Tracing Lineage in Multi-version Scientific Databases

Mingwu Zhang Daisuke Kihara and Sunil Prabhakar
Department of Computer Science, Purdue University
West Lafayette, IN 47907-1398, USA
{mzhang2, dkihara, sunil}@cs.purdue.edu

Abstract

The critical need for better tracing of lineage in scientific databases is well known [6]. It is clear that performance is not an issue for most domain scientists – rather the functionality is more important. In this paper, we highlight the importance of maintaining multiple versions of data and tracing fine-grained lineage in support of these needs. We study alternatives for managing versions, and propose a model for the example application of protein annotations. We present query rewriting algorithms for SPJ and ASPJ queries that piggy-back lineage computation with query evaluation. Our models are implemented using PostgreSQL and tested using a large, real dataset from Uniprot. We establish the validity of the approach in enabling relevant queries and study the space and time overheads. While these overheads can be high in some cases, the real gain for scientists is the novel functionality that can allow them to ascertain reliability of derived data, and foster data-driven research. To the best of our knowledge, this is the first work that can handle these types of queries for lineage tracing.

1 Introduction

With the development of high-throughput experimental technologies, scientists today are producing large volumes of data and rely more and more on Database Management Systems (DBMSs) to store and manage scientific data. However, processing of scientific data imposes challenges that are not well-addressed in modern DBMSs, which are generally designed to handle applications in the business world.

A salient example to showcase the lack of database support for scientific data is data provenance. Data provenance and lineage are almost synonymous in scientific databases. Lineage means derivation and provenance means origin or source. As their names suggest, lineage is usually used to trace the derivation process and the origin of the data. It also provides insights into the reliability and in general, quality of the data. By examining provenance, scientists may judge the data quality based on their own experiences. Provenance is also required in data dissemination and reproduction.

As a motivating example, we consider the well-known problem of error propagation in protein annotations in Uniprot [9]. Uniprot is a core functional annotation database which maintains information about millions of proteins. There are two main sources of functional annotations: *experimental* and *computational*. The most reliable annotations are determined experimentally. Unfortunately, only a small fraction of proteins have experimentally verified annotations due to the difficulty of this process. More often, proteins are annotated based upon their similarity to other annotated proteins in the database. There are a number of different methods used to estimate protein function. For example, based upon similarity of the sequence (e.g., using BLAST [1]), and scores reported by Hidden Markov Models. When a new protein is identified, a biologist may first compare it against the entire database of existing sequences using a tool such as BLAST [1]. The functional annotations from the resulting sequences with similarity beyond a user-defined threshold (i.e., high homology) are used to generate an annotation for the new sequence. The new sequence and its annotation are now added to the database and used in future BLAST searches.

A major problem with this approach is that computational annotation can be wrong. Furthermore, once a protein is annotated computationally, its annotation is treated the same way as an experimentally verified protein. Thus, future annotations may propagate and compound annotation errors through computational means. This leads to difficulties in assigning confidence to protein annotations. This difficulty in managing protein annotations is well recognized: According to a recent editorial in a leading Bioinformatics journal [6]: "Each new sequence deposited in the public database has been annotated with respect to those same databases. Function annotations are propagated repeatedly from one sequence to the next, with no record made of the source of a given annotation, leading to a potential transitive catastrophe of erroneous annotations."

In addition to the propagation of potential errors in annotations, there is a challenge posed by updates to data. Naturally, an important class of updates is simply the addition of new data. There are also instances where an annotation is found to be incorrect based upon experiments. This requires an update to the specific annotation, and also to all other derived annotations. Without the availability of dependency information, this is not possible to do. Furthermore, even if such information were available, the updates to derived data are not straight-forward to compute. Computing updates requires expert intervention. In a traditional database, derived data such as materialized views are always maintained in a timely fashion. So the lineage traced is always correct. But in scientific databases, it may not always be feasible to propagate the changes to the derived data immediately. This can lead to incorrect (stale) or phantom lineage.

The above problems can be addressed by not deleting old data, but maintaining multiple versions of data. In scientific databases, historical data is as valuable as the current data. A scientific discovery has to be repeatable before it is widely accepted. Other researchers may depend on a specific version of the data to reproduce the results. Without this data, the discovery would be unverifiable. The decreasing cost of storage allows scientific databases to store the historical data economically. Shared databases such as Uniprot are released periodically since it would be too cumbersome to handle very frequent releases. However, this delays the availability of new data. Also, older versions are simply unavailable thus forcing researchers to copy and maintain multiple versions in order to retain access to current and historical data. The availability of a multi-version system would alleviate these concerns and also enable users to access any "version" – not just the current release.

What is needed in this area is the development of a system that i) retains multiple versions of data (e.g., multiple annotations of proteins over time); ii) allows users to discover dependencies between data; iii) enables the querying of these dependencies; and iv) enables *data-driven* research through statistical analysis of these dependencies. One of the goals of bioinformatics is to help transform biology from a *hypothesis-driven* to a *data-driven* mode. In other words, biologists would like to be able to analyze data to identify promising experiments that are likely to be fruitful, instead of relying on hypotheses.

Although there has been some recent work on database support for lineage in scientific databases, the current approaches are unable to provide the types of support discussed above. The broader goal of our research is to develop a database management system that provides the necessary support for managing scientific data, as exemplified by the protein annotation application. In this paper, we develop a solution for data lineage that satisfies the above requirements. In particular, we propose materialized, fine-grained lineage for multi-version data. A prototype system has been developed on top of PostgreSQL and populated with multiple releases of Uniprot. Using this system, we study the overhead of supporting this fine-grained lineage and also demonstrate its ability to support a wide range of queries of interest to scientists. To the best of our knowledge, lineage tracing in a multi-version database has not been addressed.

This paper proposes a novel system that supports fine-grained lineage tracing for derived data. We investigate alternative designs for version management, and propose query rewriting algorithms that enable efficient computation of lineage during query processing. New types of lineage are defined and used in our model. A prototype system is developed and used to validate the claims with real protein annotation data. The algorithms are shown to improve the query execution times when tracing lineage. We also highlight some of the queries that can be executed with our system. Most importantly, this paper addresses the need for improved tracing functionality in scientific databases.

2 Related Work

Provenance or lineage has been extensively studied in the context of scientific computation such as datasets on the grid. One form of the provenance is workflow or coarse-grained provenance. In scientific computation, coarse-grained (i.e., table or file level) lineage is sufficient because typically all elements in the same file or table have undergone the same computational process. Also the lineage is used to trace the source of abnormality in the data or for the data dissemination (i.e., a description of the derivation process is disseminated along with the base data). [3] surveys the use of workflows in scientific computation.

In scientific databases, for example biological databases, keeping the coarse-grained lineage is insufficient since not all data values are processed similarly. There is a great need for DBMS support for fine-grained lineage tracing. Although the need is urgent, it remains an unsolved problem. Recently, there has been increasing interest in this area. Cui *et al.* [5] propose fine-grained tracing in the context of data warehousing where all data is produced using relational database queries. The notion of reverse queries that are automatically generated is presented in order to produce all tuples that participated in the computation of a given query. Woodruff and Stonebaker [10] support fine-grained lineage using inverse or weak inverse functions. That is, the dependence of a given result on base data is captured using a mathematical function. They adopt a lazy approach to compute fine-grained lineage upon request from the user. It is not clear if such functions can be identified for a given application. The identification task is highly non-trivial and makes the approach impractical. Bhagwat *et al.*[2] proposed three schemes to propagate annotations attached to attributes in relational data. The *where* lineage is a unique address recorded in the annotation, along with other non-lineage information. As it only records where the data is copied from, *where* lineage is not sufficient for scientific databases where data go through complex processing. The intuition behind *where* lineage is rooted in the classical view maintenance problem. Another limitation of these approaches is that the lineage information is stored as unstructured text which makes it very difficult to analyze.

Overall we see that while tracing of fine-grained lineage and storage of multi-version data is critical to supporting meaningful tracing of scientific databases, current solutions fall short of these requirements. To the best of our knowledge, ours is the first work to propose such a system and the only one that can support the types of queries discussed in Section 5 which are of direct relevance to scientists.

3 Modeling Version

The new version of base data results from the insertion, update and deletion. Reasons for such changes include: invalidation of previous belief by a new discovery; identification of an error; verification of computational results by experiments. These changes have to be propagated to the derived data resulting in new versions of the derived data. A new version may also be derived using a subset of the base data or an alternative data source. The method and the parameters used may be changed to produce a new version of the derived data.

In previous work, each release of the database is considered to be a database version. This is stored separately from other releases. In [4] redundancies between multiple versions are removed to achieve savings of storage. Such simple versioning is insufficient for supporting effective lineage tracing. Our model tracks versions at the database, table, and tuple levels. Database version refers to a state of the database (which could be the periodic release of the database as in earlier work, or an intermediate snapshot at a given time between releases); table version tracks multiple versions of the same table; and tuple version tracks multiple version of the same tuple within a table (e.g., updated versions of the same protein). This model for versions is much richer and more flexible than earlier notions of versions.

We consider two alternative models for representing versions in scientific databases. In the rest of this section, we discuss the advantages and disadvantages of these models.

3.1 Version number

The obvious choice is to represent version as an extra attribute. This easily captures tuple versions. However, in order to represent table and database versions, it is necessary to define a separate table for each table version and database version. This separate table(s) list the tuple version numbers that make up the corresponding table or database version.

Figure 1 shows a subset of the relations from the Uniprot schema. There are over 65 tables in the Uniprot schema. The **Dbentry** table contains an entry for each protein along with its name and unique identifier for Uniprot (the AC number). The **Sequence** table stores the sequence information for a given protein. It contains a foreign key into the **Dbentry** table and a string describing the sequence of amino acids that make up the protein. The **Description** table stores the functional annotation for the protein. It contains a foreign key into the **Dbentry** table, and a textual description of the function of the protein. We augment the **Sequence** and **Description** tables with a version attribute. Note that only the relevant fields from each table are shown here. The AC number for a protein may change from one version to another. We do not handle this situation as a new version of **Dbentry** and instead treat the AC number change as lineage information. Interested readers can find details in our technical report [7]. The version numbers are incremented for each new version of the tuple inserted into the table.

The deletion of a tuple can be handled either as a special value for the version or using an extra column in the schema. We choose use a special value to mark deletion. This model for version is straightforward, but awkward when querying the table version. For example, if the user wants to join the current version of two tables, she can issue the following query:

```
select * from Sequence S1, Description D1
where S1.dbentry_id=D1.dbentry_id and
S1.version !=special_value and
S1.version in (select max(S2.version)
from Sequence S2
where S2.dbentry_id=S1.dbentry_id)
```

```
and D1.version != special_value
and D1.version in (select max(D2.version)
from Description D2
where D1.dbentry_id=D2.dbentry_id)
```

This query contains a correlated sub-query. To avoid the correlated sub-query, we may introduce *curr_version* table to keep track of the current version number for each tuple. A similar table is needed to represent each table and database version. It should be noted that it is extremely difficult to represent the database version. To represent a database version, one needs to know the version number of each tuple in that version of the database. This is impractical for a database with a large number of versions.

3.2 Timestamp

In the second model,we use two timestamps to represent the tuple version explicitly and table version and database version can be easily inferred from the tuple version. For each tuple, two attributes of timestamp type, *start_time* and *end_time*, represent the valid time period for the data item [8]. These timestamps are similar to valid time in temporal databases but only the current data can be modified. Representing the version in valid time is intuitive because users usually remember when the data is derived rather than the individual version numbers of the tuples used to derive the data. The schema and part of the table are shown in Figure 1. The primary key of the sequence table is *sequence_id* while (*dbentry_id*, *start_time*, *end_time*) is a unique key. The primary key of the **Description** table is *description_id* while (*dbentry_id*, *start_time*, *end_time*) is a unique key. "Now" is defined by setting *start_time* to the current time and *end_time* to "infinite". This "infinite" timestamp could be a special value which is very large, for example "12-31-3000." The join of the current version of the two tables can be expressed as follows.

```
select * from sequence S, description D
where S.dbentry_id = D.dbentry_id and
S.start_time<=current_time and
S.end_time>current_time and
D.start_time<=current_time and
D.end_time>current_time
```

If a user wants to query for a specific version of the data, she only needs to know the time when the specific version is valid. The representation of deletion is natural in this model. If the data is deleted, the *end_time* of the current data which has a value of "infinite" will be changed to the time of deletion.

Both the table and database versions can be simply represented as timestamps. The table (database) version for a given timestamp consists of the tuples in the table (database) that are valid at that time. A user only needs to remember the time for the specific version. A simple selection query with this time value is sufficient to access the corresponding version of the table. This model also has the advantage of (efficiently) representing a large number of versions.

In Section 5, we compare the performance of these two alternatives. Our experiments suggest that timestamps give better performance than version numbers. Given this performance advantage and the difficulties of representing multiple versions (especially for table and database) using version numbers, we limit the following discussion to timestamp based versions. This model is also more intuitive and user friendly.

4 Lineage

In this section we discuss the types and granularity of lineage necessary for effective lineage tracing. We have defined new types of lineage and the details can be found in our technical report[7]. We also discuss implementation issues with respect to the efficient computation of fine-grained lineage.

Granularity An important design decision is the granularity of the lineage information. Some applications only need coarse-grained lineage (e.g., at the table level only). For example, if all tuples are derived using the same procedures. However, many applications need much finer granularity lineage (e.g., tuple-level). Consider for example, in Uniprot different annotations are derived using different methods (experimental or computational methods). Therefore it is difficult to assign table-level lineage that is meaningful for all tuples.

In earlier work, fine-grained lineage is computed upon user request and is not materialized on the grounds that it is expensive to store this extra information and also that computing fine-grained lineage is infrequent [5, 10]. The only other work that proposes to materialize fine-grained lineage is [2]. However, the type of lineage stored is insufficient to support important queries.

As the cost of storage decreases, it is both practical and economical to store fine-grained lineage. We believe that fine-grained lineage should be stored because it contains rich and interesting information. Materializing this information allows users to compute useful *ad hoc* querying. For example, in Uniprot, fine-grained lineage can be queried to reveal the dependency between protein annotations. A derived protein annotation with a shorter dependency chain may reflect higher reliability. Fine-grained dependency can help scientists identify critical annotations – those on which a large number of other annotations depend. Experimental resources can then be targeted at verifying these critical annotations. In our framework, fine-grained lineage

Description

description_id	dbentry_id	description	start_time	end_time
131063	131063	VirE locus ...	2004-07-05	2004-10-25
221909	131063	VirE locus ...	2004-10-25	2005-02-01
301527	131063	VirE locus ...	2005-02-01	infinity
148181	148181	Hypothetical protein ...	2004-07-05	2004-10-25
230469	148181	Hypothetical protein ...	2004-10-25	2005-02-01
313898	148181	Putative amino-acid...	2005-02-01	infinity

Dbentry

dbentry_id	AC	name
131063	P08061	VIE3_AGRT5
148181	P34261	YKAA_CAEEL

Sequence

sequence_id	dbentry_id	sequence	start_time	end_time
131063	131063	MHGDD...	1988-08-01	infinity
148181	148181	MSACT...	2003-02-08	2004-10-25
165976	148181	MDELE...	2004-10-25	2005-02-01
172726	148181	MPSST...	2005-02-01	infinity

Figure 1. Representing version using timestamps

may or may not be materialized depending upon user preference.

Tracing fine-grained internal lineage In a relational database, if data is derived using an SQL statement, the lineage information is internal lineage. If the data is not derived by an SQL statement, it is external lineage. If fine-grained internal lineage is computed on the fly, no fine-grained lineage will be stored. A reverse query similar to the one defined in Cui *et al.* [5] can easily be issued upon user's request to trace the fine-grained lineage on the fly. On the other hand, if fine-grained lineage is materialized, we can avoid executing reverse queries. The lineage can be piggy-backed with the query execution itself.

In order to store fine-grained lineage, we could always run the query to get the result , then generate the reverse query for each tuple in the result and store the fine-grained lineage. A more efficient alternative is to compute the lineage information as the query is being executed. There are two cases to consider: Selection-Projection-Join(SPJ) queries and Aggregate-Selection-Projection-Join(ASPJ) queries.

SPJ queries. We assume the database, table and tuples can be uniquely identified. For an ordinary SPJ query, we need to retain the *tuple_id* from the base tables when the queries are being executed. Algorithm 1 transforms the original SPJ query to trace the lineage at execution time.

Algorithm 1 Tracing internal lineage for SPJ query

```
 1: for each table that appears in the from list do
 2:    add table.tuple_id to the select list.
 3: end for
 4: create a view using the modified query
 5: for each row in the view do
 6:    for each table.tuple_id added do
 7:       insert one tuple in the where_lineage table.
 8:    end for
 9: end for
10: drop added table.tuple_id columns
```

ASPJ query. For a typical ASPJ query, the aggregate is based on the **group by** clause. Algorithm 2 transforms the original ASPJ query to trace the lineage at query execution time. For example, the following

Algorithm 2 Tracing internal lineage for ASPJ query

```
 1: remove the group by and aggregate clauses
 2: for each attribute that appears in the group by clause
    do
 3:    add the attribute to the select list
 4: end for
 5: for each table that appears in the from list do
 6:    add the table.tuple_id to the select list
 7: end for
 8: create a temporary view using the modified query
 9: create a view with aggregate and group by using temporary view
10: join the temporary view and view with the group by attributes
11: for each row in the result of the join do
12:    for each table.tuple_id added do
13:       insert one tuple in the where_lineage table.
14:    end for
15: end for
16: drop the temporary view
```

query can be used to compute how many annotations for each protein contains the keyword 'hypothetical' over all versions.

```
create table view
select dbentry_id, count(*) from  description d
where d.text like 'hypothetical'
group by dbentry_id
```

Algorithm 2 outlines the transformation steps that augment ASPJ queries with fine-grained lineage tracing. First, the **group by** and aggregate clauses are removed. The attributes in the **group by** and *tuple_id* of the base table are appended to the **select** list. The aggregate is then calculated.

```
create table temp_view as
select dbentry_id, description_id
from description d
where d.text like 'hypothetical';
create table view as
select v.dbentry, count (v.dbentry)
from temp_view v group by v.dbentry_id
```

In order to find the lineage information, *temp_view* and *view* are joined on the attributes in the **group by** clause.

```
create table depend_view as
select r.tuple_id, v.description_id
from view r, temp_view v
where r.dbentry_id=v.dbentry_id
```

Next, for each tuple in *depend_view*, for each column other than *depend_view.tuple_id*, insert a tuple in the *where_lineage* table. Finally, *temp_view* and *depend_view* are dropped.

5 Experimental Results

In this section we describe the implementation of our lineage tracing ideas in PostgreSQL, and present the performance using a real data set.

We have created a database for storing both the protein data (with annotations) and the necessary lineage data for multiple versions of the publicly available Uniprot[9] protein database. In particular, we imported versions 44, 45 and 46 of Swiss-Prot, and versions 29, 30 and 31 of the TREMBL database. Version 46 of Swiss-Prot contains 168,297 manually curated protein annotations and version 31 of TREMBL contains 2,144,471 computationally generated protein annotations. The schema of Swiss-Prot is obtained from Uniprot and modified to include version information. We use version 7.4.5 of PostgreSQL.

5.1 Functionality

In this section, we demonstrate the advantages of our approach of storing fine-grained lineage and versions for the data. These two aspects enable a large number of relevant and important queries that scientists would like to be able to answer [6]. Each of the queries presented below cannot be answered without fine-grained lineage and multi-version data. To the best of our knowledge, none of the existing work on tracing lineage is able to answer these queries.

Query 1. Show the dependency graph of a tuple. In the multi-version Swiss-Prot, queries of this type are the most common. Biologists want to see the dependency graph involving a particular protein to assess the quality of the data. The global dependency graph contains a node for each tuple and directed edges between related tuples. An edge goes from a derived data tuple to the data on which it depends. The dependency graph of a given tuple refers to the connected component containing that tuple. This query can not be expressed in SQL because SQL lacks the capability to express recursive queries. A C++ program based on the BFS algorithm is implemented to find the dependency graph of the tuples. The output of the program

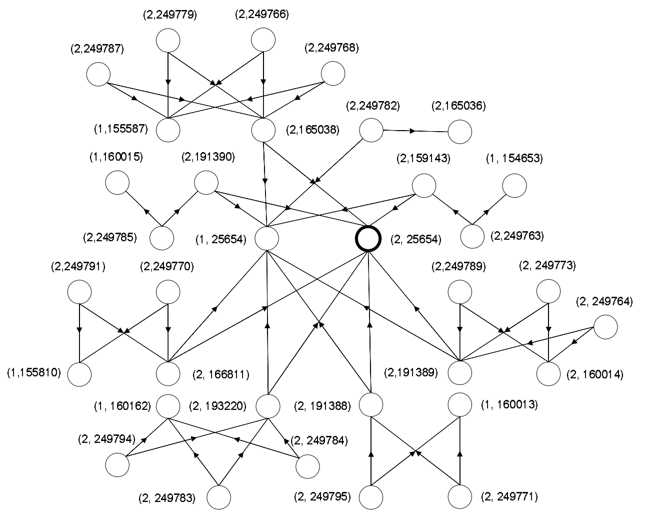


Figure 2. The dependency graph of (2, 25654)

is a graph in GML format and is read by GVF to display the dependency between the tuples[1]. Figure 2 shows the dependency graph for protein (2, 25654).[2] This graph shows all the interrelated pieces of data connected with the given tuple (shown in bold). The directed edges points from derived data to base data.

Query 2. For a given piece of derived data, determine if any of its base data has been changed. Because the update to the base data may not be propagated to the derived data, it is important to see if the data on which the derived data depends has been updated since the derivation. This query checks if any item in the dependency graph is not the current version.

Query 3. Show the data that more than n derived data depend on. In the *hypothesis-driven* research paradigm, scientists postulate a hypothesis and conduct experiments to verify the hypothesis. Scientists rely on their expertise to propose the hypothesis. In the *data-driven* research paradigm, the data and the result of experiment will suggest the next possible experiment or critical experiments to be conducted. The hypothesis in *data-driven* paradigm is guided by the data and may be more thorough and systematic. For example, in our multi-version Swiss-Prot database, if many protein annotations depend on a particular protein annotation and it is derived from a computational method, then it is important to verify its function by wet bench experiments.

Query 4. Show the statistics of the dependency. The statistics of the global dependency graph reveal

[1]Graph Modeling Language (GML) and Graph Visualization Framework (GVF).

[2]The two numbers refer to the table_id and tuple_id vales from the database. These are used in order to uniquely identify a tuple. Details can be found in [7].

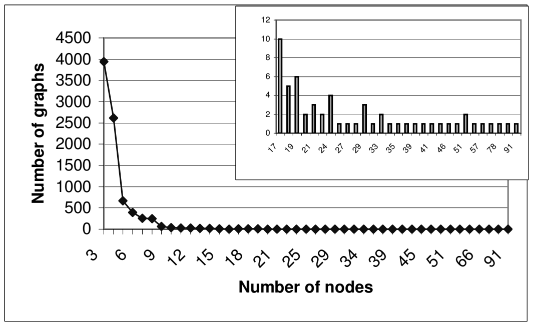


Figure 3. Dependency graph size

information about the entire database. For example, if many derived data have long derivation chains, the probability that errors have propagated to derived data is high. If fine-grained lineage is not stored, it is not possible to calculate this statistic information. Figure 3 shows the distribution of the size of tuple dependency graphs. Each point in the graph shows the total number of tuple dependency graphs with a given number of nodes. The inset graph shows a detailed subsection of the larger graph. For example, 86% of dependency graphs consist of less than 5 nodes. That is, the number of interdependent annotations is usually small. However, there are some cases where more than 80 annotations are related. We also calculate the longest shortest path in these graphs. There are total 8389 dependency graphs in the database. 8073 dependency graphs have longest shortest path of 1 and 316 graphs have longest shortest path of 2. These statistics reveal the reliability of the whole database. Because 99% of the dependency graphs in the database has less than or equal to 15 nodes and 96% of the dependency graphs have longest shortest path of 1 and all of them have a longest shortest path no more than 2, we have confidence that the reliability of the database is high. If there are many graphs have large number of nodes and long shortest paths, then the reliability of the database is in question. Wet bench experiments need to be done to verify the prediction results.

5.2 Space and Performance

We conduct experiments using queries that have 0, 1, or 2 joins, denoted as $Q1$, $Q2$, $Q3$. The cardinality of the result range from 10,000 to 1,000,000 rows. The experiments are conducted on a Sun-Blade-1000 machine with dual 1.2 GHz UltraSPARC-III+ CPUs and 2G memory running SunOS 5.8.

Version Models In Figure 4, Q is run in timestamp model database and Q' is run in a version number model database with the correlated sub-query. Q'' is run in a version number model database with current version table implemented. As can be seen from the graph, the timestamp version outperforms both ver-

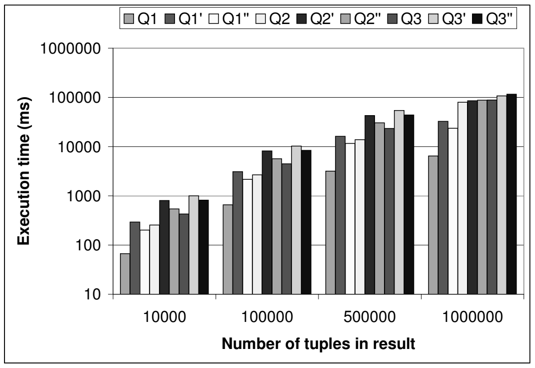
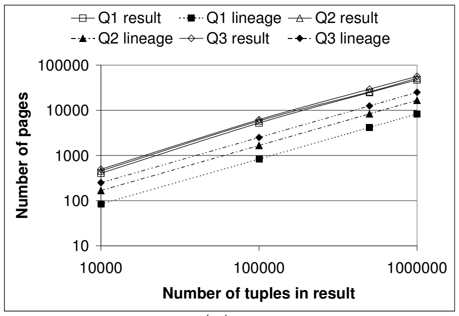


Figure 4. Timestamps vs Version number

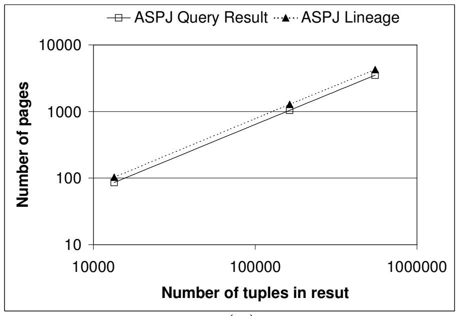
sion number alternatives. Therefore we do not consider version numbers any further.

Space In order to investigate the storage overhead of storing fine-grained lineage, we conduct experiments to measure the space cost in the number of pages (each page is 8K bytes). Figure 5 (a) shows the storage cost for query results and lineage for a range of query result sizes. It can be seen that the size of the lineage is roughly 20% (Q1), 30% (Q2) and 40% (Q3) of the result size. There is a direct relationship between the number of joined relation and the overhead ratio since for each result tuple, we need to save the joining tuples as lineage information. The number of rows of *where_lineage* depends on the number of joins, so the number of rows of *where_lineage* is linear in the number of rows returned by the query. In Figure 5 (b), we compare the space cost of ASPJ queries. The *where_lineage* used 120% of storage of the result. This result is expected because every tuple contribute to the aggregate should be recorded in the *where_lineage*, the cardinality of the *where_lineage* should be the result of query that does not have the aggregate, which could be much larger than the cardinality with the aggregate.

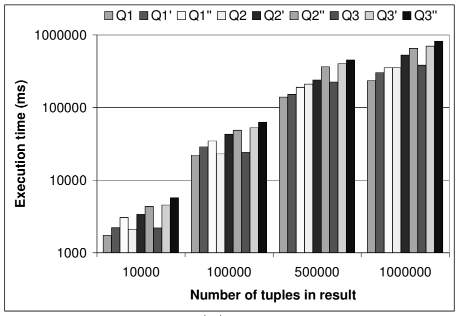
Time Calculating the lineage information at execution will incur a performance cost, we measure the execution time to compute the fine-grained lineage and compare it with the execution time without the lineage. We also implemented the reverse query algorithm. In Figure 5 (c), Q is the execution time without lineage tracing, Q' is the execution time with tracing lineage at execution time and Q'' is the execution time using a reverse query. As Figure 5 (c) shows, our algorithm incurs an increase of execution time ranging from 20% to 100%. In all cases, our algorithm outperforms the reverse query algorithm by 15% to 40%. Figure 5 (d) similarly shows the performance for an ASPJ query. As the number of tuples returned by the query increases, the execution time increases dramatically. Our algorithm is 13% to 100% faster than the reverse query algorithm. Overall, we see that while there is a space



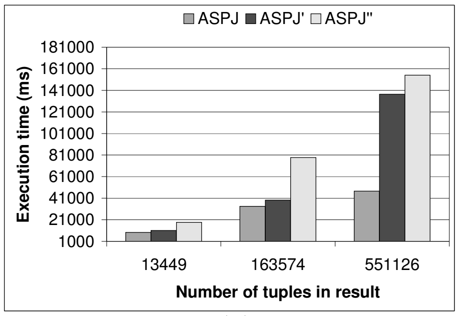
(a)



(b)



(c)



(d)

Figure 5. Space and Performance cost

and time overhead for fine-grained lineage, the benefits for scientists are significant. From the scientific data management perspective, a 20% or even 100% space overhead is of less importance than added functionality. The time overhead for most queries is tolerable given that execution time is not the critical parameter for domain scientists. For the case of ASPJ queries, it is clear that tracing lineage can be a lot slower than executing the queries without lineage tracing. However, it should be noted that even in this case, the execution took only about 150 seconds to generate the results with lineage for a dataset with over 500,000 tuples.

6 Conclusions

The critical need for better tracing of lineage in scientific databases is well known [6]. It is clear that performance is not an issue for most domain scientists – rather the functionality is more important. In this paper, we highlight the importance of maintaining multiple versions of data and tracing fine-grained lineage in support of these needs. We study alternatives for managing versions, and propose a model for the example application of protein annotations. We present query rewriting algorithms for SPJ and ASPJ queries that piggy-back lineage computation with query evaluation. Our models are implemented using PostgreSQL and tested using a large, real dataset from Uniprot. We establish the validity of the approach in enabling relevant queries and study the space and time overheads. While these overheads can be high in some cases, the real gain for scientists is the novel functionality that can allow them to ascertain reliability of derived data, and foster *data-driven* research. To the best of our knowledge, this is the first work that can handle these types of queries for lineage tracing.

References

[1] S. F. Altschul and et al. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res,*, 25(17):3389–402, 1997.

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

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

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

[5] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB J.*, 12(1):41–58, 2003.

[6] P. D. Karp. What we do not know about sequence analysis and sequence databases. *BIOINFORMATICS*, 14(9):753–754, 1998.

[7] M.Zhang, D. Kihara, and S. Prabhakar. Tracing lineage in multi-version scientific databases. Technical Report CSD TR 06-013, Computer Sciences, Purdue University, July 2006. http://www.cs.purdue.edu/homes/mzhang2/.

[8] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, San Francisco, California, 2000.

[9] http://www.pir.uniprot.org/.

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