first runs a Merge_by_ID function that uses external id equality to merge like proteins.

Then it runs a Merge_by_Sequence function to find all proteins that have identical sequences and merges them. A MANIPULATION in this case is equivalent to the concept of a "Process" described in [110]. In short, a MANIPULATION is a discrete component of a workflow, and uses a set of specific features from the input dataset.

Example 7.
*SELECT * FROM BIND*
WHERE function="receptor"

An instance of a MANIPULATION applied to a specific data item we call a *manipulation*. We write $m(d^{I_1}, d^{I_2}, ...) = d^O$, where $d^{I_1} \in D^{I_1}$, $d^O \in D^O$, etc. m is an instance of M applied to specific data items d^{I_x} within dataset D^{I_x}. For example, an instance of Merge_by_Sequence applied to the molecule LXR in HPRD and the molecule RLD in BIND results in a merged molecule LXR-RLD. At this point, WfMS do not have the capability to peek into black-box processes enough to know when a MANIPULATION is applied to a specific data item.

Manipulation 3. $MERGE$
For every pair of proteins in the dataset, MERGE runs a Merge_by_ID *function that uses external id equality to merge like proteins, then runs a* Merge_by_Sequence *function to find all proteins that have identical organism and sequence and merges them.*

Example 8.
The molecules LXR *in HPRD and* RLD *in BIND are involved in a* Merge_by_Sequence *manipulation.* m *is Merge_by_Sequence;* d^{I_1} *is* LXR*;* d^{I_2} *is* RLD*;* d^O *is the* LXR-RLD *molecule.*

A dataset may contain the molecule ABC1 as well as LXR and RLD. Currently, it is impossible to distinguish via the provenance collected in WfMSs whether the molecule ABC1 was used by a particular manipulation if it is part of an input set.

5.3 Expanding Provenance Information

5.3.1 Necessary Features

As described above, some data items may not be used by particular *manipulation*s. Additionally, each data element has multiple features; not all of these are used in any

manipulation. For instance, the Merge_by_Sequence MANIPULATION only considers the sequence and organism features of the molecule data item, and not any of the dozens of other features. Which features are used in a *manipulation* may sometimes be determined by the type signature or parameters of the MANIPULATION. At other times, features may depend on the specifics of the data item or code analysis may be required. This is easier to do in some cases, e.g. if the MANIPULATION is a database query. In the worst case, an overly conservative analysis may identify more features as potentially used in a MANIPULATION than actually are. In our experience, this conservative assignment has not been a serious issue.

Definition 14. $Necessary\ Feature$*:*
A feature of a data item is said to be necessary if it is used by a manipulation.

In Example 8, the feature used by the Merge_by_Sequence MANIPULATION is sequence and organism. However, using the signature of the MANIPULATION is not enough to determine necessary features. The LXR molecule has one sequence feature, but the RLD molecule has two, mslw and msiw. The sequence from RLD, either mslw or msiw, that was actually used is the Necessary Feature.

There is a finite depth to which we will delve to understand why a *manipulation* behaved as it did for a given data item. At one extreme, we have traditional WfMS provenance systems that say only that a MANIPULATION was performed. Above, we describe a more useful amount of information for user understanding, the necessary features. We can also increase user understanding by breaking down MANIPULATIONs further. Consider the following pseudocode for a MANIPULATION:

$MERGE\_BY\_BIO\_FUNCTION(\ a,\ b\ )$
$if\ (a\ !=\ null\ \&\&\ b\ !=\ null)$
$if\ (a.biofunction\ ==\ b.biofunction)$
$merge(a, b);$

If the user wants to know why a and b were merged using this MANIPULATION, the necessary features, $a.biofunction$ and $b.biofunction$ are needed. However, we can supply

more information:

- The statement $(a\,!=\,null\,\&\&\,b\,!=\,null)$ was satisfied.
- The statement $(a.biofunction\,==\,b.biofunction)$ was satisfied.

In other words, the complete answer could include line by line code analysis of how a data item and its features move through a *manipulation*.

Of course, in reality, there will always be a black-box limit. However, if we can make the bulk of the boxes "grayer", the user can understand the data better. In the above examples, we could have delved deeper into the Merge_by_Sequence function to discover that a BLAST algorithm [85] was used within that function, and the values it employs. This information was not available to the provenance system in our example, and hence not included in the explanation. Therefore the answer is not 100% complete. However, it is more complete than an answer that does not indicate that sequences were used to create a merge. To be able to evaluate the completeness of an explanation, we need to bound the universe with respect to which we would like the answer to be complete. At the current time, we believe a reasonable level is that of a function call, including values of arguments thereto. A larger universe, with more complete explanation may become possible in the future. For example, [104, 67] have made an impressive beginning at capturing the provenance of code execution. Note that our notion of *manipulation* is generic, and can be applied at all granularities, not just to individual function calls, but even to individual instructions. As such, we can define completeness generically as:

Definition 15. *Completeness:*
The number of relevant features and manipulations *used to explain a result data item divided by the relevant number of features and* manipulations *actually used, beginning with the recognized inputs to the system for which provenance is being maintained.*

Thus, in the Merge_by_Sequence example earlier, in which we do not capture that a BLAST function was called within the Merge_by_Sequence function, the completeness of the provenance answer would be: 75% (i.e. 2 features + 1 *manipulation* / 2 features + 2 *manipulation*s), assuming all of these were relevant.

Note that the definition of completeness adds a new concept of *relevance*. The reason for this is to recognize that not every user is interested in every last bit of detail in the provenance. Rather, a user typically has certain aspects for which she needs a full detailed explanation, and others for which a cursory statement will do. We consider a feature or *manipulation* relevant to the explanation if it is one that the user asking for the explanation would like to see.

5.3.2 Provenance Drill Down

The complete provenance structure encompassing both process and lineage information can contain an overwhelming amount of information, and may not be appropriate to return to the user. In a decision support context, large volumes of data are rolled up into a data cube [89, 141]. A user starts analyzing data by first looking at a high level view of, say, company sales; from there, based on the user's mental map of the world, he breaks the sales down by drilling down by location, or product. We wish to abstract this notion of drill down and apply it to the provenance domain. In other words, we wish to present a succinct overview of what happened to a data item, and allow the user to drill down for more complete answers based on her mental view of the data.

Consider the user query, why is LXR in the Result Set after the workflow in Figure 5.1(b). When a user asks why LXR is in the result, showing her a dump of processes and input datasets is likely to be overwhelming. It also does not contain enough information to help the user actually understand what happened to a data item. For instance, why did the MANIPULATION Screen_if_>_AVG act on LXR to include it in the result set? The initial answer is that its molecular weight, 52,102Da, is less than the average HPRD molecular weight, 100,000Da. From here the user can be surprised about two distinct variables: the molecular weight of LXR or the AVG weight of all proteins in HPRD. If the user is surprised by the weight of LXR, she can drill down and ask, "Why is the molecular weight of LXR 51,102Da?". The answer of course is because the data item associated with LXR

in the protein table has a molecularWt=51,102Da feature. However, the user may wish to inquire along a different line. She may be surprised that the AVG molecular weight is 100,000Da and wish to ask why. In this case, the answer is the molecular weight feature from every protein in HPRD. Thus, after adding finer-grained provenance information, the correct way to provide provenance answers is recursively allowing the user to drill down in whatever direction she wishes. For example, if she is surprised that the AVG molecular weight is 100,000Da, she can drill down in that direction to discover that the AVG function produced its result based on the values: ABC1.molWt = 254,301Da, Wee1.molWt = 71,597Da, LXR.molWt = 51,102Da, Chk1.molWt = 23,001Da.

5.3.3 Provenance Answer Model

Putting all of the above together, a provenance answer is a recursive exploration of provenance answer Units (PAU). Each PAU explains one *manipulation* by default.

Definition 16. *Provenance Answer Unit:*
Given a specific data item, d^O, obtained as the output of a manipulation *$\mathcal{M}$, every explanation unit will have:*
i. the manipulation*, $\mathcal{M}$,*
ii: $d^{I_1} \in D^{I_1}$, $d^{I_2} \in D^{I_2}$, ..., inputs to $\mathcal{M}$ contributing to d^O's presence in D^O,
iii. the necessary features used by $\mathcal{M}$ to go from each d^{I_1} to d^O.

A provenance answer comprises one or more units. For a given data item, each unit describes a *manipulation*, and the data item features used. For example, for the question, "Why is LXR in the result set, this could be an English sentence:

> LXR satisfied the condition in Screen_if_>_AVG: a molecular weight of 51,102Da is less than AVG molecular weight, 100,000Da. (5.1)

The *manipulation* information, and the data item features used by that *manipulation* are highlighted, so that the user can click on them to drill down. Choosing to drill down into a *manipulation* will merely describe the *manipulation* in more depth. Drilling down into either of the features will take the user to another explanation unit that describes why the chosen feature is in the observed state. In our running example, if the user chose to drill

down into the 100,000Da value, she would obtain:

The 100,000Da is the AVG molecular weight of
ABC1.molWt = 254,301Da, Wee1.molWt = 71,597Da,
LXR.molWt = 51,102Da, and Chk1.molWt = 23,001Da. (5.2)

In this manner, we hope to limit the 'firehose' effect that hampers traditional provenance approaches. The recursive nature of the user's queries leads to a natural tree structure for the model of provenance answer. Notice that the series of MANIPULATIONS is contained in the underlying answer provided by all the provenance systems in the Provenance Challenge. However, provenance answers breaks this information into logical chunks to make it more palatable to the user. Additionally, provenance answer's need for completeness necessitates inclusion of Necessary Features over and above traditional provenance information.

Traditional provenance pieces are contained within provenance answers. At each drill down level, the user will need to understand the *process* that occurred, e.g. AVG. Additionally, the final drill down that a user can ask on any data item is equivalent to its lineage. In the above example, "why is the average molecular weight 100,000Da", will return the same answer as asking the *lineage* of Query 1, i.e. every molecular weight in HPRD. However, provenance answers allow several layers in between. Even in this toy example, knowing that the molecular weight of LXR is 51,102Da is valuable information not found in prior solutions.

In Example 8, the provenance answer model needs to keep track that a Merge_by_Sequence *manipulation* was performed upon the LXR and RLD objects. Moreover, the provenance answer model must keep track that the input that was integral to that *manipulation* outcome was the HPRD.LXR.sequence=mslw feature and the BIND.RLD.sequence=mslw feature. If a series of *manipulation*s were performed, this information must be obtained for each, as well as the relative order of the *manipulation*s. To provide some measure of the how we limit the 'firehose' effect, we introduce

succinctness.

Definition 17. *Succinctness:*
The size, in bytes, of an answer presented to the user at any one step in an explanation.

5.4 Evaluation of Usability

In this chapter, we have introduced a model for provenance answers that enables deeper understanding of provenance black boxes. The first and most important question to ask is whether this exercise is worth it. Are we addressing questions that real users have? For these questions, are we producing satisfactory answers? How do our answers compare with those from other provenance systems, such as lineage tracing and process provenance?

5.4.1 Preliminary Results Revisited

To begin, we presented the users from Section 5.1 with the answers generated to their provenance queries in the form of provenance answers. Their satisfaction is shown in Figure 5.3. Compared with the results for basic provenance collected by traditional WfMSs, shown in Figure 5.2, users were better able to understand what happened to the data. In every case, the users examined the workflow starting at the output and asked about the MANIPULATION that produced the final results. From there, each user explored the series of *manipulation*s, inputs and necessary features uniquely. There was no one common method of exploration, but presenting the provenance answers in this manner allowed each user to explore in the manner most suitable to their understanding. A Chi-square test was performed to determine if provenance answers is better, the same or worse than lineage. A separate Chi-square test was performed to determine if provenance answers is better, the same or worse than process provenance. The p-values found for both tests are 0.1586 and 0.0001 respectively. In other words, using the Chi-square test, provenance answers is not significantly better than lineage, but is significantly better than process provenance. However, because the sample size we are working with is very small, the Chi-square method becomes statistically inaccurate. In the case of two-by-two

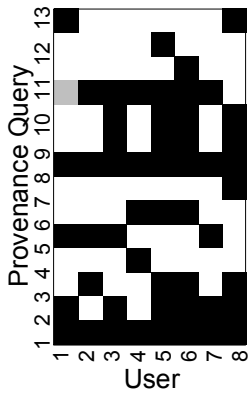


Figure 5.3: User satisfaction for provenance answers from the same users and queries as shown in Figure 5.2.

---

contingency tables that contain fewer than 50 cases, Fisher's exact probability should be used. Using Fisher's, the p-values for PA >lineage and PA >process provenance are 9.39E-22 and 3.75E-14, both of which are very significant.

5.4.2 Real Biological User Satisfaction

To understand the actual user need for provenance black-box answers and to characterize satisfactory explanations, we conducted a series of user interviews using a real dataset and provenance store. The general procedure followed is the same as that in the preliminary user interviews. Users were presented with data that is suspicious, and asked to use the provenance information to gain insight. For the interviews, we mocked up two molecule records similar to what would be found in the public database MiMI [83]. The first record, for the SOD1 molecule, was exactly as it appears in MiMI, and is biologically correct. The second record, for the Collagen molecule, was doctored to contain a mixture of Hemoglobin and Collagen information, and is biologically incorrect. The users were domain experts: five biomedical research scientists. Each scientist was given first the SOD1 record, then the Collagen-Hemoglobin record. The scientists were asked to examine each record in turn and ask aloud any question they wished in order to understand and trust the data. Listed in Table 5.2 is the union of questions asked by the entire group

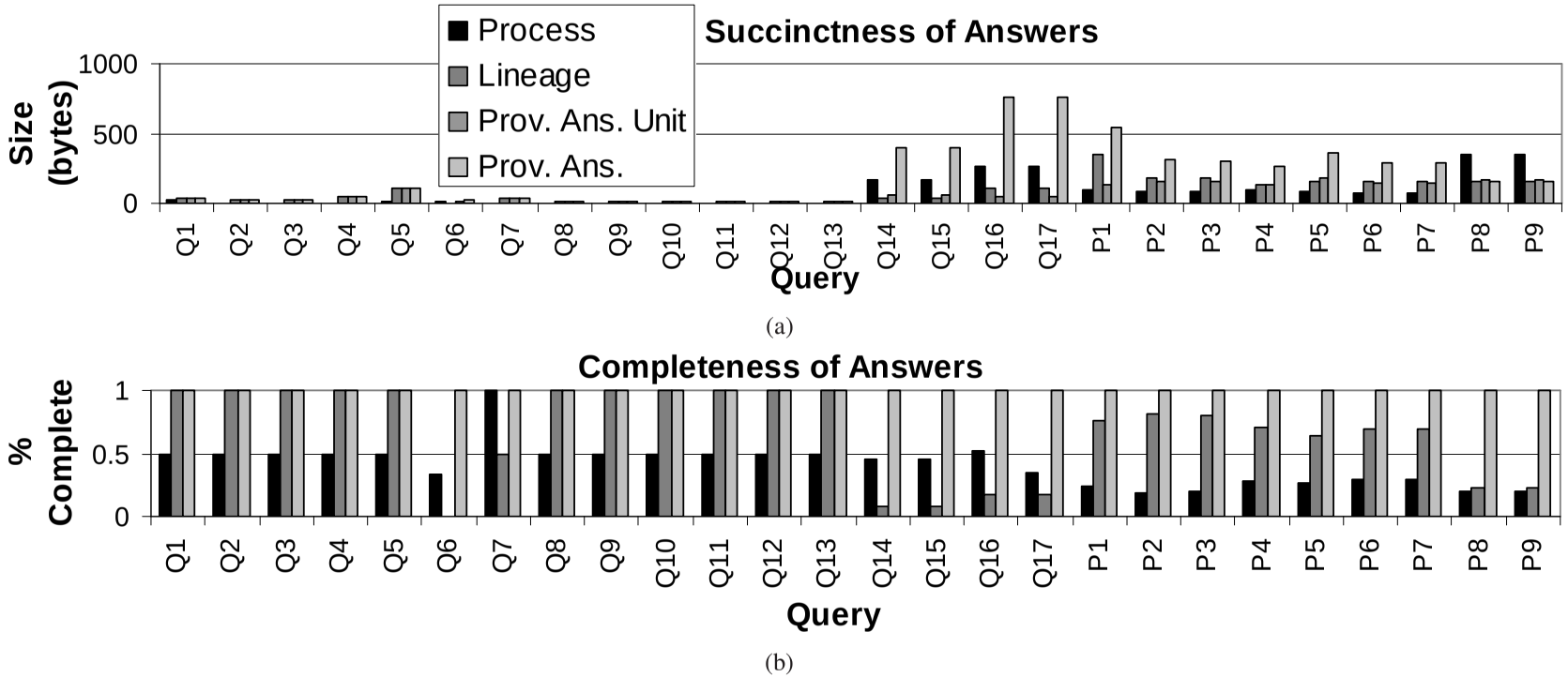


Figure 5.4: Succinctness and Completeness for the Questions in Table 5.2, Q, and the Preliminary Investigation User Questions, P.

Molecule		User Question
SOD1	Q1	**Where** does the molecule SOD1 come from?
	Q2	**Where** does SOD1's description come from?
	Q3	**Where** does the biological process "metabolism" come from?
	Q4	**Where** does the name "Superoxide Dismutase 1" come from?
	Q5	**Where** does the name "Superoxide Dismutase, copper-zinc" come from?
	Q6	**What** processes happened to the SOD1 description?
	Q7	**Why** is "Cytoplasm" a Cellular Component in SOD1?
Collagen	Q8	**Where** does the name "Hemoglobin alpha subunit" come from?
	Q9	**Where** does the name "Collagen alpha-2(l)" come from?
	Q10	**Where** does the name "HBA_Human" come from?
	Q11	**Where** does the function "transporter" come from?
	Q12	**Where** does the process "transport" come from?
	Q13	**Where** does the PTM "acetylation" come from?
	Q14	**Why** is the name "Collagen type 1 alpha 2" in this molecule?
	Q15	**Why** is the function "transporter activity" in this molecule?
	Q16	**Why** did the Hemoglobin and Collagen molecules get merged?
	Q17	**Why** did this Collagen molecule appear in MiMI?

Table 5.2: The Questions users asked while trying to understand information in MiMI.

of users, in the rough order in which they were asked. As each user asked a question, they were presented with three alternative answers: the answer given by process provenance, lineage and provenance answers. For the process provenance answer, the users were given the exact series of processes in tree form, similar to a query evaluation plan or workflow overview. The lineage answer provided the transformations and list of input tuples as described in [13, 14, 17, 29, 30, 48, 49, 50, 106, 140, 148]. They were asked to choose which versions provided acceptable answers to their question. The users were told that we were evaluating three alternative explanation models. Since they are not computer scientists, they would have no way of knowing which was prior work, and which our invention.

The results of these user interviews are found in Figure 5.5. In this figure, we show for each of the 17 questions in Table 5.2, the number of users who asked them and the number that were satisfied by the answer from the three systems being compared. For each question, we show three bars of equal height, representing how many users asked the question. We see that some questions were asked by all users while others were asked only by some. Within each bar in a group of three, one for each provenance technique, the extent of fill shows how many users were satisfied with the answer.

Every biological user started with a **Where** question. This makes sense in the context from which they judge information: which publication, lab, etc validated this information. In the case of **Where** queries, lineage and provenance answers offer the same information, as discussed previously, and both were rated similarly. In the case of Question Q6, a **What** question, *process* provenance was somewhat satisfactory, while extended provenance answers did very well. In the case of SOD1, very few users felt the need to ask **Why** questions. The data looked accurate, and they found no reason to distrust it. This was not the case for the Collagen record. First, each user defaulted to the usual **Where** questions. When the answers proved insufficient to explain the data, the users began asking **Why** questions. For every **Why** question asked, the answer provided by provenance answers

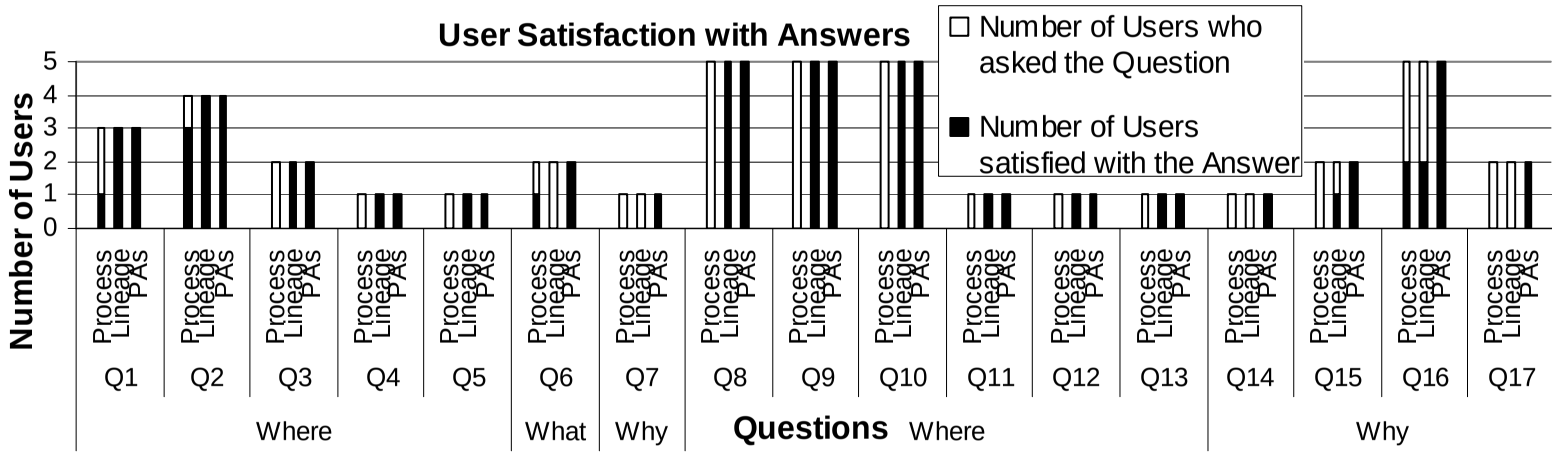


Figure 5.5: User Satisfaction with Answers provided by Process Provenance, Lineage and provenance answers or the Questions in Table 5.2.

was the clear winner.

Additionally, of the ten total **Why** questions asked about Collagen, half were accessed in a Drill Down manner, two were viewed as entire trees and the remainder in a meandering, jumping path. For example, using the Drill Down technique on the Collagen record, the user wanted to see the final merge that occurred, and discovered that it was between an obvious Hemoglobin record and a Collagen record via a Merge_by_id. From there, he wanted to drill into the Hemoglobin side of the result tree to find out how that id came to be. This continued until he traced the id through three other merges to its original source.

Two separate Chi-square tests were run to determine if users thought provenance answers were better, equal to or worse than lineage and process provenance respectively. The p-values found were 0.4961 and 0.000291381. Thus, according to the Chi-square tests, only the finding that provenance answers are better than process provenance is significant. However, as discussed earlier, the Chi-square test becomes statistically inaccurate in our experimental conditions. The Fisher test provides p-values of 0.0011 and 1.02E-16, showing that users prefer provenance answers over lineage and process provenance. Finally, all of the users stated that if they were presented such a broken record with no information to assist them in understanding it, they would immediately leave whatever site they were using. Thus, when users do not trust the data, provenance answers provide a valuable, usable and satisfactory tool for delving into problem.

5.4.3 Succinctness and Completeness

The answers provided by process, lineage and provenance answers for biological (Q) and preliminary (P) user queries were evaluated with respect to both Succinctness and Completeness; results are shown in Figure 5.4. The answers to the user queries, while real, do not stress the limits of size, and can be measured in a few bytes. However, because the same information is used across all three techniques, it shows the difference

in succinctness, despite the overall diminutive size. Lineage and Process provenance vary widely in succinctness between answers, while extended provenance answer units stay uniform even when provenance answers as a whole contain more information as seen in 5.4(a). Also, because provenance answers were designed with completeness in mind, every provenance answer is complete, as shown in Figure 5.4(b). Lineage and Process provenance, on the other hand vary wildly with respect to completeness; the completeness for each is dependent on whether number of input tuples or number of processes is more abundant in the answer to the user question.

5.5 Structures for Provenance Answers

Given the need to peek further into provenance black-boxes, several pieces of information need to be stored. First, all Necessary Features must be associated with the appropriate *manipulation* and data item. Maintaining this information will allow us to answer questions with feature-level granularity. Second, the order of *manipulation*s must be maintained. In order to understand why RLD and LXR were merged, we must understand the order in which the *manipulation*s were performed. In this case, because RLD and LXR both satisfied the selection condition, and then satisfied a merge condition. Third, the original input datasets must be maintained. The system should not have to rebuild the input tuples for explanation. Instead explanations should be given about the transformations from input to output. Finally, feature and *manipulation* information must be stored at data item granularity. Figure 5.6 shows the structures needed to provide provenance answers for the merged RLD-LXR-NR13H molecule.

Throughout the provenance community there has been a question about the merits of lazy versus eager provenance [132]. On the one hand, lazy provenance, which is computed after runtime, only when asked for, is less storage intensive, and is used by [29, 30, 48, 49, 50]. On the other hand, eager provenance as found in [15, 21, 32, 64, 67, 71, 139, 144], stores annotations and dependencies at runtime and

removes the problem of tuple tracing through views since annotations are carried with each piece of data. Moreover, lazy provenance requires either transformations to be reversible, or for a user to define the reverse transformation. Many workflow-driven transformations are not reversible, and most users are not willing to supply a reverse transformation. Because of this constraint, we focus on eager provenance for provenance answers in this work.

Additionally, we should note that some structures needed for provenance black-box answers already exist. Current provenance structures [101, 75, 125], computation plans, query evaluation plans, and even logic proof trees can all be leveraged to provide the basis for provenance black-box answers. For instance, while a query evaluation plan cannot be substituted, it holds some essential information for provenance black-box answers: the tree of MANIPULATIONs.

5.5.1 Base Data Structures

The implementation of the provenance answers model discussed in Section 5.3 can require unacceptable quantities of storage. Consider Query 1 applied to a 1000 row protein table using the query evaluation plan shown in Figure 5.1(a). If the data item LXR was returned, the provenance answer implementation discussed above will store 1000 features with the AVG *manipulation*. Moreover, if LXR is just one data item in a large result set, every data item in that result set will contain an AVG *manipulation* with 1000 features.

While this can be reduced via proper pointer usage so that repeated elements are stored once and referred to many times, the basic provenance answer implementation is still large in its storage costs. The worst case size of the basic storage method is $O(D + oNf)$, where D is the size of the input datasets, o is the number of data items in the output dataset, N is the number of *manipulation*s performed per data item in the output, and f is the average number of necessary features per *manipulation*.

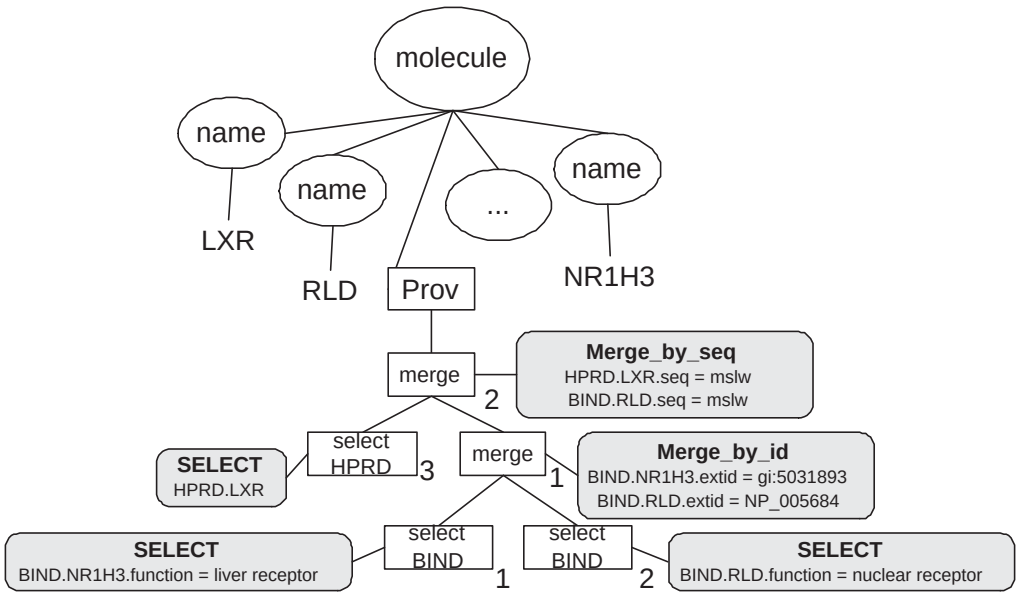


Figure 5.6: The structures needed to fully record provenance answers. Notice that *manipulation* record lists the necessary features of each data item. Not shown, but needed, are the input datasets.

5.5.2 Ambiguous-only Storage

Provenance answers require storage of the input datasets, the sequence of *manipulation*s used to generate each data item, and the features used by each of these *manipulation*s. While standard compression techniques can be used to compress each of these pieces, they will no longer be easily queriable. This is unacceptable; we wish to reduce the storage size needed for provenance answers, while still allowing fast user queries. To this end, we highlight the first possible reduction: the space needed to store the Necessary Features.

Consider the case of the Compute_AVG *manipulation*. There is only one feature called "molecularWt" for a protein. By storing only MANIPULATION =Compute_AVG, feature=molecularWt, we can still explicitly answer questions about the result set. It is only when there are multiple features with the same name, such as repeated elements in an XML document that we must store the complete feature records. For instance, if a merge is performed using the

Merge_by_sequence MANIPULATION, and a molecule has fifteen isoforms and

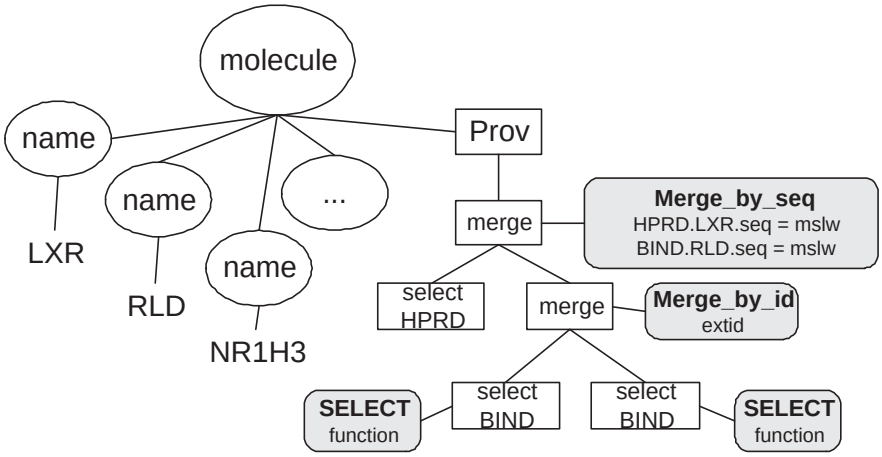


Figure 5.7: The Ambiguous-only structures needed to record provenance answers.

---

therefore fifteen sequences, which one was actually used by the merge function?

Definition 18. *Ambiguous Features*
A feature is ambiguous iff the schema cardinality for the feature is not restricted to 1.

A provenance answer consists of a tree of *manipulation*s and the set Necessary Features for each *manipulation*. A Necessary Feature is Ambiguous if there are multiple possible values associated with the feature. In this case the feature and value must be stored. However, if the feature is Non-ambiguous, and can only have one possible value, then the full feature-value information can be reconstructed from just the feature information.

Based on this insight, we can create the Ambiguous-only storage method. If a feature is ambiguous, then the feature must be explicitly referenced by its path (the exact reference for a tuple or XML element) and value. However, if a feature is non-ambiguous, then we only need to store the feature's path. Thus, instead of storing two strings, we only store one. Moreover, the feature path is the same for many *manipulation*s, while the value is different for each instance. We can reduce the actual storage costs further by storing the feature path once and merely pointing to it from every *manipulation* that uses it, consequently reducing the storage from a string to a pointer. Figure 5.7 shows the MiMI dataset using ambiguous storage to reduce the size needed for provenance answers. Notice

that since extid is a feature with a unique value in every data item, it is a non-ambiguous feature and can be stored in a less space intensive manner. However, the seq feature can have a cardinality greater than one, and therefore must be stored explicitly.

Theorem 5.5.1. *Ambig. Store Schema Dependency*
Ambiguous-only storage size is equal to or less than Basic storage size and is determined by the cardinality of the feature domain.
Proof: *Basic and Ambiguous-only storage size for each data item is* $f * (p + v) * N$ *where* f *is the number of Necessary Features used and* N *is the number of* manipulations*;* p *is the size of the string needed to represent the feature's path, and* v *is the size of the string needed to represent the feature's value. For every data item in the result set, if the schema cardinality for a feature is zero or one (?), then the Ambiguous-only storage size is* $fNt + p$*, where t is the size of the pointer.*

Utilizing Ambiguous-only storage requires that the database be static or that the input database be versioned so that the set of contributing data items is clearly established for any output to be explained. Consider the consequences of allowing updates to a database that used ambiguous-only answer storage. If the user executes Query 1, the molecularWt feature for every protein is used. Ambiguous-only storage will only store the information that the feature 'molecularWt' was used. If a new protein is now inserted into the database, the provenance answer instantiation will automatically include it into the answer for LXR. Since the new protein did not in fact contribute to the result set, this is obviously incorrect. Obviously this static requirement will make ambiguous-only storage unusable for some applications. However, a subset of applications remain for which ambiguous-only storage is a viable alternative. These include 'publication' systems such as HPRD, SwissProt or NCBI, that release information in a manner that restricts updates. Fortunately, as discussed in the Section 5.5.3, there is also a structure that reduces storage size but does not require a static dataset.

5.5.3 Watermarking

Given many *manipulation*s that contain ambiguous features, the size needed to store provenance black-box answers can still grow to be large. In essence, for a result set

greater than one, ambiguous-only storage will repeat the same *manipulation* and necessary features over and over. The only difference for each data item will be the values of the necessary features.

Instead of repeating data, by tagging each intermediate result set with necessary features, we can instead tag the original input datasets. If we assign a 'color' to each distinct *manipulation*, any feature used by that *manipulation* will be 'colored' in the input datasets. Imagine a result set of 50 molecules generated by the query "SELECT * FROM BIND WHERE chromosome=X;". Using Basic or Ambiguous-only methods, each of those 50 molecules will have a reference to the SELECT *manipulation* and the chromosome feature. Using watermarking, we can say SELECT='red', and color all chromosome features used by the SELECT *manipulation* red. Figure 5.8 shows this solution applied to the MiMI dataset. The Merge_by_Sequence (black) *manipulation*, in the LXR-RLD-NR13H molecule can be matched to the exact sequence attribute used in the input LXR and RLD molecules.

The size of a watermarked store is still $O(Nf)$ where f is the number of Necessary Features used and N is the number of *manipulation*s. However, because watermarking can be achieved by storing an integer for every Necessary Feature instead of a string, substantial savings can actually be achieved.

Watermarking has another unique benefit. In all previous provenance answer strategies, a key must exist for the data item. Consider Basic provenance answer storage, applied to Example 8. Provenance answers require us to keep the set of input data items contributing to the output. In the example these are uniquely identified by name as HPRD.LXR and BIND.RLD. However, many times there is no key for a data item. To surmount this problem, Basic and Ambiguous-only provenance answer implementations would have to assign a unique id to a data item before using it in a *manipulation*. Watermarking automatically identifies the object in the input dataset, even if it is keyless, merely by the presence of a color in that data's feature set.

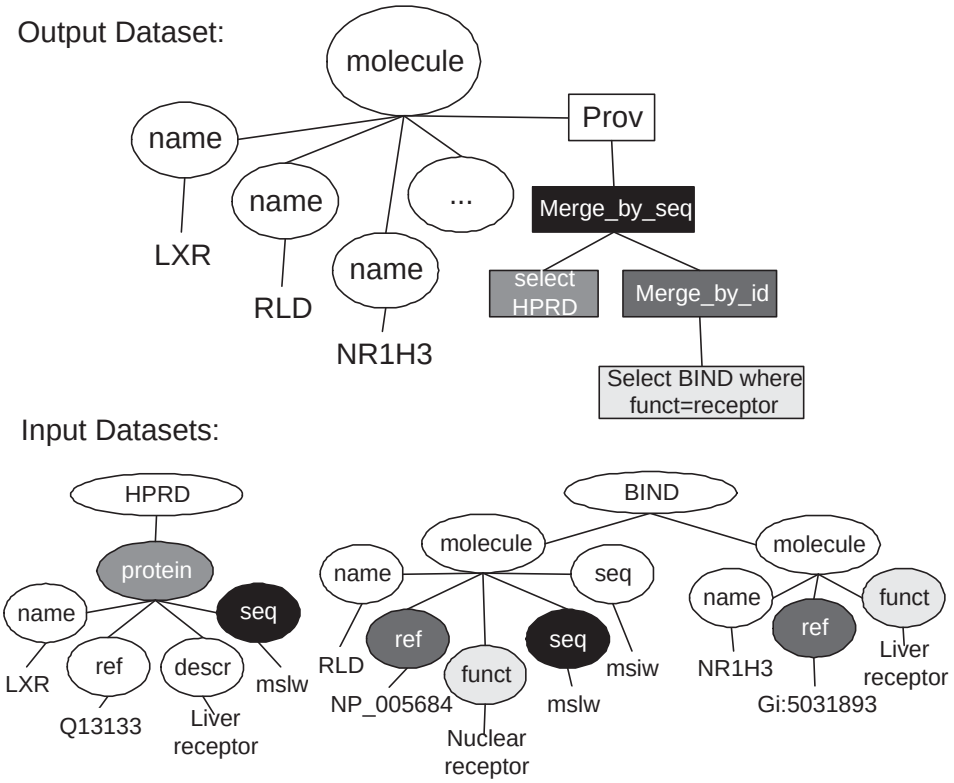


Figure 5.8: The watermarking strategy for provenance answers in MiMI.

Unfortunately, in this form, watermarking cannot be used whenever there exists a MANIPULATION that produces intermediate results that are the result of an aggregation. Consider the SQL query evaluation plan in Figure 5.1(a). Each node in the evaluation plan can be considered a MANIPULATION. If this is the case, the $AVG(molecularWt)$ is an aggregation MANIPULATION used to create intermediate results. Once the query is complete, there exists nothing to watermark for that *manipulation* in the input datasets. In order to handle this, a new node containing the aggregation result is added to the 'input' dataset. This new node can then be colored with the appropriate *manipulation*, and watermarking can be applied even when aggregation MANIPULATIONs are used.

5.6 Evaluation

In Section 5.4, we showed that provenance answers can be very valuable to a user. Here we study the overheads incurred by provenance answers. To this end, we built a protein interaction database from multiple source databases and captured provenance answer material. Data in the protein database is derived through a workflow comprising multiple MANIPULATIONs, with a variation of between 1 and 10 manipulations being applied to create any one data item. MANIPULATIONs include: ingestion of information from HPRD [114], ingestion of information from DIP [55, 58, 117, 146], ingestion of information from IntAct [78], merge_by_id, merge_by_protein_name. The final database size is 60MB; the provenance store is 400MB. To show that our model for provenance answers is implementable and usable, we applied each provenance answer storage technique to a real-world dataset creation workflow. Provenance answers require an extension of a DBMS or workflow system; however construction of final provenance answer answers can utilize existing DBMS and provenance tools. The workflow first ingests all datasets. Then it iteratively applies the two merge MANIPULATIONs on the three ingested datasets until there is only one dataset and no new merges occur. All experiments were run on a Dell workstation with Pentium 4 CPU at 2GHz with 640MB RAM and 74.4GB disk space

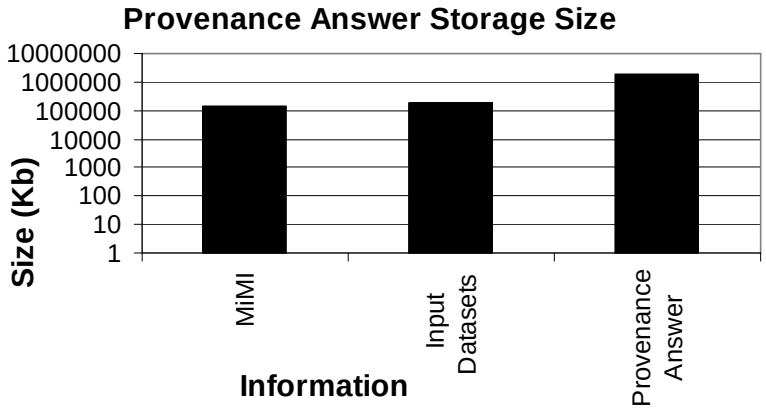


Figure 5.9: The space needed to store provenance answers.

---

with Windows XP. The algorithms were implemented as a Java application with a mySQL backend.

5.6.1 Time and Space

Figure 5.9 describes the space needed for various aspects of provenance answer: the final dataset produced by running the series of MANIPULATIONs described above, the input datasets, and the Basic implementation of the provenance answer model. Provenance answers can grow larger than the input and output datasets. The size of the provenance answer store is highly dependent upon the number of *manipulation*s performed. Figure 5.10 shows how the provenance answer store grows with the number of *manipulation*s. These results concur with the linear nature of the manipulation storage aspect of the provenance answer model. They also show that while provenance answers may be large, they grow linearly.

In Figure 5.11, we show the storage costs associated with the possible data structures for provenance answers. In order to highlight the differences in storage needed for the extra component of provenance answers not found in prior provenance work, the sizes for the underlying provenance structures are not reported. The ambiguous-only results presented reflect the dependence this method has on the underlying schema. No feature in

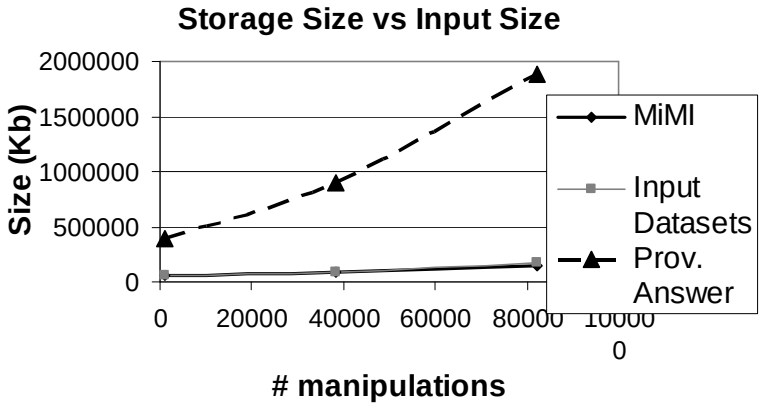


Figure 5.10: The growth of provenance answer storage.

---

the dataset used by a MANIPULATION is non-ambiguous. Therefore Basic and Ambiguous sizes are the same, as stated in Theorem 5.5.1. However, the watermarking strategy produces a provenance answer store that is 62% the size of the Basic provenance answer store, showing that savings are possible with a good implementation.

Figure 5.12 shows the overhead incurred by storing provenance answer information while running the series of MANIPULATION discussed above. The overhead is the time it takes to store all information above and beyond traditional provenance information that is needed for successful provenance answers. None of the implementations of provenance answer create a large time-sink. Indeed, with a worst case of 7.4%, storing provenance answer information will not significantly slow down any system. Watermarking does even better with only a 7% overhead, since the size of the information required is less.

5.6.2 Querying

Provenance answers would be useless if they took too long to generate. Because the underlying structures lend themselves to the type of queries asked by users, we found that queries are both easy to express and quick to execute. We took the questions asked by the biological user interview participants, and determined a set of representative queries. Queries q1-q2 ask traditional provenance questions that do not utilize provenance answer

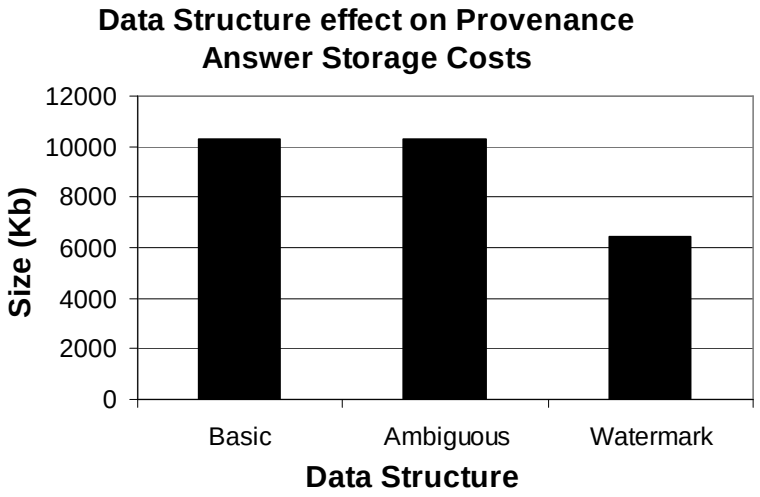


Figure 5.11: The space needed for the three possible ways of storing provenance answers.

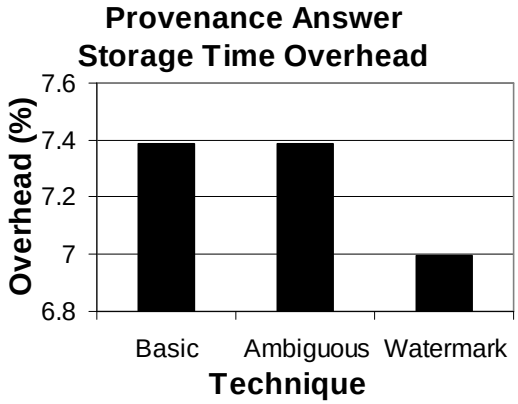


Figure 5.12: The time overhead incurred by storing provenance answer information at database construction time.

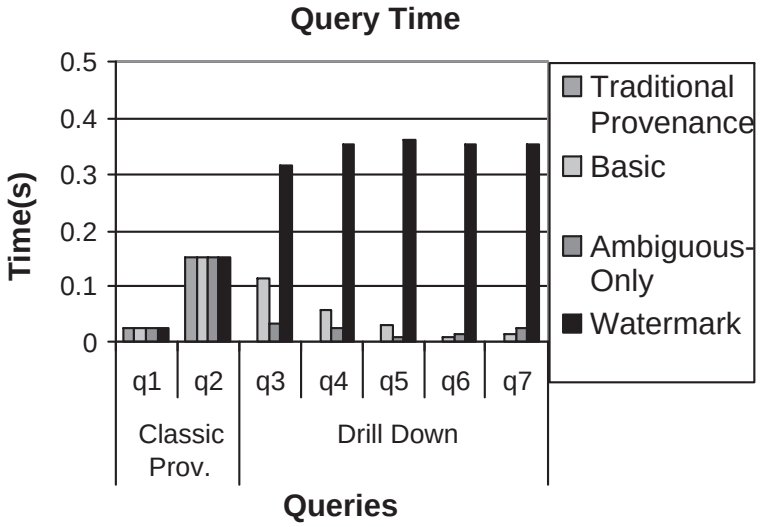


Figure 5.13: Query execution time for provenance answers.

---

information. Query q1 asks for the last manipulation performed to create a record; Query q2 asks for the last three. These two traditional provenance queries provide a baseline to compare against. Meanwhile Queries q3-q7 are queries representing the five **Why** questions in Table 5.2. These queries were expressed as SQL statements.

Figure 5.13 shows the query times. Each query was run five times on a cold cache, and the average was taken of the middle three values. The times reported correspond to the time needed to return the entire provenance answer tree. Unfortunately, many of the provenance answer queries take more time, since they are returning more information. However, there are exceptions to this statement. Provenance answer queries, except for watermarking, do better than some traditional provenance queries, since we are taking one explanation step at a time, in order, instead of searching the entire provenance tree for a particular manipulation. The provenance answer times are on par with Query q1, but return more information. Watermarking requires an extra join to get to manipulation descriptions, so does worse than other methods. Finally, we would like to point out that even relatively long-running provenance answer queries are fast.

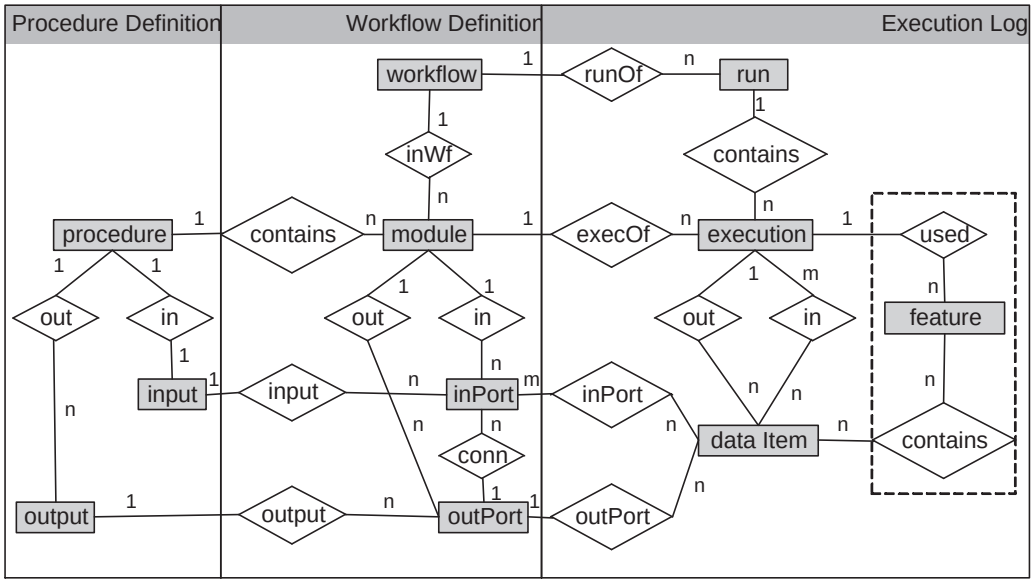


Figure 5.14: An abstract overview of the provenance needs of a Scientific Workflow System, from the VisTrails contribution to the Second Provenance Challenge, annotated to allow provenance answers, shown in the dashed box.

5.7 Provenance answers in a Workflow System

Because provenance answers leverage many provenance structures, it is relatively easy to add provenance answers capability to a workflow system. In general, in order to make such a system work with provenance answers, the following pieces must be adapted:

1. The API a workflow component uses to integrate with the workflow.
2. The calling of components while running the workflow.
3. The provenance storage backend.

Figure 5.14 contains a high-level view of the needs of a scientific workflow system which we have annotated to accommodate provenance answers.

For concreteness, we will briefly look at how provenance answers can be added into VisTrails [32]. Beginning with Requirement 1 above, VisTrails allows users to create Packages to define custom MANIPULATIONs (called modules). When creating a Package, the user creates an module identifier, and a module registry that stores the inputs and

outputs expected. Half the battle is already won, since we already have hooks for the MANIPULATION identifier and the features to consider as Necessary Features. The code within the package must be written to keep track of the values of all Necessary Features. From here, we must consider Requirement 2; when the workflow is executed, a record of the execution of each module is already created in the VisTrails Logs. There must be an extension of the interaction between modules and workflow execution though, since each module must report back the exact Necessary Feature set for storage in the Logs. Finally, the logs, both file-based and database in VisTrails, must be altered. In particular, the table module_exec needs to keep track of the data item worked on and the Necessary Features utilized.

5.8 Conclusions

Workflow systems that capture provenance today are limited in the provenance they capture. Unlike provenance in database systems, the black boxes in workflow systems are not well defined, and therefore limit the granularity at which provenance can be captured. This leads to provenance information that allows workflow systems to recreate the steps involved, but limit human understanding of what happened to individual data items.

In this work, we demonstrate the need that users have for knowing more detailed information about the black boxes. We determine the set of information that must be maintained to make black boxes "grey". Additionally, because this information further complicates an already difficult to understand provenance store, we describe a method of presenting this fine-grained provenance information to the user that can be easily integrated with other provenance viewing systems [45, 121]. Our experimental evaluations show that real users benefit from this increased information, and the storage and time costs of collecting and providing it are manageable.

CHAPTER VI

WHY NOT?

Why did the chicken not cross the road? Why not Colbert for President? Why did Travelocity not show me the Drake Hotel as a lodging option in Chicago? Why do I not have blue eyes? Except for the unfathomable chicken, there is an explicit reason for each of these events not occurring [1]. Understanding why events do not occur is a natural process we use to understand our world. In the arena of databases and software systems, these questions often sound like: Why did this program not complete? Why did this tuple not appear in the result set? etc. The typical response to such questions is an epic debugging session in which the exact series of events is painstakingly traced until the answer is found. Currently, provenance can help explain surprises within a result set. However, what happens when the surprise is not what is found within the result set, but what is missing from the result set? Consider the following set of user problems:

- A scientist searches MiMI [83] for: "sterol AND organism= 'Homo sapiens"'. A known function of ABC1 is "sterol transporter activity", so why is it not in the result set?
- A business traveler searches for flights on a popular flight booking web site, he cannot understand why there is no direct flight from DTW to LAX listed. He took

[1] On November 1, 2007, the South Carolina Democratic Party executive council refused Colbert's ballot application by a 13-3 vote. Graduate students don't make enough to stay at the Drake Hotel, as noted in my cost preferences. I have no blue-eyed ancestors, and according to Mendelian Inheritance do not have the double recessive genes required.

that flight last week, so why is it not in the result set?

- A fan wants to see all the scoop about "the king" in *Return of the King* and types "Vito Mortensen" in IMDB. No Vito Mortensen is returned. Why not?

There is one running theme throughout the problems encountered above, despite the differences in domain: the user does not have the ability to alter their query in any way to garner better understanding of the dataset and result set. For instance, in a standard database system, if the user queries: SELECT name FROM employees WHERE salary >$100,000, and there are no results, the natural inclination is to slightly alter the query. Thus, the user may turn around and enter: SELECT name FROM employees WHERE salary >$75,000. In other words, an experienced classic database user has the means to explore the database and query space. A traditional database user is comfortable using this methodology to explore the characteristics of a dataset, and would have no need to ask WHY NOT?. Unfortunately, many applications and users no longer fit this paradigm. In the above examples, the users are not database users, they are application users who have no access to the underlying dataset. They cannot sift through the dataset to determine WHY NOT? when they encounter an unexpectedly missing result. Additionally, the applications themselves limit the type of queries the users can submit. In the Business Traveler Example above, Travelocity only allows the user to choose dates and location; it is impossible for the user to subtly alter the query to comb through the dataset to find the flight he thinks he knows about. Finally, in a traditional database, a standard, well-understood set of operators exist. In many applications this is not true, and the presence of complex, programmatic operations will obfuscate why data is not in the result set. In the MiMI Example, why is ABC1 not in the result set after the query? The user knows that there is a database behind the application, but how does the keyword query interface with it? How are the results retrieved and displayed? Is there a bug? In actuality, the only reason ABC1 is not in the result set after this query is because MiMI only displays the top 100 hits, and ABC1 falls outside the range. This sort of WHY NOT? question

Author	Title	Price	Location	pubDate
	Epic of Gilgamesh	$150	Middle East	2000 BC
Euripides	Medea	$16	Europe	431 BC
Homer	Odyssey	$49	Europe	900 BC
Hrotsvit	Basilius	$20	Europe	970 AD
Shakespeare	Coriolanus	$70	Europe	1623 AD
Sophocles	Antigone	$61	Europe	442 BC
Virgil	Aeneid	$92	Europe	29 BC

Table 6.1: The set of books in *Ye Olde Booke Shoppe*.

could never be addressed via the sift and comb database search method.

6.0.1 The Problem

After performing a set of relational operators, application functions, or mixture of both, a result set is formed. For instance, the data found in *Ye Olde Booke Shoppe*, in Table 6.1, is the result set of a manual curation of Library A and a Natural Language Processing of Library B, with a merge and duplicate removal process applied to the two outputs. In other words, a set of non-relational manipulations created the result set. When a user queries the *Ye Olde Booke Shoppe* database, a set of relational operators, and perhaps user functions, is used.

Once a result set is formed, if a user is unable to find what she wished, she must specify what she is seeking, using key or attribute values. Using this information, we describe how to offer explanations to the user about why the data is not in the result set.

Example 9. *Table 6.1 contains the contents of Ye Olde Booke Shoppe, and how the titles were included in the bookstore's listings. If a shopper knows that all "window display books" are around $20, and wishes to make a cheap purchase, she may issue the query: Show me all window-books. The result from this query is: (Euripides, "Medea"). Why is (Hrotsvit, "Basilius") not in the result set? Is it not a book in the book store? Does it cost more than $20? Is there a bug in the query-database interface such that her query was not correctly translated?*

WHY NOT? is a series of statements about the potential reasons the data of interest to the user is missing from the result set. We can leverage provenance records [26, 35, 32, 74],

query specification and the user's own question to help understand WHY NOT?. Thus, whenever a divergence occurs between the two, it is a good candidate for finding why a piece of data is not in the result set. In the example above, we can trace (Hrotsvit, "Basilius")'s progress through all the manipulations performed on (Euripides, "Medea"). Every manipulation at which the two do not behave similarly is a possible answer to "Why Not?".

Throughout the rest of this work, for ease of reader comprehension, we utilize a classic book database, with standard relational operators, and a few user defined, "server-side" functions. However, we would like to emphasize that the problem we are addressing exists outside of traditional databases, and our techniques can be applied to applications as well.

In this work, Section 6.1, we provide a model and definitions that allow us to describe a piece of data not in the result set, and ask why it is not there. Moreover, we provide a model which allows us to answer WHY NOT? questions. In Section 6.2.1 we discuss how WHY NOT? answers can be computed. The evaluation of our methods is presented in Section 6.4. In Sections 6.5–6.7, we discuss an extension to this work and conclude.

6.1 Foundations

In Chapter II, we define a basic logical data unit a *data item*, and the concept of Manipulations. In Chapter V, we extend the concept of manipulations. In this chapter, we utilize these definitions. To refresh, each MANIPULATION takes one or more data sets as input and produces a dataset as output. We write $M(D^{I_1}, D^{I_2}, ...) = D^O$ to indicate that MANIPULATION M takes datasets D^{I_1}, D^{I_2}, etc as input to generate data set D^O as output.

For example, the MANIPULATION Select_Books_<$20 applied to the *Ye Olde Booke Shoppe* (shown in Figure 6.1(a)) dataset produces an output set comprising (Euripides, "Medea") and (Hrotsvit, "Basilius"). An instance of a MANIPULATION applied to a specific data item we call a *manipulation*. We write $m(d^{I_1}, d^{I_2}, ...) = d^O$, where $d^{I_1} \in D^{I_1}$, $d^O \in D^O$, etc. m is an instance of M applied to specific data items d^{I_x} within dataset

D^{I_x}. For example, an instance of Apply_SeasonalCriteria applied to the book (Hrotsvit, "Basilius") results in $\emptyset$. Another example MANIPULATION is:

Manipulation 4. *$Apply\_SeasonalCriteria$*
Returns all books that satisfy a seasonal criteria.

Example 10.
Given a Mother's Day Seasonal Criteria (based on the date), d^{I_1} is (Hrotsvit, "Basilius", \$20); d^{I_2} is (Euripides, "Medea", \$16). d^O is (Euripides, "Medea"). Since "Medea" fits the Seasonal Criteria.

6.1.1 WHY NOT? Identity

When attempting to answer WHY NOT?, we have three known pieces from which to draw information: the query, the result set and the question. The query, Q, is the original query or workflow posed against a dataset, and can be broken down into a series of MANIPULATIONS. The result set, R, is the result of that query on the dataset, and the question, S, is the unsatisfied user's WHY NOT?. The question directly translates into the Unpicked.

Definition 19. *$Unpicked$:*
Given a key or attribute set, k, from the user question, S, if k does not exist in the result set, then the set, U, of data items, d, from the input dataset that satisfy the user's question is the Unpicked.

For instance, If a shopper wishes to make a cheap purchase, she may issue the query selecting for window-books. She then asks, "Why is "Basilius" not in the result set?". The Unpicked data item, (Hrotsvit, "Basilius"), is specified by its title feature. Alternately, a user may ask "Why are no 'Penguin' books in the result set?", in which case a keyword-style query across all attributes of the input set will generate the Unpicked (which will contain books with a Publisher=Penguin and Penguin in the title field, etc).

Additionally, given the definition of an Unpicked data item, if the specified key or attribute(s) is found in a data item in the result set, then we do not have an Unpicked, or a valid WHY NOT? query. Finally, note that the Unpicked may be everything; "Why is the result $\emptyset$" or "Why not anything".

Definition 20. $Valid$ WHY NOT? $Query:$

A valid WHY NOT? *query: i. contains an attribute(s) that can be mapped to an Unpicked data item(s) in the input datasets, and*

ii. applies to a single result set.

Now that we have an Unpicked Data Item, in order to determine WHY NOT?, we must be able to trace data items through manipulations, and understand the relationship between data items in the output to data items in the input. In essence, this is the accepted provenance concept of lineage [29, 48, 49]. From [49], "Given a transformation instance $\tau(I) = O$ and an output item $o \in O$, we call the actual set $I^* \subseteq I$ of input data items that contributed to o's derivation the *lineage* of o, and we denote it at $I^* = \tau^*(o, I)$". In other words, the *lineage* of a data item is the set of input tuples that have influenced the inclusion or appearance of that data item in the result set. We utilize the same definitions found in [29, 48, 49] for lineage with the following exception: the lineage of a MIN or MAX output data item is the data item(s) containing the reported value, not the entire input set. In this work, we denote this version of a lineage relationship with $o \overset{m}{\angle} I^*$.

Unfortunately, *lineage* is a concept that applies only to data items in the result set, and traces data items through manipulations from the result set to the input sets. A data item's appearance in the result set is the sine qua non for lineage. In this case, the data items we are interested in are **NOT** in the result set, and therefore do not have lineage. Instead, we must define a new concept, successor.

Definition 21. $Successor:$

Given a manipulation m that takes in dataset I and outputs O, $d' \in O$ is a successor of $d \in I$, iff $d' \overset{m}{\angle} d$.

Definition 22. $Unpicked\ Successor:$

For any Unpicked data item, u, if $\exists d' \in$ the output of m such that d' $\overset{m}{\angle} u$, then there is an Unpicked Successor.

Even though an Unpicked data item by definition does not exist in the result set, or even after a manipulation, we can use this definition of successor to watch how Unpicked data items move through workflows. Notice that a successor depends purely upon the

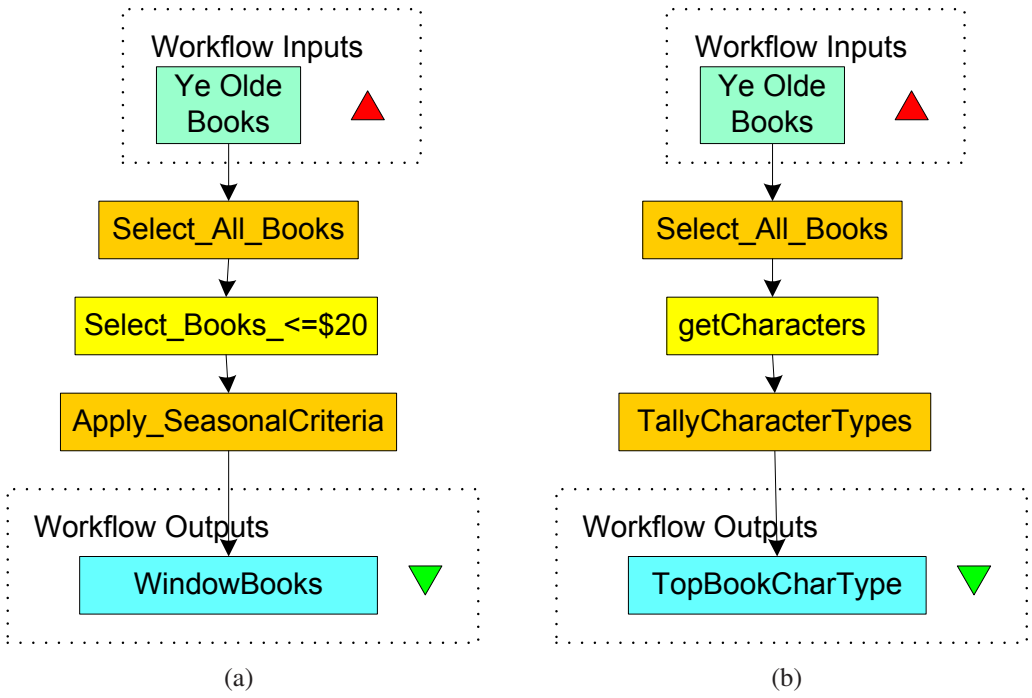


Figure 6.1: A set of workflows. (a) Finds the Window Display for *Ye Olde Booke Shoppe*. (b) Determines the top character genre in *Ye Olde Booke Shoppe*.

---

notion of lineage, not attribute values. After a query, if a user asks, "Why not $61?", it does not matter if a manipulation projects out the attribute $61. Using lineage, the tuple (Sophocles, "Antigone") is directly associated with the input tuple (Sophocles, "Antigone", $61). We do not care how many manipulations we go though between the Unpicked and an Unpicked successor. As long as there exists some lineage relationship between a data item and an Unpicked data item, there is an Unpicked Successor. However, we must make one modification to classic lineage. Traditionally, lineage will trace the ancestry of a data item through multiple manipulations back to the source dataset. In our case, we must only compute lineage through one manipulation, not to the original datasets.

6.2 WHY NOT? Answers

Definition 23. *Picky Manipulation:*
A manipulation is "Picky" with respect to an Unpicked data item set if:
i. an Unpicked data item or Unpicked successor is in the manipulations input set,

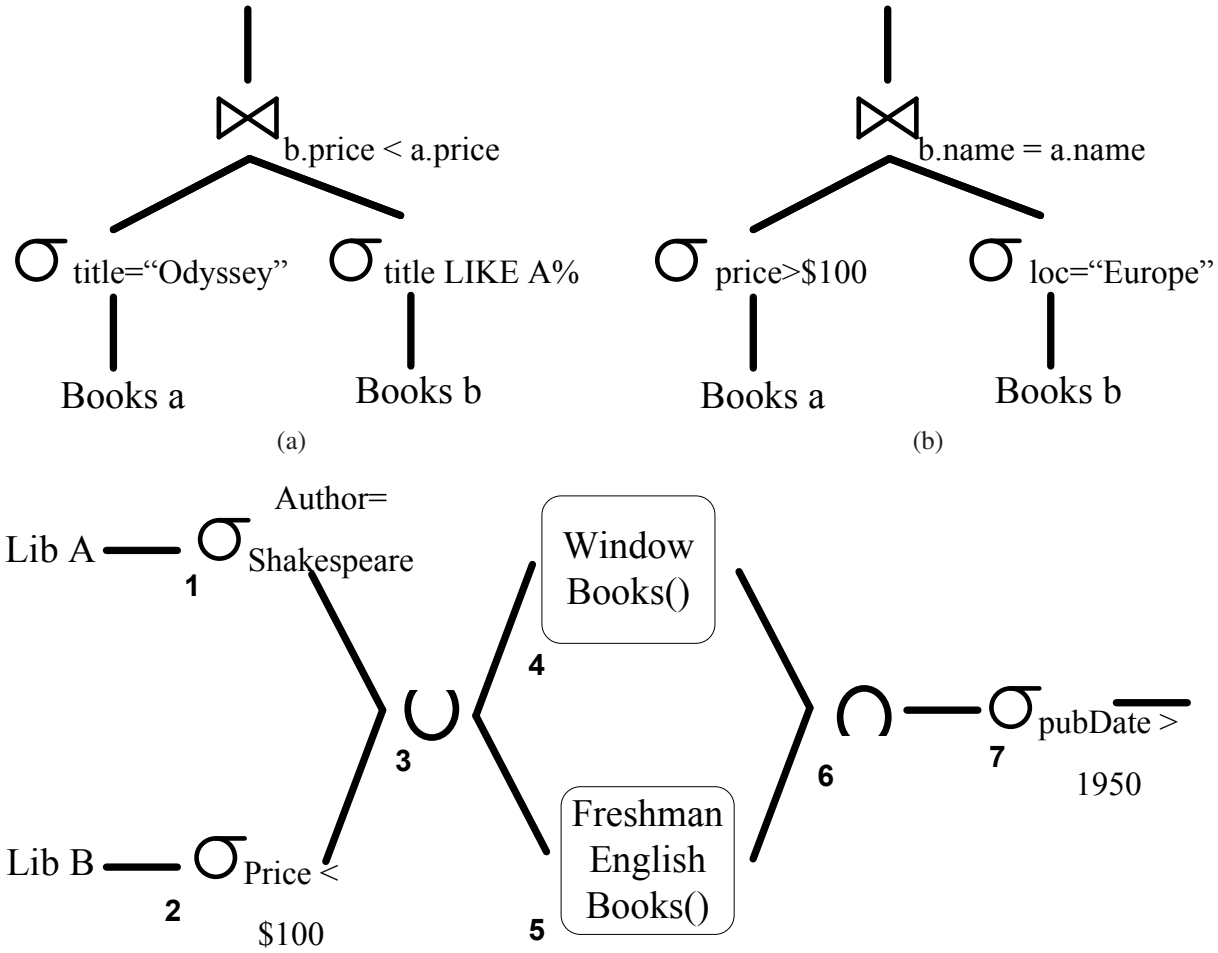


Figure 6.2: A set of query evaluation plans. (a) Queries *Ye Olde Booke Shoppe* for all books priced less than The Odyssey. (b) Queries *Ye Olde Booke Shoppe* for all books priced greater than $100 and written in Europe. (c) Creates a result set with all Shakespeare books in LibA and all books <$100 in LibB, determines the intersection of "Window Books" and "Freshman English Books" in this set and outputs any that were published after 1950. (Operators are numbered for ease of reference.)

ii. the manipulation does not output an Unpicked Successor, and
ii no Unpicked Successors exist further down the workflow.

Notice that whether a MANIPULATIONs is picky or not is dependent upon the Unpicked data item of interest. Consider the query evaluation plan in Figure 6.2(a). The result set for this will contain (Sophocles, "Antigone") If the user wonders why (Virgil, "Aeneid") is not in the result set, the Picky Manipulation is $\bowtie_{b.price<a.price}$. However, if the user wonders why (Hrotsvit, "Basilius") is not in the result set, the Picky Manipulation is $\sigma_{titleLIKEA\%}$.

The discussion and examples thus far have focused on a singular path of MANIPULATIONs. However, this does not need to be the case. The execution of a workflow is a directed acyclic graph (DAG), and can thus have many paths, as in Figure 6.2(c).

Let us walk through the series of operations in Figure 6.2(c), following the data item (Euripides, "Medea"). Operators **2**, **5** and **7** are likely Picky Manipulations. If (Euripides, "Medea") were fed into any of these operators, it would not be part of the output. However, only operator **5** is a Picky Manipulations. Manipulation **7** is not picky since the precursors (Euripides, "Medea") never reach it. Meanwhile, because the intermediate results further down the DAG still contain (Euripides, "Medea") despite **2** not including it as a precursor in the intermediate result set, **2** is not Picky. In the event that operator **1** also excluded (Euripides, "Medea") from the intermediate result set, then the set of Picky Manipulations would be **1** and **2**.

Because the Unpicked is a set of data items, and these data items can behave differently given a workflow, each Unpicked data item taken on its own may have a different Picky Manipulation. However, because we are interested in when the last Unpicked Successor finally gets weeded out of the ultimate result set, we do follow all successors blindly and care if different manipulations weed out particular successors. As long as one Unpicked Successor remains in an intermediate dataset, we have not yet found our Picky Manipulation.

This leads to a formulation of what can be used to answer a user's WHY NOT?

question.

Definition 24. WHY NOT? *Answer:*

Given an Unpicked data item, d, and a result set, R, produced by a series of manipulations, M, upon an input data set I, a WHY NOT? *answer will return the manipulation(s) $m \in M$ at which the last Unpicked successor was excluded from the result set.*

6.2.1 Determining WHY NOT?

Given our definition of Picky Manipulations, and Successors, the algorithms for finding the Picky Manipulations are very straightforward. We use this moment, though, to point out that there are essentially two generic algorithms that will return the same results, but have very different execution times depending on the position of the Picky Manipulation.

Bottom Up

A generic algorithm to find the Picky Manipulation, and thus the answer to WHY NOT? is presented in Algorithm 7. It checks the output of every manipulation beginning at the DAG sources and makes sure there are successors throughout the DAG until finally lighting upon the Picky Manipulation. Finding the Picky Manipulation runs in $O(n * s)$ time where n is the number of manipulations in the DAG and s is the time it takes to determine Unpicked successors.

Top Down

An alternative strategy to find the Picky Manipulation, and thus the answer to WHY NOT? is presented in Algorithm 8. It begins with the outputs of the penultimate manipulation and checks the lineage for every data item. If successors to the Unpicked are found, then the ultimate manipulation is the Picky Manipulation. If no successors are found, the algorithm iteratively checks manipulations in a Breadth First Search manner moving towards the Sources. Finding the Picky Manipulation still runs in $O(n * s)$ time where n is the number of manipulations in the DAG and s is the time it takes to determine Unpicked successors.

Algorithm 7: Answering WHY NOT? Bottom Up.

```
   Input: DAG, M, of manipulations, m
   Input: Input Dataset, I
   Input: Queue, Q, initialized with Source
   Input: Unpicked, U
   Output: Picky Manipulation(s), picky
 1 # Run in Breadth First Search order from Source to Sink  forall m manipulations ∈ Queue Q do
 2 |   Dataset O = ApplyManipulation(m, Input Dataset I);
 3 |   if successorExists(O, U) then
 4 |   |   Q.add(m.children);
 5 |   end
 6 |   else
 7 |   |   flagPossPicky(m);
 8 |   end
 9 end
10 # Run in Depth First Search order from Source to Sink List picky = findTruePickies(Source m);
11                         findTruePickies(manipulation m)  if m.isPossPicky() then
12 |   forall m.children do
13 |   |   List deeperPickies.add(findTruePickies(m.child(i));
14 |   end
15 |   if ) then  deeperPickies.isEmpty(
16 |   |   deeperPickies.add(m);
17 |   end
18 |   return deeperPickies;
19 end
20 else
21 |   forall  m.children do
22 |   |   r
23 |   end
24 |   eturn findTruePickies( m.child(i) );
25 end
```

Top Down vs. Bottom Up and Intermediate Datasets

Obviously, there are advantages to both algorithms. Top Down will find a Picky Manipulation close to the top, Bottom Up will do better with earlier Picky Manipulations. In both Bottom Up and Top Down, we are faced with a distinct choice:

- Keep all intermediate result sets. Find the data items(s) in input and intermediate datasets that could correspond to it.
- Start with initial data items, and re-run, flagging all intermediates that are potential Unpicked data item(s).

There is obviously a trade-off in space and time for these two approaches. This has been explored in [34] in the form of Strong and Input-Only Identity, in which intermediate result sets are stored and only the input datasets are saved respectively. Each of the techniques described can utilize either method of storage. If all intermediates are stored,

Algorithm 8: Answering WHY NOT? Top Down.

```
   Input: DAG, M, of manipulations, m
   Input: Input Dataset, I
   Input: Queue, Q, initialized with Sink
   Input: Unpicked, U
   Output: Picky Manipulation(s), picky
 1 # Run in Breadth First Search order from Source to Sink List picky;
 2 forall m manipulations ∈ Queue Q do
 3     Dataset O = ApplyManipulation(m, Input Dataset I);
 4     if ! successorExists(O, U) then
 5         Q.add(m.children); flagPossPicky(m);
 6     end
 7     else
 8         if (m.parent).isPossPicky() then
 9             picky.add( m.parent );
10         end
11     end
12 end
13 return picky;
```

then we merely search through all input and intermediate data items for possible Unpicked matches. On the other hand, if only input data items are kept, then we must re-run the set of MANIPULATIONS .

6.3 Finding Successors

As seen in the algorithms above, providing WHY NOT? answers hinges upon our ability to find the Unpicked Successors. A brute force method is outlined in Algorithm 9.

Algorithm 9: A brute force algorithm to find the Unpicked Successors of a manipulation.

```
   Input: manipulation, m
   Input: manipulation, m
   Input: Input Dataset, I
   Input: Unpicked, U
   Output: Unpicked Successors, S
 1 List Successors S;
 2 Dataset O = ApplyManipulation(m, Input Dataset I);
 3 forall o data items ∈ Dataset O do
 4     forall u data items ∈ Unpicked U do
 5         if o ∠(m) u then
 6             S.add(o);
 7         end
 8     end
 9 end
10 return S;
```

However, the method presented in Algorithm 9 is $O(OU)$, where O is the size of the output set and U is the size of the Unpicked set. Since we must do this for every

manipulation in the DAG, as shown in Algorithms 7–8, the total time to determine WHY NOT? would be $O(nOU)$ where n is the number of manipulations in the DAG. Using lineage, which would trace through all previous transformations, n, the total time would be $O(n^2OU)$.

Obviously, any shortcuts we can find in determining Unpicked Successors will greatly improve our WHY NOT? efficiency. The bottleneck is finding the lineage for every single manipulation output and checking whether an Unpicked data item is contained in the lineage. The basic method of finding successor is to actually apply the manipulation with all inputs and compute the result. Given y = manip (i1, i2, x, ...), by looking backward to find the lineage of y. Instead of laboriously checking lineage on every data item output from every manipulation, are there properties of manipulations that we can utilize to skip manipulations, or look at only a subset of outputs? What we want is Successor Visibility.

Given an input dataset, I, and output dataset, O, and a manipulation, m, for every data item produced by m, we can write $o_1 = m(p_1, p_2, ...i_1, i_2...)$ where $o_1 \in O$ and $i_1 \in I$.

Definition 25. *$Successor\ Visibility$:*
A manipulation has successor visibility with respect to i_x if we can determine (for all values of i_x and o_y) $O(1)$ time whether there exist o_1, o_2, etc. such that $o_y \overset{m}{\angle} i_x\ \forall i_x$.

In other words, if we can determine the successor of a data item after a manipulation, without performing the computation of the manipulation, or exploring alternative values for i_1, i_2, etc, then there is Successor Visibility. Moreover, a sequence of manipulations can have successor visibility if each manipulation in the sequence has successor visibility with respect to the appropriate (chain-forming) input.

For instance, in the workflow in Figure 6.1(a), the module Select_All_Books takes in a data item from the books table, and produces an exact representation of it as a string. As such, it is possible to correlate the input and output data items without re-running the manipulation. Thus Select_All_Books has Successor Visibility. Indeed, the chain of manipulations from Select_All_Books through Select_Books_<=$20 has successor visibility. Notice that the manipulation Apply_SeasonalCriteria does not have successor

Manipulation	**Visible?**	**Successor, o_x, given i_x**
Projection	Yes	All o_x with matching attribute-value set
Selection	Yes	All o_x with exact attribute-value matches
Rename	No	
Join	Yes	All o_x with matching attribute-value set on the "left" or"right"
Division	No	
MIN or MAX	Yes	o if o contains the data item attribute-value
COUNT, SUM AVERAGE	Yes	o

Table 6.2: Visibility Rules for Relational Operators.

visibility. In Table 6.2, we work through a set of relational operators, and determine whether visibility can be used instead of lineage.

Domain computations, encompassing user defined functions (UDF) in SQL and all modules in a workflow, are potentially problematic in determining WHY NOT?. The implementation problem arises from the potential inability to compute Unpicked Data item successors. In a perfect white-box world, there would be no implementation issue to cloud our model of WHY NOT?. Indeed, in some cases, even the Lineage Method will not work. However, even in this imperfect world, a surprising amount of domain computations have successor visibility. Indeed in [148], tracing lineage through non-relational operators is discussed in detail. For instance, two workflow modules found in [116] are described below. Both have successor visibility.

- Taking a protein identifier, searching SwissProt and returning the protein record.
- Taking the results from a NCBI query and removing duplicates.

In fact, given 100 workflows sampled at random from myExperiments [116], a total of 478 modules exist. Of these, 273 modules satisfy visibility, and 205 do not. Any modules

that did not have a clear mapping to an understandable relationally-visible concept were counted as non-visible. Even with this very harsh criteria, over half of the modules were visible. Moreover, work such as [148] are attempting to make workflow modules more visible by tracing and understanding the underlying operating system calls.

6.4 Evaluation

Is it possible to provide WHY NOT? answers to users? Given a database of tuples, and a complex query can we find the Picky Manipulation(s)? Can we do it in reasonable time? Our findings are positive. In this section, we demonstrate the feasibility of WHY NOT? answers. As discussed in Sections 6.2.1–6.3, there are two methods for finding Picky Manipulation(s), Top Down and Bottom Up, and two methods for finding successors: lineage and visibility. To date, there is only one system that supports lineage as a first class operator, Trio. Trio [13, 106] is built on top of Postgres, and has the ability to trace the lineage of any tuple found in a view back to the original input tuple(s). Since one of the methods proposed for finding successors requires lineage, we used Trio as our backend database. All algorithms were implemented in Java and run on a Dell Windows XP workstation with Celeron(R) CPU at 3.06GHz with 1.96GB RAM and 122GB disk space.

We utilized the Crime queries that are so often used to showcase Trio, since they had complex query evaluation plans that could provide a variety of answers for WHY NOT? questions[2]. Additionally, while we used the classic Crime dataset as a template, we expanded the number of tuples so that it was less of a toy dataset. The total size of the crime database is 4MB. Queries 1–4 produce 7695, 65,319, 140,699 and 5 tuples respectively.

We ran four base queries, performed against the expanded Trio crime dataset. The

[2]Unfortunately, MiMI's keyword interface limited the query complexity in the query logs to an uninteresting set.

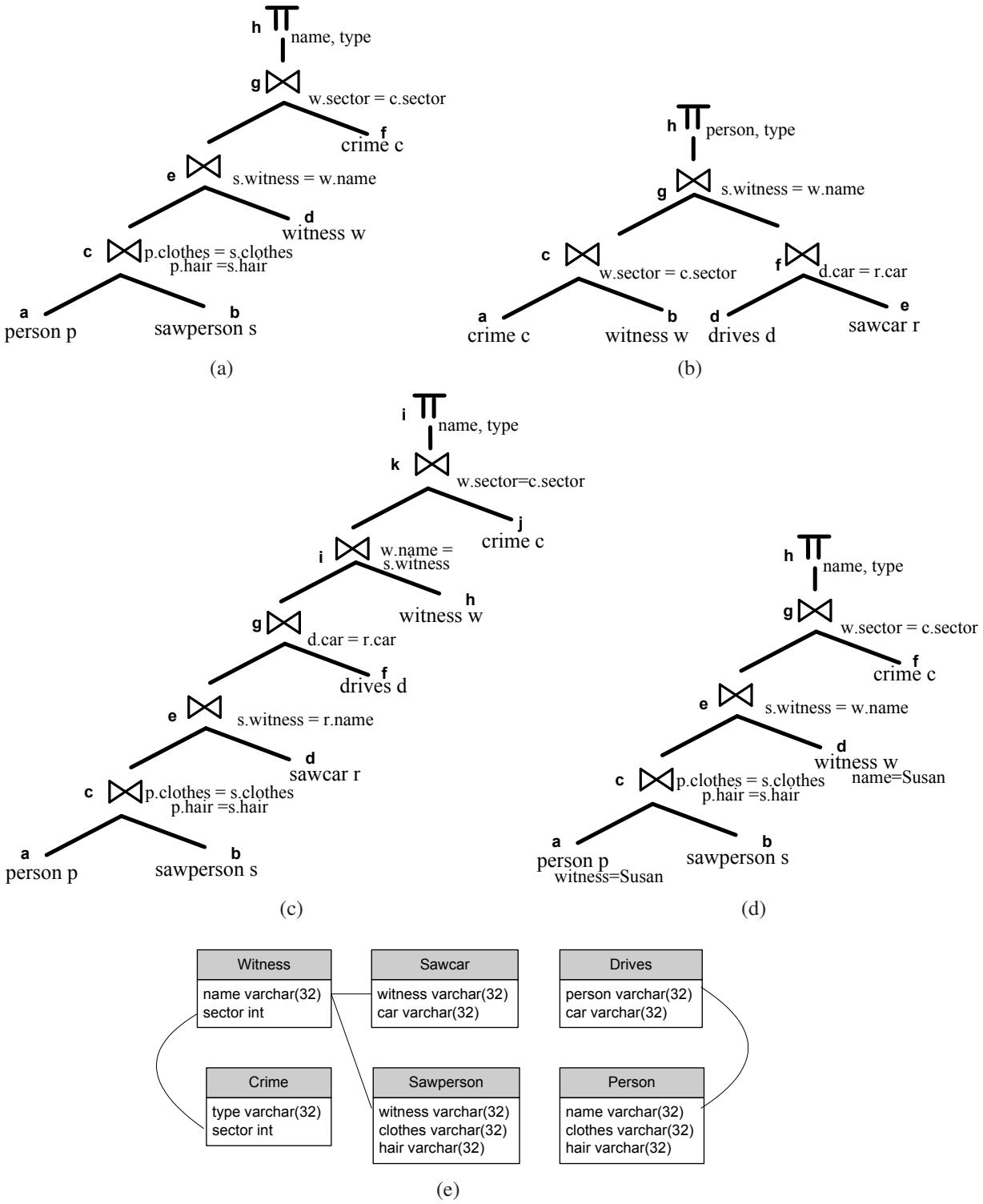


Figure 6.3: (a)-(d) The query evaluation plans for the Crime Queries used (Queries 1–4 respectively). (f) The Trio Crime Schema.

evaluation plans for all four queries were determined using "Explain", and are shown in Figure 6.3. For each query, we then asked a series of WHY NOT? questions by specifying an attribute that existed in the input dataset but not in the final result set. For instance, WHY NOT? "Mary", where Mary could be a potential value for a suspect or witness.

6.4.1 Bottom up vs. Top Down

Figure 6.4 shows the run times to find the Picky Manipulation given an Unpicked set using either the Bottom Up (BU) or Top Down (TD) approach, using lineage to find Unpicked Successors. TD does significantly better than BU for all query evaluation plans except Query 4. Given the nature of the query evaluation plans, this is to be expected. Consider the query evaluation plan for Query 1 in Figure 6.3(a), and the Unpicked data item UP1, "Antigone". There are only five tuples in the entire crime database that can be mapped to an Unpicked with "Antigone": a tuple from the Witness table with Witness.name="Antigone", three tuples from the Sawcar table with Sawperson.witness="Antigone" and one tuple from the Sawperson table with Sawperson="Antigone" (schema for the crime database is in Figure 6.3(e). UP1 does not ever exist in manipulations **a**, or **f** in Figure 6.3(a). However, these are not Picky Manipulations since it does exist in another set of paths, **b**, **c**, **d**, **e**, and **g**. The true Picky Manipulation is **h** since this is where the attribute "Antigone" finally disappears from the result set. As such, the TD algorithm only tests one manipulation, while the BU algorithm must work through all eight. Conversely, Query 4 is highly selective very near the sources. As such, TD must check all eight manipulations. BU does not save very much though. Although the Picky manipulations are very close to the sources, BU still must check four of the manipulations. Thus using the TD algorithm is best when there is low selectivity early in the query evaluation plan. However, while TD may be worse than BU when there is high selectivity early in the evaluation plan, it is not much worse since BU must do almost as much work.

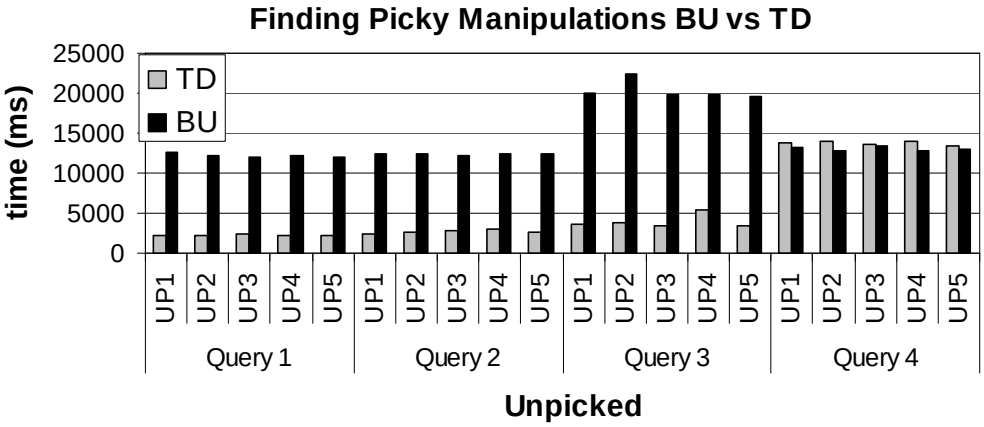


Figure 6.4: Finding the Picky Manipulation for an Unpicked Set using the BU or TD algorithm using Lineage.

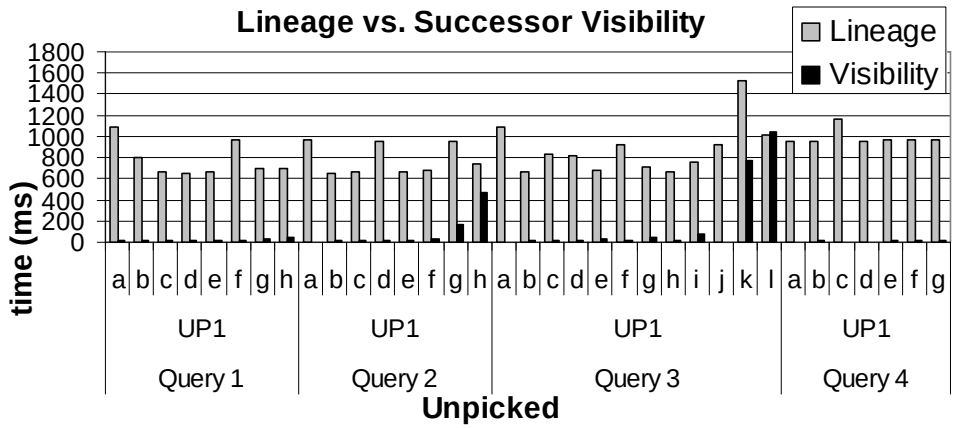
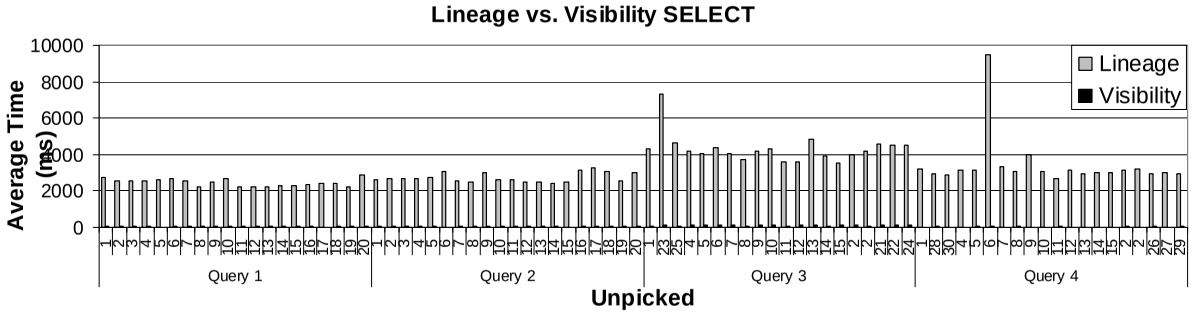
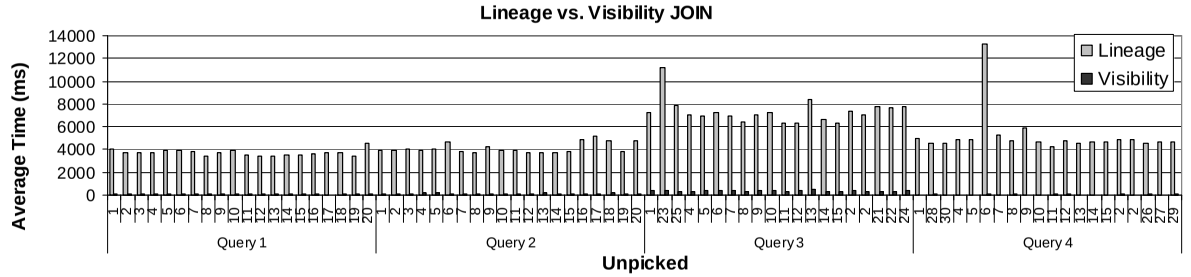


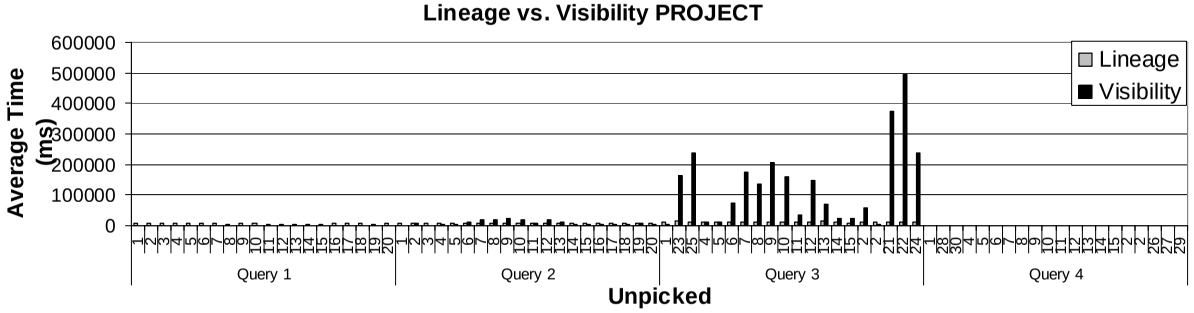
Figure 6.5: Using Lineage vs. Successor Visibility Rules to find the existence of the Unpicked in the manipulation's output.



(a)



(b)



(c)

Figure 6.6: Lineage and Visibility time for each operator within Query 1 for a single and multiple Unpicked set.

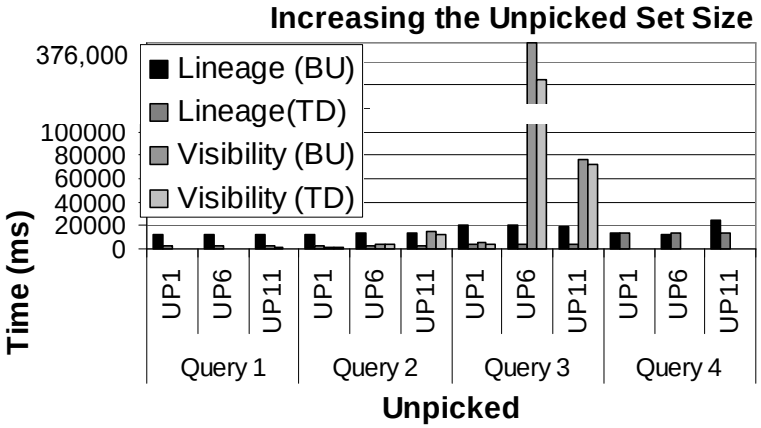


Figure 6.7: Increasing numbers of Unpicked Data Items.

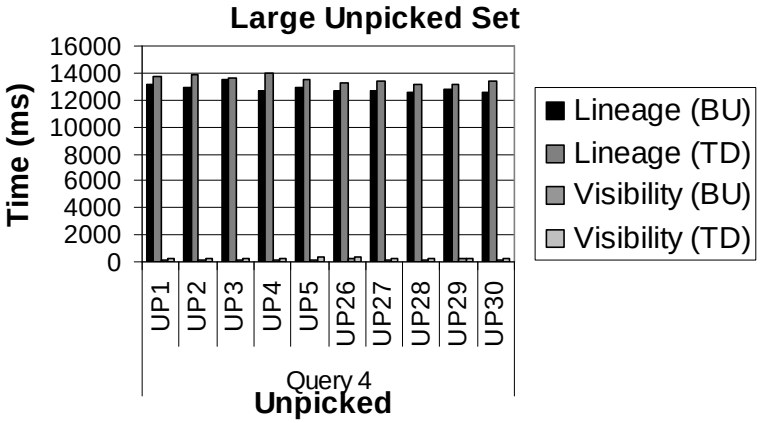


Figure 6.8: A Large Unpicked set.

6.4.2 Lineage vs. Successor Visibility

As discussed in Section 6.3, to find the Picky Manipulation, we must check all appropriate manipulations, and see if any of their outputs can be mapped to an Unpicked data item. Above we show the difference between the TD and BU algorithms. However, as discussed in Section 6.3, we wish to find a set of Successor Visibility Rules that decrease the time of finding an Unpicked Successor to $O(1)$. In Figure 6.5, we show how much savings can be accomplished by using Successor Visibility Rules as opposed to using traditional lineage. The labels on the X-axis map to the manipulations labeled for each Query in Figure 6.3. Overall, using successor visibility rules causes a marked decrease in the amount of time needed to detect Unpicked Successors. Only in Query 3, manipulation **l** is lineage equal to successor visibility. This manipulation (and **k** before it) is dealing with 140,699 tuples, and the data structures used to implement successor visibility finding struggle to keep up.

In Figure 6.6 we show the average time for lineage and visibility for all queries and Unpicked data items run in all experiments broken down by relational operator type. For all selection and join operators, using successor visibility rules does much better than lineage. However, for projections in Query 3, using lineage is better. Remember that Query 3 generates a huge result set, and the structures used for successor visibility begin to thrash at that level of output. Otherwise, using Successor Visibility Rules enables a drastic reduction in time needed to find the Picky Manipulation.

6.4.3 Size of the Unpicked Set

In Figures 6.4–6.5, the attribute looked for specified a small set of tuple in the input data set, usually about 5. As such, there were on average 5 Unpicked data items per WHY NOT? question for each query. Figures 6.7–6.8 shows how the WHY NOT? algorithms fare with a change in the number of Unpicked data items. In Figure 6.7, the first set of queries has only five Unpicked data items, the second has ten and the third has fifteen. In other words, doubling and tripling the number of Unpicked data items in consideration.

Finally, Figure 6.8 shows how the algorithms perform when there is a very large number of Unpicked data items. For clarity we show only the results from Query 4, and compare against WHY NOT? questions that have only a few Unpicked Data items. Unpicked UP1–5 have five Unpicked tuples returned, while Unpicked UP26–30 specify the attributes most found in the database, returning up to 50 Unpicked tuples. Luckily, the number of Unpicked tuples does not affect the overall runtime of either TD or BU algorithms using Lineage or Succinctness.

6.5 Discussion

Unfortunately, Picky Manipulations may mask the true culprit MANIPULATIONS. Consider the query evaluation plan in Figure 6.2(b), which finds all books whose authors are from Europe and are priced greater than $100. Given the input dataset in *Ye Olde Booke Shoppe* and a result set of $\emptyset$, a user may ask "why were no results returned" (a.k.a. Why not anything?). If results are produced from both the selection on the both the books and author table, the join will be called the Picky Manipulation.

This is the correct answer, there are books that cost more than $100, but there are no overlaps with books that were written in Europe. However, because we wish to allow users to explore the underlying queries, workflows and datasets more easily, without reading code or writing extra queries, can we facilitate their exploration?

Using the workflow or base query as a starting point, there are a finite set of possible ways to tweak it without adding operators. For instance, in this case, we can slacken the condition on price, the condition on location or both.

Definition 26. *$Suspect\ Manipulation$*
A MANIPULATION *, m, is a Suspect Manipulation if*
i. it exists prior to a Picky Manipulation, p,
ii. it is commutative with p,
iii. removal of m would result in a precursor of the Unpicked data item's inclusion in the result set of the next Picky Manipulation.

For instance, when wondering why there are no books greater than $100 and written in

Europe, the join in Figure 6.2(b) is Picky. However, if we look at the two manipulations that exist prior to it in the workflow, if one of them is altered slightly, perhaps we will get some data items in the result set. For instance, if we drop the requirement of >$100, the result set will contain all of the books except (, "Epic of Gilgamesh", $150). On the other hand, if we drop the condition of "written in Europe", then (, "Epic of Gilgamesh", $150) is in the result set. Thus, both of the selection manipulations are suspect.

Because we ultimately wish to provide a user with an answer about why a data item is not in the result set, we would like to show the user only one MANIPULATION at a time so that they can process the information. For this to happen, we must have some heuristic with which to rank the Suspect Manipulations. To this end, we establish a notion of Suspect Manipulation Dominance.

Definition 27. *$Suspect\ Manipulations\ Dominance$:*
A Suspect Manipulation, p, is dominant over another Suspect Manipulation, q, if p occurs closer to the source in the DAG.

In the example of finding books from Europe greater than $100, there are three trial versions of the workflow: (1) slacken the price constraint, (2) slacken the Europe constraint and (3) slacken both the price and the Europe constraint. Remember that the user's query was "Why $\emptyset$. All three trial versions will produce results, and therefore include Unpicked Data items in the result set. In two of the trial versions, the $\sigma_{price<\$100}$ is suspect and in two versions the $\sigma_{location=Europe}$ is suspect. Thus, both have equivalent dominance.

The algorithm for finding a suspect manipulation given an Unpicked data item, is a heuristic method and involves loosening the set of constraints placed upon a data item as it moves through a query evaluation plan. For instance, given a selection operator on books with a condition title="Medea", remove the condition and see if the Unpicked data item appears in the result set. By doing this for all manipulations within an evaluation plan, it may be possible to highlight the suspect manipulation.

Obviously this strategy of loosening of constraints is not guaranteed to find the suspect manipulation. Consider a selection of books with less than three authors whose names

begin with “A”. By removing the condition (names begin with “A”) from the author name selection condition, we will be providing MORE names to the count, thereby decreasing the number of books that get considered. However, the space of possible suspects, and their reasons, is infinite, and so we begin with these basic rules to see how many suspects we can find. If a domain computation does not have successor visibility, then the most we can try is remove it from the workflow, assuming the types out and into the previous and next manipulations respectively match. However, if there is successor visibility, and it is possible to have some understanding of the module itself, then we can apply modified versions of the above substitutions.

6.6 A Case Study: MiMI

A scientist searches MiMI [83] through the keyword interface for: “sterol AND organism=‘Homo sapiens”’. A known function of ABC1 is “sterol transporter activity”, so why is it not in the result set? Using the techniques described in this work, it is possible to provide the framework to answer this user’s question. First, a DAG of manipulations, created from the provenance records in MiMI, the query evaluation plan used, and any subsequent manipulations must be created. Figure 6.9 contains the DAG for the user keyword query. Once this is in place, we can utilize either the Top Down (TD) or Bottom Up (BU) algorithms described in Section 6.2.1. A slightly tricky step arises when determining Successor Visibility. Lineage requires white-box manipulations such as relational operators. This requires that we use Successor Visibility Rules for all manipulations outside of the relational part of MiMI, manipulations **a**, **b**, **c** and **h**. As long as a Successor Visibility Rule exists for each of these manipulations, we can find the Picky Manipulation that excluded ABC1 from the result set. In this case, it was Manipulation **h**, that only presents the top 100 results.

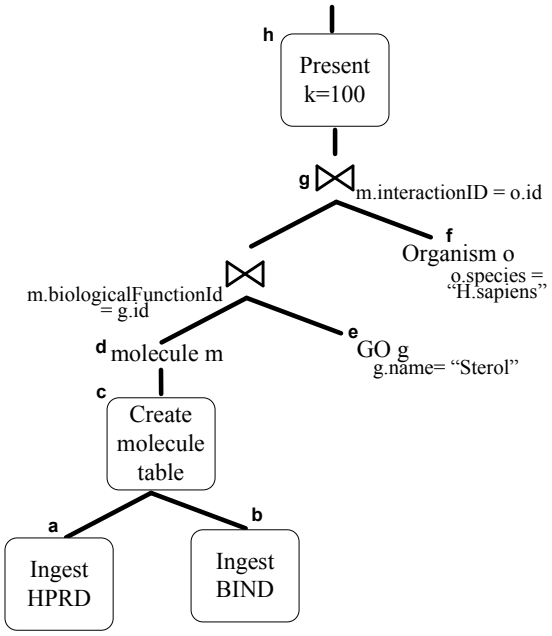


Figure 6.9: The (simplified) DAG of manipulations for a MiMI query through the keyword interface.

6.7 Conclusions

In this work, we outline a new problem facing users data whose access is restricted. When users are unable to sift through the data themselves, it is impossible to discover why a data item is not in the result set. Is it not in the input datasets? Is some manipulation between the input and the user discarding it? Etc. We provide a framework that allows users to ask WHY NOT? questions about data items not in the result set (Unpicked). Additionally, we create a model that allows us to pinpoint where the Unpicked data item was discarded from the final result set.

We implement the model using two different algorithms for finding the manipulation of interest, and two different methods for finding a data item's successor. We show how these methods compare using a well-known set of queries.

CHAPTER VII

RELATED WORK

Provenance assists scientists in interpreting and reproducing results, understanding the experiment and reasoning that produced a result, assess data quality, and track where and who the data inputs came from. Because of the experimental nature of traditional science, this assistance is vital for understanding the veracity of a piece of data. Consider Table 7.1, reproduced from [127]. The last three lines show the number of protein interactions found with two, three and four different techniques, respectively. There is a large dropoff in the number of confirmed interactions across different experimental methods. It should be noted that the correctness of the experimental evaluation for all of these experiments is not in question. Different techniques just give different answers. A scientist must understand where the data came from, and who reported it in order to fully understand its significance. In the past, lab notebooks contained all of this information. However, the volume of scientific data has increased, and many experiments are now performed in-silico, or with heavy post-experimental computation. This information is still essential to understand the data, but is un-capturable in a lab notebook. Provenance can provide this information. Unfortunately, biological and other scientific databases have dealt with provenance tracking in *ad hoc* ways. The Saccaromyces Genome Database [40] uses triggers to store records of updates to the database, but this only provides the history of local changes. In most cases the provenance of copied data is recorded manually. For example, in the Nuclear Protein Database, links to the PubMed bibliographic database or to other protein databases such as Entrez or UniProt are entered manually by the curator

Experimental method category	Number of interacting pairs
All: All methods	9347
A: Small scale Y2H	1861
A0: GY2H Uetz et al. (published results)	956
A1: GY2H Uetz et al. (unpublished results)	516
A2: GY2H Ito et al. (core)	798
A3: GY2H Ito et al. (all)	3655
B: Physical methods	71
C: Genetic methods	1052
D1: Biochemical, in vitro	614
D2: Biochemical, chromatography	648
E1: Immunological, direct	1025
E2: Immunological, indirect	34
2M: Two different methods	2360
3M: Three different methods	1212
4M: Four different methods	570

Table 7.1: From [127], a comparison of the number of interacting pairs found via different methods. Of particular interest are the last three rows that show many interactions are not found across experimental methods.

alongside the relevant data.

Currently no standard exists for provenance, although the Open Provenance Model [110] is attempting to create a base definition. The First and Second Provenance Challenges [101] have been integral in forcing people to define and create this standard. At a high level, there are three general types of provenance:

1. Process: This includes both "Retrospective" and "Prospective". Retrospective provenance includes information such as invocation records of runtime and resources. Prospective provenance includes workflows and is a statement about how to produce the data

2. Lineage: This is a causality graph of relationships between data and computations.

3. Annotations: This is a user defined piece of information that adds information to and travels with a piece of data.

	Database	**Workflow**
Data	Set of tuples	Set of non-uniform objects
Operators	Relational Operators	Black boxes
Prov. Shows	Derivation for a data item in the database	Derivation for a data product

Table 7.2: Database vs. Workflow Provenance.

Moreover, there are two main categories of provenance systems: Database and Workflow. Table 7.2 summarizes the main differences between the two groups. While the title of this book is "Incorporating provenance into Database Systems", the word Database does not mean the same as in "Database Provenance". In this work, we have explored issues involved in making database systems, including outside workflows that utilize databases, use provenance in an efficient and meaningful manner. The term "Database Provenance" implies provenance solely within a database. Finally, it should be noted that there is a large amount of overlap among systems, and which type of provenance and category of system they belong to. For instance, the Earth System Science Workbench (ESSW) [66] allows users to attach annotations. It also keeps a provenance record of all processes run, and is organized in such a way that the lineage of a data item can be traced. Work such as [34, 33] has explored problems in the grey areas between true database provenance systems, and workflow systems. Overviews of different provenance system types can be found in [18, 28, 29, 52, 70, 124, 132].

7.1 Provenance in Databases

7.1.1 Lazy vs Eager

There are two possible methods for maintaining provenance in a database: eager and lazy [132]. Lazy provenance is computed after runtime, only when asked for. Thus it is less storage intensive, and is used by [29, 30, 48, 49, 50]. Lazy provenance requires either transformations to be reversible, or for a user to define the reverse transformation. [48, 49, 50] show the requirements for each relational operator and the ability to

compute provenance after the fact. On the other hand, eager provenance as found in [15, 21, 22, 64, 71, 139, 144], stores annotations and dependencies as they are created at runtime. This removes the problem of tuple tracing through views since annotations are carried with each piece of data. In this work, we were focused mainly on eager provenance, although the concept of visibility in Chapter VI builds on the formalisms presented in the classic lazy literature [29, 30, 48, 49, 50].

7.1.2 Lineage

In its purest sense, *lineage* is the origin of a data item [13, 14, 17, 29, 30, 48, 49, 50, 106, 140, 148]. A basic example would be:

Query 2.

CREATE TABLE standardWt AS
*SELECT **
FROM HPRD
WHERE molecularWt = 100,000Da

For Query 2, the lineage of any tuple in the standardWt table would be it's corresponding tuple in HPRD. Things can get more complicated, though. For example, given Query 1 from Chapter V, the lineage of any tuple produced would be the entire set of tuples in HPRD, since the AVERAGE function touches all tuples in the HPRD table. Along similar lines, but applied to data transformations, [138] allow users to query data transformations.

In [29], the distinction between *why* and *where* provenance is presented. In this work, *why* is the lineage of a tuple, while *where* is actual attribute a particular value came from. [29] is often cited as a defining statement of provenance concepts, even though the words used have changed. Later work, [23], uses the properties of query languages to answer provenance questions.

Trio Most database provenance remains in the realms of theoretical examination. However, in the case of lineage, Trio [13, 14, 106, 140] is a functioning database built upon Postgres in which lineage is a first class citizen. Trio allows multiple uncertain values

to be listed for any attribute. When queries are performed upon this uncertain information, lineage is used as a support and explanation for uncertain entries in the result.

7.1.3 Annotation

Metadata often is used interchangeably with provenance. Annotation is a version of metadata that can be used to explain properties of the data. Annotation is placed upon attributes within a database and is propagated through queries [15, 42, 131]. DBNotes [42] puts this work into practice. An early version of annotation can be found in [88] which creates source attribution to allow easy querying of merged documents. Additionally, [46, 68] have studied how annotations move through views.

Recently [64, 71] have used annotations and commutative semi-rings within a database system. By altering the semi-ring's operators and set K, they can do lots of fancy things, such as compute lineage, or restrict access to attributes.

7.2 Provenance in Workflows

While workflow systems have been the de rigour in the business management community, their benefits, and domain specific problems are currently being actively explored by the scientific community. Workflow systems [3, 4, 19, 32, 52, 62, 75, 86, 94, 97, 98, 101, 125, 109, 151, 150] are being utilized as a method for representing and managing data intensive, complex computations. Using a graphic layout, these tools help scientists conceptualize programmatic tasks without writing actual programs. Not only do workflow systems assist scientists execute complex computations, they also enable automation, reproducibility and result sharing, by keeping detailed records of the processes run. Indeed, some commercial workflow-like systems exist, such as Mac OS X Automator, Microsoft Windows Workflow Foundation, and Yahoo! Pipes. Figure 7.1, from [102], shows an overview of the workflow systems who competed in the Provenance Challenge [101].

	Redux	Mindswap	Karma	JP	myGrid	VisTrails	ES3	ZOOM	RWS	COMAD	PASS	SDG	NCSD2K	NCSCI	VDL	OPA	Wings/Pegasus
1. Characteristics of Provenance Systems																	
1.1 Execution Environmer	Workflow system	Web	Workflow system	Workflow syst and job submission	Workflow system	Workflow system	Operating system	Workflow system	Workflow system	Workflow system	Operating system	Workflow system	Visual Program. Env	Workflow system	Workflow system	Technolog independe	Workflow system
(actual system)	WWF		XBaya and BPEL	EGEE glite (inc Condor Dagman and JDL)	Taverna and Xscufl	VisTrails system	IDL and Bash	Technology independent	Kepler, Ptolemy	Kepler, Ptolemy	Linux	Kepler, Ptolemy	D2K	Cyber Integrator	VDS		Wings and Pegasus
1.2 Execution Environment (for the challenge)				EGEE VOCE VO		VisTrails system and shell		SQL scripts			Shell script				Grid	Java	Grid
1.3 Provenance Representation	RDBMS	OWL	XML View and RDBMS	key-value pairs in RDBMS	RDF	XML View and RDBMS	XML view	RDBMS	Internal	Internal and XML view	Internal	RDF (external) and SAM/Web (internal)	RDF	RDF	RDBMS	Internal and XML view	OWL and RDBMS
1.4 Query Language	SQL	SPARQL	Karma query API + SQL	JPIS + JPPS queries + Perl	ProQA + RDQL +TriQL	vtPQL and SQL	ES3 Queries	SQL + transitive closure	Internal Graph QL	Internal Graph QL	Custom	Semantic Extended DASL (SEDASL)	iTQL	iTQL	SQL +XML Querying + text querying	XQuery, PQuery + Java	SPARQL + SQL
1.5 Research Emphasis	E/R/S/Q	R/S/Q	E/R	R/S/Q	E/R/Q	E/R/S/Q	R/S/Q	S/Q	E/R/Q	E/R/Q	E/R/S	R/S/Q	Q	Q	E/R/S	R/Q/S	E/R/S/Q
1.6 Challenge Implementation	Run	Run	Run	Run	Simulated	Run	Run	Simulated	Partial	Partial	Run	Partial	Run	Run	Run	Partial	Run
2. Properties of Provenance Representation																	
2.1 Includes workflow representation	yes	no	no	yes	yes	yes	no	yes	yes	yes	no	yes	no	no	yes	no	yes
2.2 Data Derivation vs Causal Flow of Events	E	D E	D E	E	D	D E	D E	D E	D E	D E	D E	D	D	D	D	D E	D
2.3 Arbitrary annotations in scope/implemented	(+AS/+AI)	(+AS/+AI)	(-AS/+AI)	(+AS/+AI)	(+AS/+AI)	(+AS/+AI)	(+AS/-AI)	(-AS/+AI)	(+AS/-AI)	(+AS/-AI)	(+AS/+AI)	(+AS/+AI)	(+AS/+AI)	(+AS/+AI)	(+AS/+AI)	(-AS/+AI)	(+AS/+AI)
2.4 Time supported/required	(+TS/+TR)	(+TS/+TR)	(+TS/+TR)	(+TS/-TR)	(+TS/+TR)	(+TS/+TR)	(+TS/+TR)	(+TS/+TR)	(+TS/-TR)	(-TS/-TR)	(+TS/-TR)	(+TS/-TR)	(+TS/-TR)	(+TS/-TR)	(+TS/+TR)	(+TS/-TR)	(+TS/+TR)
2.5 Naming required (if yes, then what)	keys for ports and data	no	GUIDs for data, services, workflows	no	lsids for all data	no	UUID assigned to every provenance relevant object	Unique ids assigned to every provenance relevant object	yes for tokens	yes for collections and their members	yes, unique ids for files per volume	unique ids (urls or lsid) for data to be tracked	URIs	URIs	logical file names	no	logical file names and file domain metadata attributes
2.6 Tracked data, and granularity	port level (I/O) but not their contents	all data	Any GUID assignable data	arbitrary	I/O of any type, but not their contents	changes to workflow specification and execution log	file or process	I/O objects of any type (data items and collections)	uniform streams of tokens	collections hierachica trees of het. data	file or process	I/O of any type, but not their contents	any relationshi (content not stored)	any relationshi (content not stored)	file	anything	files and nested file collections
2.7 Abstraction mechanims	layered provenanc model	grouping on files, processes, and workflows	grouping on data products, processes, and nested workflows		aspects	layered provenance model	script or job	user view of composite steps			N/A				data sets and nested procedure	tracers	reusable templates - workflow instances - execution details

Figure 7.1: From [102], an overview of existing workflow systems that capture and store provenance information.

7.2.1 Method of Provenance Capture

Automatic capture of provenance information is currently handled in three distinct ways:

1. Embedded in a workflow execution environment [3, 4, 19, 32, 52, 62, 75, 86, 94, 97, 98, 101, 125, 109, 151, 150].

2. Via the operating system [67, 104, 122].

3. Using instruments and services [51, 73, 74, 75, 76, 105, 125].

In the case of [3, 4, 19, 32, 52, 62, 75, 86, 94, 97, 98, 101, 125, 109, 151, 150], a user manipulates their data through a workflow system, and the workflow framework quietly stores provenance information. Systems such as PASS [104, 122] and ES3 [67] try to free the user from being forced to work within a workflow environment by using information from the operating system to track what processes and files are being used. Finally [51, 73, 74, 75, 76, 105, 125] keep track of provenance from processes that are constantly churning, such as signal readings or equipment monitoring.

7.2.2 Systems Used for Scientific Exploration

Many systems have been used in conjunction with scientific exploration. A few highlights include:

1. **Chimera** [61, 62]: Sloan Sky Survey - galaxy cluster finding [6].

2. **Karma** [125, 126]: Weather forecasting [20].

3. **Kepler** [4, 19]: Bioinformatics [93].

4. **Swift** [151]: Aphasia and other Medical Diseases [129].

5. **PASOA/PReServ** [75, 94]: Protein Compressibility [74].

6. **Pegasus** [86]: Modeling earthquakes [56].

7. **Taverna** [52, 97, 98, 109, 149, 150]: Proteomics [99] and Molecular Biology Applications [116].

8. **VisTrails** [32, 121, 120]: Oceanographic Exploration [10] and Radiation Oncology [5].

7.3 Annotation Provenance

Annotation provenance is no longer the rage. However, it is an important form of provenance, and can enable scientists to work more productively with their data. Annotation systems can also track the lineage of data, and any processes applied to it. Much of the reason annotation systems are no longer being built is because workflow systems such as Taverna [52, 97, 98, 109, 149, 150] and VisTrails [12, 32, 120, 121] are allowing users to annotate the data within a workflow environment. However, two older systems should be mentioned as some of the earliest annotation and lineage work: Collaboratory for Multi-Scale Chemical Science (CMCS) [108, 111] and the Earth System Science Workbench (ESSW) [66].

7.4 Logging, Archiving and Version Control

Our approach to provenance overlaps with several other well-studied areas, including transaction logging, data availability, schema evolution, archiving, file synchronization, and version control. In the rest of this section we discuss the relationship between our work and these areas.

Logging Many database and file systems use *transaction logging* or *journaling* in order to provide crash recovery. Such logs store detailed information about update operations applied to the database. This information is necessary to undo the effects of any transactions that had not committed at the time of a crash. Since provenance tracking is similar in some respects to logging, one might argue that provenance tracking

is redundant or unnecessary in a database system that already performs logging. However, logging serves a much different purpose, and transaction logs do not provide as much information as provenance; so, to achieve the same effect, it would be necessary to add extra instrumentation that stores additional information to the logging system. In our opinion, this would be a mistake: such application-level code and data has no place in a system-critical mechanism.

Data availability One natural question is whether it makes sense to retain provenance information if the original data source becomes unavailable. The answer is an emphatic *yes*: such provenance information is impossible to reproduce, so potentially priceless. Provenance information for "lost" data can even help us recover the lost data from copies. For example, suppose two databases T_1 and T_2 are constructed using data from S, that the construction process is recorded by provenance stores P_1, P_2, and that later S disappears. We can still be fairly certain about the contents of S, since we can use the provenance records of T_1 and T_2 to partially reconstruct S. Even if T_1 and T_2 disagree about the contents of S (which could easily happen due to changes to S or due to errors in T_1, T_2, P_1 or P_2), this information may be better than nothing.

Version control, archiving, and synchronization Version control [92], archiving [27, 133], change management [2] and file synchronization [65] are closely related to our approach to provenance, but they do not address the same problem. Such techniques aim to preserve or reconcile the states of the data as it evolves over time, but they tell us only how the versions differ, not how the changes were *actually* performed. Moreover, these systems typically do not track changes that span multiple systems. Conversely, provenance identifies the source of information in the current version, but gives us no guarantee that the cited information has been preserved. The information may be in a database that has been updated since the data was extracted, and if the database has not been archived, there will be no confirming evidence for the information that has been extracted. We believe that

both provenance recording and archiving are necessary in order to preserve completely the "scientific record."

7.5 Compression

The Factorization Algorithms in Chapter IV, [35], are similar to work in workflow specification from process logs [137], which attempts to create an accurate workflow, with an eye to processes, but our work attempts to understand and reduce the size of arguments found in provenance files. Compiler optimization [43, 59] has also similarities to the provenance reduction studied here.

Other XML compression work has similarities to the provenance compression focused upon here. In particular, [25, 31] describe finding common paths within XML documents to reduce the overall space. Not only can these techniques be applied to the *reduced* provenance store we create, we are able to provide a further reduction by finding common subtrees despite dissimilar arguments. Additionally, [90, 96, 134] propose general XML compressors that can also be applied to the provenance store we create. XML compression [90] creates a smaller store than the reduction provided in this work. However, the XML compression systems do not result in a store that can be queried with an uncompressed dataset via a standard query language. While XGRIND [134] does support exact and substring querying of the compressed store, it does not support joins and thus cannot build relationships among data and provenance elements; specifically, there is a lack of support for value or structural joins between provenance pointers and the provenance store. Luckily, these compression techniques can be further applied to the reduced provenance store we create.

Finally, [11, 91, 93, 115] describe and implement scientific experiment management systems. These allow scientists to change a parameter of any manipulation and re-execute all downstream processes. We show that our reduction techniques can work with provenance generated within a workflow framework by our use of the Karma

and PReServ provenance stores. The Factorization strategies presented in Chapter IV may not greatly affect the fairly normalized provenance collected by workflow systems [3, 4, 19, 32, 52, 62, 75, 86, 94, 97, 98, 101, 125, 109, 151, 150]. However, the Inheritance strategies can still be applied. Systems such as [133], allow users to adjust the granularity at which versioning takes place to reduce the storage space needed. Unfortunately, if approaches such as this are applied to provenance, valuable information could be lost.

7.6 Provenance Visualization and Usability

Others have looked at making provenance more usable. For instance, ZOOM [45, 44] shows users abstractions of sub-workflows. The user is then able to glean a general understanding of the workflow, and poke deeper into abstracted layers for more information. The Drill Down querying presented in Chapter V is very similar, but instead of relying upon a workflow specification to determine abstractions, we use a measure of how much information is in the provenance store to present different layers of information to the user. Others such as [41] provide an explorer-like interface, while [121] return a visual abstraction of the provenance presented. Finally [148] is attempting to store enough operating-level information to explain away provenance black boxes and make them transparent to the user by describing the underlying system calls.

This work attempts to answer user queries about results that are created via processes and datasets that are opaque to the user. A few systems outside of the provenance community are also attempting to answer user questions about the data presented. In particular, we would like to mention [87, 107] who attempt to do this for programmatic interference. For example, “Why did MSWord capitalize this word?”. While the details of how they accomplish this task are completely different from ours, the underlying problem remains the same: users are confronted daily with processes and data that they do not understand. Finally, work such as [95, 97, 98, 143] attempts to assist users trust experimental data by using provenance information to weed out bad data or assign a

credibility level to existing data.

7.7 Finding Successors and Picky Manipulations

This work draws heavily upon the formalisms and concepts set out by [50, 48, 49], and uses the implementation of them in Trio [13, 106, 140]. Moreover, work such as [148], is attempting to extend the ability of tracing lineage through non-relational operators by recording system-level calls and recording what happens for each input despite not being able to see in the workflow-module black box.

Several groups are also beginning to think about why items are not in the result set. For instance, [80] defines the concept of the provenance of non-answers. A non-answer, is very similar to our concept of an Unpicked Data item. However, instead of attempting to find the manipulation that excluded it from the result set, [80] look for the *attribute* within the Unpicked that caused it to be excluded from the result set. By substituting an "always true" value for each attribute in the tuple until it is included in the result set, they can can pinpoint the attribute(s) responsible. Additionally, [112] looks at data publishing security, and allows users to verify that their query results are complete as well as authentic. While their motivation and methods are security focused, they too are attempting to give users more control and ability to probe the underlying data.

CHAPTER VIII

CONCLUSIONS

The importance of maintaining provenance has been widely recognized, particularly with respect to highly-manipulated data. Currently there are two main provenance research avenues: provenance generated within workflow frameworks, and provenance within a contained relational database. Workflow provenance allows workflow re-execution, and can offer some explanation of results. Within relational databases, knowledge of SQL queries and relational operators is used to express what happened to a tuple. There is a disconnect between these two areas of provenance research. Techniques that work in relational databases cannot be applied to workflow systems because of heterogeneous data types and black-box operators. Meanwhile, the real-life utility of workflow systems has not been extended to database provenance. In the gap between provenance in workflow systems and databases, there are myriads of systems that need provenance. For instance, when creating a new dataset, like MiMI, using several sources and processes, or building an algorithm that generates sequence alignments, like miBlast. These hybrid systems cannot be mashed into a workflow framework and do not solely exist within a database. This work solves issues that block provenance usage in hybrid systems. In particular, we look at capturing, storing, and using provenance information outside of workflow and database provenance systems.

We have constructed a database of protein interactions (MiMI), which is heavily used by biomedical scientists, by manipulating and integrating data from several popular biological sources. The provenance stored provides key information for assisting

researchers in understanding and trusting the data. In this book, we describe several desiderata for a practical provenance system, based on our experience from this system. We discussed the challenges that these requirements present, and outlined solutions to several of these challenges that we have implemented.

A major challenge is how to create provenance records of manual altering of database records. Curated databases in bioinformatics and other disciplines are the result of a great deal of manual annotation, correction and transfer of data from other sources. Provenance information concerning the creation, attribution, or version history of such data is crucial for assessing its integrity and scientific value. General purpose database systems provide little support for tracking provenance, especially when data moves among databases. In this book, we investigated general-purpose techniques for recording provenance for data that is copied among databases. We described an approach in which we track the user's actions while browsing source databases and copying data into a curated database, in order to record the user's actions in a convenient, queriable form. We presented an implementation of this technique and used it to evaluate the feasibility of database support for provenance management. Our experiments showed that although the overhead of a naïve approach is fairly high, it can be decreased to an acceptable level using simple optimizations.

Because more provenance is being captured, there is an increasing need to store and manage provenance for each data item stored in a database, describing exactly where it came from, and what manipulations have been applied to it. Storage of the complete provenance of each data item can become prohibitively expensive. In this book, we identified important properties of provenance that can be used to considerably reduce the amount of storage required. We identified three different techniques: a family of factorization processes and two methods based on inheritance, to decrease the amount of storage required for provenance. We used the techniques described to significantly reduce the provenance storage costs associated with constructing MiMI [83], a warehouse of data

regarding protein interactions, as well as two provenance stores, Karma [125] and PReServ [75], produced through workflow execution. In these real provenance sets, we were able to reduce the size of the provenance by up to a factor of 20. Additionally, we showed that this reduced store can be queried efficiently and further that incremental changes can be made inexpensively.

As scientific data becomes increasingly processed and manipulated, provenance information is essential to help end users understand it's derivation, significance and veracity. Current provenance stores, while storing adequate information to automatically recreate a dataset, are often unable to express in a human-understandable way what has happened to the data. They contain both too much and too little information to be valuable to a human. Because the data is manipulated via a series of black boxes, it is often impossible for a human to understand what happened to the data, without utilizing a euphoria-causing stimulant like qat[1]. We highlighted the missing information that can assist user understanding. Unfortunately, provenance information is already very complex and difficult for a user to comprehend, which can be exacerbated by adding the extra information needed for deeper black-box understanding. In order to alleviate this, we developed a model of provenance answers that assists the user by allowing the user to decide on the fly what information should be presented. We showed the benefits of this model to users of a real scientific dataset with provenance information. Finally, we showed that the structures and information needed for this model are a negligible addition.

Finally, while relational and workflow provenance systems are geared toward explaining why a data item is in the result set, they cannot answer why data items are not in a result set. We introduced the concept of WHY NOT? queries: the ability to ask why data items are not in the result set. We allow researchers to specify data items

---

[1]The last "Q" word with two or three letters (Qat, Qi, Qis, Qua, Suq) allowed in Scrabble™. This work is dedicated to David Gammack with love. Thank you for the countless hours spent with me, especially the ones laughing and playing.

they are looking for that are not in the result set, and determine which manipulation was responsible for weeding it out. We developed a model for answers to WHY NOT? queries, and described two algorithms for finding the manipulation that discarded the data item of interest. Moreover, we worked through two different methods for tracing the discarded data item that can be used with either algorithm. Using our algorithms, it is feasible for users to find the manipulation that excluded the data item of interest, and can eliminate the need for exhausting debugging.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] D. Addington. *Hand in the Bush: The Fine Art of Vaginal Fisting*. Greenery Press, 1998.

[2] D. Agarwal, D. Barman, D. Gunopulos, N. Young, F. Korn, and D. Srivastava. Efficient and effective explanation of change in hierarchical summaries. In *KDD*, pages 6–15, 2007.

[3] G. Alonso and C. Hagen. Geo-opera: Workflow concepts for spatial processes. In *Symposium on Large Spatial Databases*, pages 238–258, 1997.

[4] I. Altintas, O. Barney, and E. Jaeger-Frank. *Provenance Collection Support in the Kepler Scientific Workflow System*, pages 118–132. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Provenance and Annotation of Data edition, 2006.

[5] E. W. Anderson, S. P. Callahan, G. T. Y. Chen, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Visualization in radiation oncology: Towards replacing the laboratory notebook. Technical report, SCI Institute Technical Report, No. UUSCI-2006-17, University of Utah, 2006.

[6] J. Annis, Y. Zhao, J.-S. Vöckler, M. Wilde, S. Kent, and I. T. Foster. Applying chimera virtual data concepts to cluster finding in the sloan sky survey. In *SC*, pages 1–14, 2002.

[7] G. Bader, D. Betel, and C. W. Hogue. BIND: the biomolecule interaction network database. *Nucleic Acids Research*, 31(1):248–250, 2003.

[8] G. Bader, I. Donaldson, C. Wolting, B. F. F. Ouellette, T. Pawson, and C. W. V. Hogue. BIND–the biomolecular interaction network database. *Nucleic Acids Research*, 29(1):242–245, 2001.

[9] G. Bader and C. W. V. Hogue. BIND–a data specification for storing and describing biomolecular interactions, molecular complexes and pathways. *Bioinformatics*, 16(5):465–477, 2000.

[10] A. Baptista, B. Howe, J. Freire, D. Maier, , and C. T. Silva. Scientific exploration in the era of ocean observatories. *IEEE Computing in Science & Engineering*, 10:53–58, 2008.

[11] R. S. Barga and L. A. Digiampietri. Automatic capture and efficient storage of escience experiment provenance. In *Concurrency and Computation: Practice and Experience*, 2007.

[12] L. Bavoil, S. Callahan, P. Crossno, J. Freire, C. Scheidegger, C. Silva, and H. Vo. VisTrails: Enabling interactive multiple-view visualizations. In *IEEE Visualization*, pages 18–26, 2005.

[13] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB Seoul, Korea*, pages 953–964, 2006.

[14] O. Benjelloun, A. D. Sarma, C. Hayworth, and J. Widom. An introduction to ULDBs and the Trio system. *IEEE Data Eng. Bull.*, 29(1):5–16, 2006.

[15] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[16] A. Birkland and G. Yona. The BIOZON database: a hub of heterogeneous biological data. *Nucleic Acids Research*, 34:D235–242, 2006.

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

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

[19] S. Bowers, T. McPhillips, M. Wu, and B. Ludscher. Project histories: Managing data provenance across collection-oriented scientific workflow runs. In *DILS*, pages 27–29, 2007.

[20] K. A. Brewster, D. B. Weber, S. Marru, K. W. Thomas, D. Gannon, K. Droegemeier, J. Alameda, and S. J. Weiss. On-Demand Severe Weather Forecasts Using TeraGrid via the LEAD Portal. *TeraGrid Conference*, June 2008.

[21] P. Buneman, A. Chapman, and J. Cheney. Provenance management in curated databases. In *ACM SIGMOD*, pages 539–550, June 2006.

[22] P. Buneman, A. Chapman, J. Cheney, and S. Vansummeren. *A Provenance Model for Manually Curated Data*, pages 162–170. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Provenance and Annotation of Data edition, 2006.

[23] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *ICDT*, pages 209–223, 2007.

[24] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for XML. *Computer Networks*, 39(5), August 2002.

[25] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, 2003.

[26] P. Buneman, S. Khanna, K. Tajima, and W.-C. Tan. Archiving scientific data. In *SIGMOD*, pages 1–12, June 2002.

[27] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. *ACM Trans. Database Syst.*, 29:2–42, 2004.

[28] P. Buneman, S. Khanna, and W.-C. Tan. Data provenance: Some basic issues. In *FSTTCS*, pages 87–93, 2000.

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

[30] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

[31] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML documents. In *DBLP*, pages 199–216, 2005.

[32] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, and C. T. S. H. T. Vo. VisTrails: Visualization meets data management. In *SIGMOD*, pages 745–747, 2006.

[33] A. Chapman and H. Jagadish. Issues in building practical provenance systems. *Data Engineering*, pages 38–44, December 2008.

[34] A. Chapman and H. Jagadish. *Provenance and the Price of Identity*, pages 162–170. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Provenance and Annotation of Data edition, 2008.

[35] A. Chapman, H. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, pages 993–1006, 2008.

[36] A. Chapman, M. Jayapandian, C. Yu, and H. V. Jagadish. MiMI: Michigan molecular interactions. In *ISMB*, 2005.

[37] A. Chapman, C. Yu, and H. V. Jagadish. Effective integration of protein data through better data modeling. *OMICS*, 7(1):101–102, July 2003.

[38] A. Chebotko, S. Chang, S. Lu, F. Fotouhi, and P. Yang. Scientific workflow provenance querying with security views. In *WAIM*, 2008.

[39] F. Chen et al. OrthoMCL-DB: querying a comprehensive multi-species collection of ortholog groups. *Nucleic Acids Research*, 34:D363–8, 2006.

[40] J. Cherry, C. Adler, C. Ball, S. Chervitz, S. Dwight, E. Hester, Y. Jia, G. Juvik, T. Roe, M. Schroeder, S. Weng, and D. Botstein. SGD: Saccharomyces genome database. *Nucleic Acids Res.*, 26(1):73–79, 1998.

[41] K. Cheung and J. Hunter. Provenance Explorer - customized provenance views using semantic inferencing. In *International Semantic Web Conference*, pages 215–227, 2006.

[42] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *ACM SIGMOD*, pages 942–944, 2005.

[43] J. Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[44] S. Cohen, S. C. Boulakia, and S. Davidson. Towards a model of scientific workflows and user views. In *DILS*, pages 264–279, 2006.

[45] S. Cohen-Boulakia, O. Biton, S. Cohen, and S. Davidson. Addressing the provenance challenge using ZOOM. *Concurrency and Computation: Practice and Experience*, 20:497–506, 2008.

[46] G. Cong, W. Fan, and F. Geerts. Annotation propogation revisited for key preserving views. In *CIKM*, pages 632–641, 2006.

[47] T. Corica, B. Brown, and B. Presley. *A Guide to Programming in C++*. Lawrenceville Press, Inc, 1998.

[48] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *ICDE*, pages 367–378, 2000.

[49] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, pages 41–58, 2001.

[50] Y. Cui, J. Widom, and J. Wiener. Tracing the lineage of view data in a data warehousing environment. In *ACM Transaction on Database Systems (TODS)*, 2000.

[51] S. M. S. da Cruz, P. M. Barros, P. M. Bisch, M. L. M. Campos, and M. Mattoso. Provenance services for distributed workflows. In *CCGRID*, pages 526–533, 2008.

[52] S. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludascher, T. McPhillips, S. Bowers, and J. Freire. Provenance in scientific workflow systems. *IEEE Data Engineering Bulletin*, 32(4):44–50, 2007.

[53] S. B. Davidson, J. Crabtree, B. P. Brunk, J. Schug, V. Tannen, G. C. Overton, and C. J. Stoeckert Jr. K2/Kleisli and GUS: Experiments in integrated access to genomic data sources. *IBM Systems Journal*, 40(2):512–531, 2001.

[54] 2000. http://www.infobiogen.fr/services/dbcat.

[55] C. M. Deane, Ł. Salwiński, I. Xenarios, and D. Eisenberg. Protein interactions: Two methods for assessment of the reliability of high throughput observations. *Molecular & Cellular Proteomics*, 1(5):349–356, May 2002.

[56] E. Deelman, S. Callaghan, E. Field, H. Francoeur, R. Graves, N. Gupta, V. Gupta, T. H. Jordan, C. Kesselman, P. Maechling, J. Mehringer, G. Mehta, D. Okaya, K. Vahi, and L. Zhao. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *e-Science*, 2006.

[57] G. Dellaire, R. Farrall, and W. A. Bickmore. The nuclear protein database (NPD): sub-nuclear localisation and functional annotation of the nuclear proteome. *Nucleic Acids Research*, 31(1):328–330, 2003.

[58] X. J. Duan, I. Xenarios, and D. Eisenberg. Describing biological protein interactions in terms of protein states and state transitions: The LiveDIP database. *Molecular & Cellular Proteomics*, 1(2):104–116, January 2002.

[59] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *ACM SIGPLAN*, pages 358–365, 1997.

[60] R. Finn et al. PFam: clans, web tools and services. *Nucleic Acids Research*, 34:D247–D251, 2006.

[61] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46, 2002.

[62] I. Foster, J. Vockler, M. Eilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *SSDBM*, pages 37–46, July 2002.

[63] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. The virtual data grid: a new model and architecture for data-intensive collaboration. In *CIDR*, 2003.

[64] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: Queries and provenance. In *PODS*, pages 271–280, June 2008.

[65] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, 2005.

[66] J. Frew and R. Bose. Earth system science workbench: A data management infrastructure for earth science products. In *SSDBM*, pages 180–189, 2001.

[67] J. Frew, D. Metzger, and P. Slaughter. Automatic capture and reconstruction of computational provenance. *Concurr. Comput. : Pract. Exper.*, 20(5):485–496, 2008.

[68] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and querying databases through colors and blocks. In *ICDE*, pages 82–92, 2006.

[69] Gene Ontology Consortium. Creating the gene ontology resource: design and implementation. *Genome Res*, 8:1425–1433, August 2001.

[70] B. Glavic and K. R. Dittrich. Data provenance: A categorization of existing approaches. In *BTW*, pages 227–241, 2007.

[71] T. J. Green, G. Karvounarakis, N. E. Taylor, O. Biton, Z. G. Ives, and V. Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *SIGMOD*, pages 1131–1133, June 2007.

[72] R. Grossi. On finding common subtrees. *Theor. Comput. Sci.*, 108(2):345–356, 1993.

[73] P. Groth, M. Luck, and L. Moreau. A protocol for recording provenance in service-oriented grids. In *OPODIS, Grenoble, France*, 2004.

[74] P. Groth, S. Miles, W. Fang, S. C. Wong, K.-P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *HPDC*, 2005.

[75] P. Groth, S. Miles, and L. Moreau. PReServ: Provenance recording for services. In *Proceedings of the UK OST e-Science second All Hands Meeting 2005 (AHM'05)*, 2005.

[76] P. Groth, S. Miles, and L. Moreau. A Model of Process Documentation to Determine Provenance in Mash-ups. *Transactions on Internet Technology (TOIT)*, 2008.

[77] J. Han et al. Evidence for dynamically organized modularity in the yeast protein-protein interaction network. *Nature*, 430:88–93, 2004.

[78] H. Hermjakob et al. IntAct - an open source molecular interaction database. *Nucleic Acids Research*, 32:D452–D455, 2004.

[79] P. Heus and R. Gomez. QIS-XML: A metadata specification for quantum information science. Technical report, arXiv.org, 2007.

[80] J. Huang, T. Chen, A. Doan, and J. Naughton. On provenance of non-answers to queries over extracted data. In *VLDB*, 2008.

[81] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native XML database. *The VLDB Journal*, 11(4):274–291, 2002.

[82] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. pages 13–24, 2007.

[83] M. Jayapandian, A. Chapman, V. Tarcea, C. Yu, A. Elkiss, A. Ianni, B. Liu, A. Nandi, C. Santos, P. Andrews, B. Athey, D. States, and H. Jagadish. Michigan Molecular Interactions (MiMI): Putting the jigsaw puzzle together. *Nucleic Acids Research*, pages D566–D571, Jan 2007.

[84] P. Kersey et al. The international protein index: An integrated database for proteomics experiments. *Proteomics*, 4(7):1985–1988, 2004.

[85] Y. J. Kim, A. Boyd, B. D. Athey, and J. M. Patel. miBLAST: Scalable evaluation of a batch of nucleotide sequence queries with blast. *Nucleic Acids Research*, 33(13):4335–4344, 2005.

[86] K.Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. A. A. Fernandes, and G. Mehta. Adaptive workflow processing and execution in pegasus. In *3rd International Workshop on Workflow Management and Applications in Grid Environments (WaGe08)*, pages 99–106, 2008.

[87] A. J. Ko and B. A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *SIGCHI*, pages 151–158, 2004.

[88] T. Lee, S. Bressan, and S. E. Madnick. Source attribution for querying against semi-structured documents. In *ACM WIDM*, pages 33–39, 1998.

[89] H.-J. Lenz and A. Shoshani. Summarizability in OLAP and statistical data bases. In *SSDM*, pages 132–143, 1997.

[90] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *SIGMOD*, 2000.

[91] B. Ludäscher, K. Lin, S. Bowers, E. Jaeger-Frank, B. Brodaric, and C. Baru. Managing scientific data: From data integration to scientific workflows. *GSA Today*, Special Issue on Geoinformatics, 2005.

[92] A. Marian, S. Abiteboul, G. Cobena, and L. Mignet. Change-centric management of versions in an XML warehouse. In *VLDB*, pages 581–590, 2001.

[93] T. McPhillips, S. Bowers, and B. Ludscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *DILS*, pages 248–263, 2006.

[94] S. Miles, E. Deelman, P. Groth, K. Vahi, G. Mehta, and L. Moreau. Connecting scientific data to scientific experiments with provenance. In *E-SCIENCE*, pages 179–186, 2007.

[95] S. Miles, S. C. Wong, W. Fang, P. Groth, K.-P. Zauner, and L. Moreau. Provenance-based validation of e-science experiments. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(1):28–38, 2007.

[96] J. Min, M. Park, and C. Chung. XPRESS: a queriable compression for XML data. In *SIGMOD*, 2003.

[97] P. Missier, S. Embury, M. Greenwood, A. Preece, and B. Jin. Quality views: Capturing and exploiting the user perspective on data quality. In *VLDB*, pages 977–988, 2006.

[98] P. Missier, S. M. Embury, M. Greenwood, A. Preece, and B. Jin. Managing information quality in e-science: the qurator workbench. In *SIGMOD*, pages 1150–1152, 2007.

[99] P. Missier, A. Preece, S. Embury, B. Jin, M. Greenwood, D. Stead, and A. Brown. Managing information quality in e-science: A case study in proteomics. In *Proc 1st Workshop on Quality of Information Systems (QoIS 2005)*, pages 423–432, 2005.

[100] L. Moreau, J. Freire, J. Futrelle, R. McGrath, J. Myers, and P. Paulson. The open provenance model. Technical report, University of Southampton, 2007.

[101] L. Moreau, B. Ludäscher, et al. The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*, 2007. http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge .

[102] L. Moreau, B. Ludscher, I. Altintas, R. S. Barga, S. Bowers, S. Callahan, G. C. JR., B. Clifford, S. Cohen, S. Cohen-Boulakia, S. Davidson, E. Deelman, L. Digiampietri, I. Foster, J. Freire, J. Frew, J. Futrelle, T. Gibson, Y. Gil, C. Goble, J. Golbeck, P. Groth, D. A. Holland, S. Jiang, J. Kim, D. Koop, A. Krenek, T. McPhillips, G. Mehta, S. Miles, D. Metzger, S. Munroe, J. Myers, B. Plale, N. Podhorszki, V. Ratnakar, E. Santos, C. Scheidegger, K. Schuchardt, M. Seltzer, Y. L. Simmhan, C. Silva, P. Slaughter, E. Stephan, R. Stevens, D. Turi, H. Vo, M. Wilde, J. Zhao, and Y. Zhao. Special issue: The first provenance challenge. *Concurrency and Computation: Practice and Experience*, 20:409–418, 2008.

[103] N. Mulder et al. Interpro, progress and status in 2005. *Nucleic Acids Research*, 33:D201–D205, 2005.

[104] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *USENIX Annual Technical Conference*, pages 43–56, 2006.

[105] S. Munroe, S. Miles, L. Moreau, and J. Vzquez-Salceda. PrIMe: a software engineering methodology for developing provenance-aware applications. In *SEM*, pages 39–46, 2006.

[106] M. Mutsuzaki, M. Theobald, et al. Trio-One: Layering uncertainty and lineage on a conventional DBMS. In *CIDR*, pages 269–274, 2007.

[107] B. A. Myers, D. A. Weitzman, A. J. Ko, and D. H. Chau. Answering why and why not questions in user interfaces. In *SIGCHI*, pages 397–406, 2006.

[108] J. D. Myers, C. Pancerella, et al. Multi-scale science: Supporting emerging practice with semantically derived provenance. In *Proceedings of the Semantic Web Technologies for Searching and Retrieving Scientific Data Workshop*, 2003.

[109] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: lessons in creating a workflow environment for the life sciences: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1067–1100, 2006.

[110] Open provenance model. http://twiki.ipaw.info/bin/view/Challenge/OPM, 2008.

[111] C. Pancerella, J. Hewson, W. Koegler, et al. Metadata in the collaboratory for multi-scale chemical science. In *Dublin Core Conference*, 2003.

[112] H. Pang, A. Jain, K. Ramamritham, and K. Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2005.

[113] S. Paparizos, S. Al-Khalifa, A. Chapman, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native system for querying xml. In *SIGMOD*, pages 672–684, 2003.

[114] S. Peri et al. Development of human protein reference database as an initial platform for approaching systems biology in humans. *Genome Research*, 13:2363–2371, 2003.

[115] M. Reich, T. Liefeld, et al. Genepattern 2.0. *Nature Genetics*, 38:500–501, 2006.

[116] D. D. Roure and C. Goble. myExperiment - a web 2.0 virtual research environment. In *International Workshop on Virtual Research Environments and Collaborative Work Environments*, 2007.

[117] L. Salwinski et al. The database of interacting proteins: 2004 update. *Nucleic Acids Research*, 32:D449–D451, 2004.

[118] C. Santos, D. Eggle, and D. J. States. Wnt pathway curation using automated natural language processing: combining statistical methods with partial and full parse for knowledge extraction. *Bioinformatics*, 21(8):1653–1658, 2005.

[119] O. Sasson et al. ProtoNet: hierarchical classification of the protein space. *Nucleic Acids Research*, 31(1):348–352, 2003.

[120] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. Silva. Querying and re-using workflows with vistrails. In *SIGMOD*, 2008.

[121] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.

[122] M. Seltzer, J. Ledlie, C. Ng, D. Holland, K. Muniswamy-Reddy, and U. Braun. Provenance aware sensor data storage. In *NetDB*, 2005.

[123] A. Sheth. Metadata storage (ASEMR) deployed at the Athens Heart Center. personal communication, Oct 2006.

[124] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[125] Y. Simmhan, B. Plale, and D. Gannon. A framework for collecting provenance in data-centric scientific workflows. In *ICWS*, 2006.

[126] Y. Simmhan, B. Plale, D. Gannon, and S. Marru. *Performance evaluation of the Karma provenance framework for scientific workflows*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Provenance and Annotation of Data edition, 2006.

[127] E. Sprinzak, S. Sattath, and H. Margalit. How reliable are experimental proteinprotein interaction data? *Journal of Molecular Biology*, 327:919–923, 2003.

[128] C. Stark, B. Breitkreutz, et al. BioGRID: a general repository for interaction datasets. *Nucleic Acids Research*, pages D535–D539, 2006.

[129] T. Stef-Praun, B. Clifford, I. Foster, U. Hasson, M. Hategan, S. Small, M. Wilde, and Y. Zhao. Accelerating medical research using the swift workflow system. *Health Grid*, 2007.

[130] U. Stelzl et al. A human protein-protein interaction network: A resource for annotating the proteome. *Cell*, 122:957–968, 2005.

[131] W. Tan. Containment of relational queries with annotation propagation. In *DBPL*, pages 37–53, 2003.

[132] W. C. Tan. Research problems in data provenance. *IEEE Data Eng. Bull.*, 27(4):45–52, 2004.

[133] N. Taylor. Oracle content services application developer's guide 10g release 1 (10.1.2.2). Technical report, Oracle, 2006.

[134] P. M. Tolani and J. R. Haritsa. XGRIND: A query-friendly XML compressor. In *ICDE*, pages 225–234, 2002.

[135] A. H. Y. Tong, B. Dress, G. Nardelli, G. D. Bader, B. Brannetti, L. Castagnoli, M. Evangelista, S. Ferracuti, B. Nelson, S. Paoluzi, M. Quondam, A. Zucconi, C. W. V. Hogue, S. Fields, C. Boone, and G. Cesareni. A combined experimental and computational strategy to define protein interaction networks for peptide recognition modules. *Science*, 295:321–324, January 2002.

[136] UniProt. http://www.ebi.ac.uk/uniprot/.

[137] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[138] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and querying data transformations. In *ICDE*, pages 81–92, 2005.

[139] Y. R. Wang and S. E. Madnick. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB*, pages 519–538, 1990.

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

[141] N. Wiwatwattana, H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. X3: A cube operator for XML OLAP. In *ICDE*, pages 916–925, 2007.

[142] N. Wiwatwattana and A. Kumar. Organelle DB: a cross-species database of protein localization and function. *Nucleic Acids Research*, 33:D598–604, 2005.

[143] S. C. Wong, S. Miles, W. Fang, P. Groth, and L. Moreau. Provenance-based validation of e-science experiments. In *Proceedings of 4th International Semantic Web Conference (ISWC), Lecture Notes in Computer Science*, pages 801–815, 2005.

[144] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 97–102, 1997.

[145] I. Wootten, S. Rajbhandari, O. F. Rana, and J. S. Pahwa. Actor provenance capture with ganglia. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 99–106, 2006.

[146] I. Xenarios, Ł. Salwínski, X. J. Duan, P. Higney, S.-M. Kin, and D. Eisenberg. DIP, the database of interacting proteins: a research tool for studying cellular networks of protein interactions. *Nucleic Acids Research*, 30(1):303–305, 2002.

[147] J. Zhang, A. Chapman, and K. LaFevre. Fine-grained tamper-evident data pedigre. In *ICDE submitted*, 2009.

[148] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar. Tracing lineage beyond relational operators. In *VLDB*, pages 1116–1127, 2007.

[149] J. Zhao, C. Goble, and R. Stevens. *An Identity Crisis in the Life Sciences*, pages 254–269. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Provenance and Annotation of Data edition, 2006.

[150] J. Zhao, C. A. Goble, R. Stevens, and S. Bechhofer. Semantically linking and browsing provenance logs for e-science. In *ICSNW*, pages 158–176, 2004.

[151] Y. Zhao, J. E. Dobson, I. T. Foster, L. Moreau, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *SIGMOD Record*, 34(3):37–43, 2005.