Deciding how to store provenance

Kiran-Kumar Muniswamy-Reddy

TR-03-06

Computer Science Group
Harvard University
Cambridge, Massachusetts

Deciding How to Store Provenance

Kiran-Kumar Muniswamy-Reddy
kiran@eecs.harvard.edu
Harvard University

Abstract

Provenance of a file is metadata pertaining to the history of the file. Provenance, unlike normal metadata stored in file systems, is retrieved primarily by running queries. This implies that provenance has to be indexed and should have a query interface. We believe that databases are the most appropriate place to store provenance as they provide both indexing and query capabilities.

The goal of this paper is to explore the most appropriate schema and database technology for storing provenance. In the paper we discuss the different possible schemas for storing provenance and the tradeoffs in choosing each of the schemas. We then characterize the behavior of some of the popular database architectures under provenance recording/querying workloads. The database architectures that we considered are: RDBMS, Schemaless Embedded Databases (Berkeley DB), XML, and LDAP. Finally, we present preliminary performance results for the database architecture for provenance recording and some common provenance queries. Our results indicate that schemaless embedded databases have the best performance under most provenance workloads. The results also indicate that RDBMS has the best space utilization under most provenance workloads.

1 Introduction

The provenance of a file is metadata that tracks the file history. In computer terms, it consists of information about the objects that a particular object is based on, the process of creation/modification of an object, etc. For example, consider a process 'P' that reads from files 'A' and 'B', performs some computation, and writes to a file 'C'. Then the provenance of C consists of, the input files 'A' and 'B', the application 'P' that modified the file, the command line arguments and environment of process 'P', the processor type on which 'P' is running, etc.

Provenance metadata like all file metadata needs to be maintained reliably. Provenance metadata, however, has the following additional properties:

- Provenance metadata is primarily accessed by executing queries. Provenance metadata must be indexed to support queries.
- In some cases, provenance must persist even after the data it describes has been removed.

Below we examine various methods that can be used to store provenance. One approach to store provenance is to use **Extended attributes (EA)** [3] to store provenance metadata. EA are arbitrary name-value pairs associated with a file. EA has the advantage that it is maintained by the file system. EA, however, has many disadvantages: first, EA does not provide a query interface nor does it provide indexing, second, EA support is not consistent across all file systems (for e.g., some file systems allow one block worth of EA to be stored per file), and third, EAs are deleted when a file is deleted.

Another approach, taken by astronomers, is to store provenance in-line in the file. While this approach is simple, it does not facilitate any form of querying. Since provenance is mostly about recording ancestors (trees), one approach could be to use existing work and create custom solution. However, while this may be good for some systems, it is not very useful when users are not interested in trees but partial provenance. For the system to be useful, it has to support all kinds of queries well, so a tree only solution is not appropriate.

The most appropriate approach is to store provenance in a Database. A database stores data reliably, allows indexes to be built on top of provenance data, and has a rich query interface that can be used query objects. Since the database is a separate entity from the storage system, provenance is maintained persistently even after data deletion.

This leads to the questions: what database schema is best suited for storing provenance? What database architecture is best suited to store provenance metadata? The database architectures that we considered in this project are: RDBMS, Schemaless Embedded databases, XML, and LDAP.

To answer these questions, we first examine the advantages and disadvantages of various possible schema for recording provenance. Then we characterize the behavior of different database architectures for recording/querying provenance. Finally, we evaluate the performance of the different database architectures under various workloads.

The Provenance schema design is driven by commonly used provenance queries. Based on interactions of our group members with some provenance users and the discussions in the PASS workshop sessions, some of the common provenance queries are as follows:

1. **Retrieve the files with a particular attribute:** An example of this query is, "find me all the files that I ran with *nparticles* as an argument". This query can also be used to avoid running redundant experiments by querying the provenance database for files that have been through the required set of transformations.

2. **Retrieve complete ancestor/descendant tree of a file:** An ancestor of a file 'A' is any file 'B' that was read by a process before modifying 'A'. 'B' is an ancestor of 'A' as 'A' could have possibly been derived from 'B'. Similarly 'A'

is a descendant of file 'B'. Ancestry queries are a transitive closure on the immediate parents of each file. An ancestry query can return the full tree or *partial history* of a file. This is useful to view the source of a file or to generate a script that will reproduce the file. The descendant query is an inverse of the ancestor query. An example use of the descendant query is to notify all the descendants of a file that the source they derived from is corrupt. Files can also be ranked by how many other files depend on them transitively. The file ranking can be used to set backup or replication policies.

3. **Compare the provenance of two files:** If a process outputs erroneous data, this query can be used to find out what went wrong by comparing the provenance of the output with the output of a successful execution.

4. **Time based query:** This query is used to retrieve all the provenance of a file during a particular period of time. An example application of this query to verify changes to a system in a particular time interval.

5. **Retrieve complete provenance of a file:** This query is used to return all the provenance of a file. An example application of this query is in setting Information Lifecycle policies of a file.

Based on these queries, the provenance metadata that needs to be recorded can be broadly classified into the following two categories:

- **Dependency Provenance:** This refers to provenance metadata used to record the the fact that one file (child) was derived from information in another file (parent). For example, if a process reads files 'A' and 'B' and writes to a file 'C'. Then the file 'C' is the child/descendant and 'A' and 'B' are parents/ancestors. Dependency can be stored in two directions: from parent to child and from child to parent. In this paper, we only discuss dependency from child to parent for brevity's sake.

- **Annotation Provenance:** Annotations are assertions made by users/system regarding a file's history. Annotations are recorded as name-value pairs and are generally used to do bookkeeping. For example, a user can annotate a file with summary of the data in the file and later look it up.

Dependency provenance can be used to answer queries 2,3,4, and 5. Annotations can be used to answer queries 1,3,4, and 5.

The rest of the paper is organized as follows. Section 2 discusses the schema choices. Section 3 discusses the database architectures. Section 4 presents preliminary results. Section 5 discusses schema and database backends in other provenance systems and database benchmarks. Section 6 identifies future work and concludes.

2 Schema Discussion

2.1 Dependency metadata

The dependency metadata of a file, as we mentioned earlier, consists of all the files that a particular file could have been derived from, i.e., a record of all the parents of a file. Dependency metadata can be stored using the following two methods:

- **Single dependency record:** In this method, a single provenance dependency record is maintained for every file in the system. When a parent-child dependency relationship is established between two files, the provenance of both the child and parent are retrieved from the database, the provenance of the child is concatenated with the provenance of the parent and stored back into child's provenance record.

- **Multiple dependency records:** In this method, multiple records are stored for each file, each record representing a single parent-child relationship between two files. A new record is added to a file's provenance whenever a dependency between two files is established.

The advantages of using a single dependency record per file are as follows. Queries execute faster due to two reasons: the total number of records are fewer and only one record needs to be retrieved for a file. This schema, however, requires two records to be retrieved every-time a new record is stored. Therefore this schema is inefficient when the provenance generation rate is high. Space utilization is also not very good as mutiple copies of the same metadata exist in many records. Also with time, individual provenance records can become really huge. retrieving the whole record when a user wants to lookup a small part can become inefficient.

The multiple dependency method has the advantage that it doesn't require lookups while recording provenance. This schema is also good when a user only wants to lookup only part of the provenance of the file. For example if a user wants to see all the parents between time 'X' and time 'Y', this schema can be used to lookup the relevant part of the provenance. Queries take longer as provenance is rebuilt by looking up multiple records. Also, there are more records in the database compared to single dependency method, resulting in longer query times. Space consumption is also higher as there are multiple records per file.

We chose to use the multiple dependency method for evaluation. If we know before hand that a workload running on a system generates very few provenance records, and the provenance trees are small, the single record provenance schema is a better choice. But since we cannot predict this before hand, the multiple dependency is a better and safer choice.

2.2 Annotation metadata

An annotation consists of attribute name-value pairs, the name of the annotation and its value. Annotation metadata can be represented in the following methods:

- **Single Table:** All annotations are stored in a single table irrespective of the type of their value. Each record consists of a file identifier, the annotation name and the annotation value. The annotated value is stored in its binary format. Along with the data value, the user should store some metadata to distinguish the data type of the annotation.

- **One table for every annotation type:** In this method, a separate table is used to store annotations with a particular datatype value. For example, all annotations with Integer values are stored in one table and all attributes with String values are stored in a separate table.

- **one table for every "annotation name-type" combination:** Here every annotation name and value type has its own separate table. For example, all annotations with the name *particles* and value of type Integer are stored in one table and annotations with name particles but type String are stored in another table.

The single table method is efficient as it provides a one point lookup and one point storage. But it has the disadvantage that the table cannot be indexed by data values in some databases (like RDBMS and LDAP) as the data type is not fixed. This makes queries based on annotation values inefficient. The second method allows indexing of annotation data values.

The MCS compared the last two methods and found that the third method is good only while searching for a file with 10 or more attributes. Hence we chose to use the second method, one table for every annotation type, for our evaluation.

3 Database architectures

In this section, we examine the characteristics of four different database architectures. The database architectures are as follows:

1. **Schemaless Embedded Database:**

   Schemaless databases provide a simple `put` and `get` interface to store and retrieve data. The put command takes a key and corresponding data (blob) and stores it in the database. The get command takes a key and retrieves the data (blob) stored in the database corresponding to the key. Also the database is a library that links into the user address space, unlike most other databases.

   Since the database is directly embedded into the program, it eliminates the performance penalty of a client-server architecture. The fact that it does not impose any schema on the data and that it links directly into the user program, make it the fastest available implementation.

   One of the disadvantages of schemaless databases is that it does not have a strong query layer and does not support dynamic queries well. Indexes for databases configured with duplicate keys cannot be introduced at any arbitrary point in time. Any such required indexes have to be introduced at design time. This is not a big disadvantage as the requirements of system is generally known before hand.

2. **Relational Database Management System (RDBMS):**

   In RDBMS, a database is a collection of uniquely named tables. Each row of a table contains a set of simple attributes that represent an entity. SQL is the most commonly used language to manipulate data in a RDBMS.

   RDBMS imposes a static schema on the data stored. The schema can be altered by running add/drop column, but it is not as flexible as SED. This constraint allows RDBMS provide users with the ability to run dynamic queries. Users can also add indexes at an arbitrary point after the system has been deployed.

   We expect RDBMS to have the minimal space consumption for dynamic tables. Dynamic tables are tables with variable length columns (VARCHAR, BLOB, or TEXT). In Dynamic tables, each record is preceded by a bitmap that indicates which columns contain the empty string (for string columns) or zero (for numeric columns). Hence NULL or numeric zero values are marked on the bitmap and are not stored to disk. Also, numeric columns are packed and stored. Also, String indexes are prefix compressed. We are using dynamic tables in our evaluations and expect RDBMS to have the best space utilization. RDBMS is, however, heavyweight as it follows the client-server model and we expect it to be slower than Embedded databases.

3. **XML Databases**

   XML databases store and retrieve data in XML format. XML databases have become increasingly popular as they make data portable. The query languages available to query XML databases are XPath, XQuery and XSLT.

   There are two kinds of XML databases:

   - *XML-enabled database:* Data is stored in a traditional database like relational or object-oriented. The interface provided to applications, however, accepts and outputs data in XML format.
   - *native-xml database:* In a native-xml database, the data is stored in XML documents.

   In this paper, we will be evaluating native-xml databases, in particular BDBXML [2], an XML database engine built on a schemaless embedded database. Since it is embedded we expect the insertion performance to be comparable to RDBMS. While the performance of BDBXML will not be as good as an embedded database, it provides better querying and indexing capabilities. However, as it stores data in XML format, the space consumption is higher than embedded and RDBMS.

4. **Lightweight directory access protocol (LDAP):**

   LDAP is a database that is optimized for reading and is designed to be used as a general purpose directory. LDAP is designed to store and query hierarchically organized data. LDAP organizes data in a tree and we explored LDAP to see if we can leverage the internal structure to store dependency information. We, however, found that LDAP does not export the internal structure. As future work, we plan to explore LDAP's internal structure and the possibility of representing dependency information using that. In this report, we only explored the default interface provided by LDAP.

We expect LDAP to be slower than the other databases for the following reasons. LDAP uses transactions for all its operations and does not allow transactions to be turned off. LDAP follows a client server model, but its overheads are higher than RDBMS because the client and server communicate using TCP/IP. RDBMS uses unix domain sockets if possible. Also, the client and server encode their data in BER format. No other database uses similar encoding of data. Also LDAP is designed for read mostly access.

The space utilization of LDAP is also high. It stores data in an encoded format. Also every record needs to have a unique primary key that is not part of the record. It also maintains some addition internal per record. The space utilization of LDAP can be close to that of BDBXML.

However, LDAP is the only database that allows indexes to be created on substring matches. Some implementations of LDAP (like IBM's LDAP implementation) allow indexes on $\leq$ and $\geq$ operations. Hence we expect LDAP to perform better than other database architectures for wildcard match and range queries. LDAP is good for workloads where provenance is generated at a low rate but a lot of range and wildcard queries are run.

3.1 Query interface

All database architectures have not been designed with the same query abilities and interface. Each database architecture has different querying abilities, some being more more powerful than the others. We were able to express all the queries that we considered with equal ease in each of the architectures with the exception of BDB. BDB does not have a query layer and we had to write programs that implemented the necessary functionality. We would however like to state the effort required to write BDB query programs was not very high. In summary, the query capabilities did not really prove to be a hindrance.

4 Evaluation

In this section, we first describe the environment in which the experiments were run. We then describe the workloads we ran and conclude the section by discussing the results.

4.1 Environment Setup

We ran all the benchmarks on a 4-core Intel(R) Xeon(TM) CPU 2.80GHz CPU with 4GB of RAM. All experiments were run on a 40GB 5400 RPM TOSHIBA MK4019GAXB, ATA DISK drive. The machine was running Red Hat Linux 9 with a Red Hat 2.4.20-31.9.progeny.5smp kernel. We recorded elapsed, system, user times, and the amount of disk space utilized for all tests. We also recorded the wait times for all tests. Wait time is mostly I/O time, but other factors like scheduling time can also affect it. The results discussed in this paper are results from single runs. We, however, confirmed that our results are not anomalies by running each experiment at-least 2 times.

4.2 Database architecture representatives

For the evaluations, we used **Berkeley DB release 4.2.27** [8] (BDB) as a representative of schemaless embedded databases as it is the most popular distribution available. We used **MySQL 5.0** as a representative of RDBMS databases as it is the most popular open source RDBMS package. For XML databases, we used **Berkeley DB XML version 2.2** (BDBXML). Among all the open source XML database software available, Berkeley DB XML and myXMLDB [4], a native-XML that runs on top of myXML, were the most stable releases available. We chose Berkeley DB XML over myXMLDB as myXMLDB, apart from being slow, cannot store documents larger than 256MB. We used **OpenLDAP version 2.3.11** [5] (LDAP) as a representative of the LDAP architecture as it the most popular open source LDAP implementation available. LDAP supports various backends and we chose to use a Berkeley DB backend as it is the most commonly used backend.

4.3 Configurations

Initially we ran the experiment using default configurations and the initial results were all I/O bound. We picked a cache size so that the workloads would be non I/O bound [1].

We made the following changes to default configurations of the databases for running the benchmarks:

- **BDB:** We set the cache size to 256MB.
- **MySQL:** We set the size of the `key_buffer`, the buffer used to index blocks, to 256MB. We ran all queries on MySQL by setting the the query cache size to 256MB. Note that the data cache is not configurable in MySQL, instead it relies on the operating system to perform filesystem caching for data reads.
- **BDBXML:** We set the cache size to 256MB.
- **LDAP:** For each of the database in LDAP, we set the cachesize to be 256MB. We further tuned the performance by setting transactions to be asynchronous and by setting maximum log file size to 256MB.

4.4 Workloads

We ran five workloads in all: provenance insertion workload, am-utils provenance insertion, provenance tree query, attribute query, and complete provenance query.

Provenance insertion benchmark: The purpose of this benchmark is two fold. First, to evaluate the database that has the best insertion rate. Second, to provide worst case scenario for the query benchmarks. In the provenance insertion benchmark, we generated dependency information for `n` number of files. For each file, we randomly generated between 1 and 10 ancestor and between 1 and 10 annotations. We chose 1 and 10 because they

[1] We wanted to run workloads such that X% of data is in memory for all backends. But since the space requirements of each database varies considerably, we decided to use a constant cache size and vary the number of records

provide worst case for later benchmarks. For these experiments we restricted ourselves to only two types of annotations: Integer and String. We generated 500 random annotation-names with maximum string length of 25 and always selected an annotation name from this pool. We setup the databases to maintain a total of 8 indexes. For the dependency table, we maintain indexes on the file-id and on the time the dependency was generated. For the Integer and string annotation tables, we maintain an index on the file-id, annotation-name, and the time the annotation was generated. The indexes are designed to speedup annotation-name based queries and time based queries.

We ran this workload 4 times in all, by setting n to 25k, 50k, 75k and 100k respectively. We ran it for different values of n in order to evaluate if the relative database performance varies when the number of records increase.

Am-Utils compile provenance insertion benchmark: In this workload, we read the provenance collected by Provenance Aware Storage System(PASS) [20] for an Am-Utils compile and insert the provenance into the databases. The purpose of this benchmark is evaluate how databases perform under read workloads. The benchmark has only dependency records. The number of records generated by the benchmark are small; the benchmark inserts a total of 10,778 records for 7,313 files. In the future, we plan to increase the number of records by reading and inserting the provenance from a larger workload like Linux kernel compile or a workload that has multiple copies of Am-Utils.

Provenance tree query: For the provenance tree query benchmark, we chose 7,000 files randomly from the data generated by the Am-Utils provenance insertion benchmark and ran a provenance tree query on each of the file. This benchmark serves to evaluate how each database performs for tree queries. The maximum tree depth in this experiment was 333 and a total of 1,493,391 records need to be looked up to complete the benchmark.

Attribute query: In this benchmark, we used the dataset generated for the 50k record insertion benchmark and ran the "retrieve all files with this attribute" query for 2,000 randomly chosen files. This benchmark serves to evaluate how each database performs for attribute-based provenance queries. We chose 2,000 files so that the time required to complete the benchmark is sufficient distinguish the performance the databases. A total of 1,097,333 records need to be looked up to complete the benchmark.

Complete provenance query: In this benchmark, we used the dataset generated for the 50k record insertion benchmark and ran the "retrieve complete provenance of the file" query for 300 and 2,000 randomly chosen files respectively. This benchmark serves to evaluate how each database performs for complete provenance queries. We initially ran this benchmark with all four databases for the 2,000 file case. We chose 2,000 files so that the time required to complete the benchmark is sufficient distinguish the performance the databases. We however, did not run this experiment for LDAP in the 2,000 file case as it was very slow. The 300 file case was chosen to in order to have case where LDAP completes in a reasonable time. A total of 3,241 records need to be looked up to complete the 300 file case. A total of 21,803 records need to be looked up to complete the 2,000 file case.

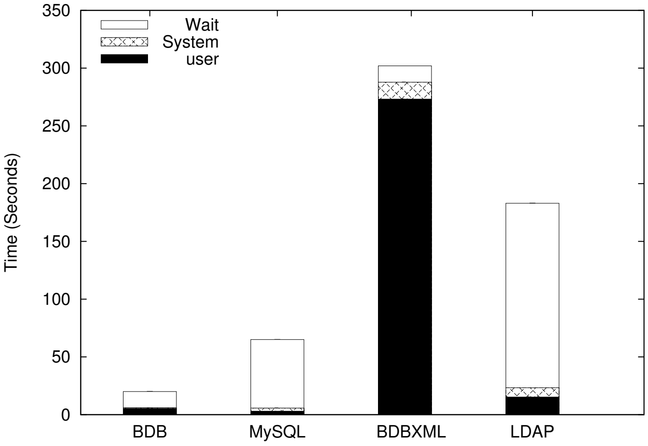

Figure 1: **Provenance Insertion Benchmark for 25k files: Elapsed times.**

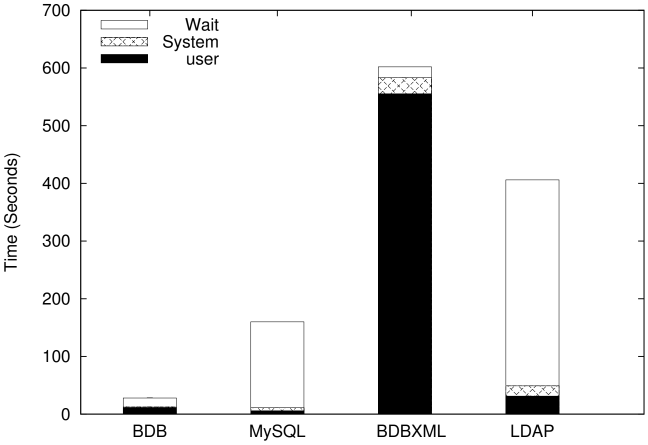

Figure 2: **Provenance Insertion Benchmark for 50k files: Elapsed times.**

Note that we used the same random-number seed for all experiments ensuring that all databases get the same data. Also, we wrote C clients to do the database operations for MySQL and LDAP. We wrote C and C++, programs linked with BDB and BDBXML libraries respectively, to do the database operations on BDB and BDBXML.

4.5 Results

4.5.1 Provenance insertion benchmark:

Elapsed time comparison: Figure 1 shows the elapsed time for provenance insertion benchmark for the 25k files case. The 25k benchmark generates 137,199 dependency records, 69,240 String annotations and 68,514 Integer annotations. BDB has the best performance, as expected. MySQL has the second best per-

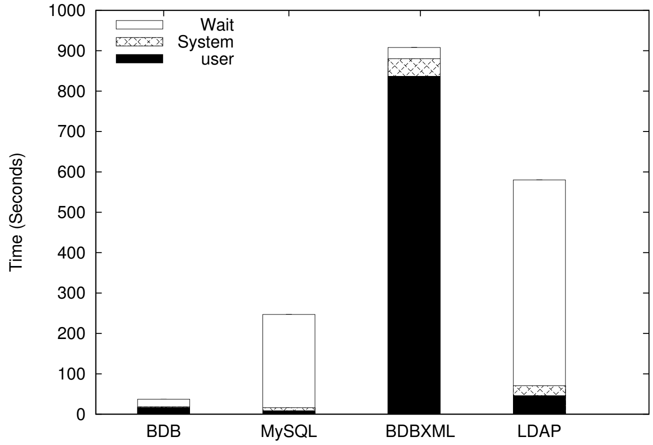

Figure 3: **Provenance Insertion Benchmark for 75k files: Elapsed times.**

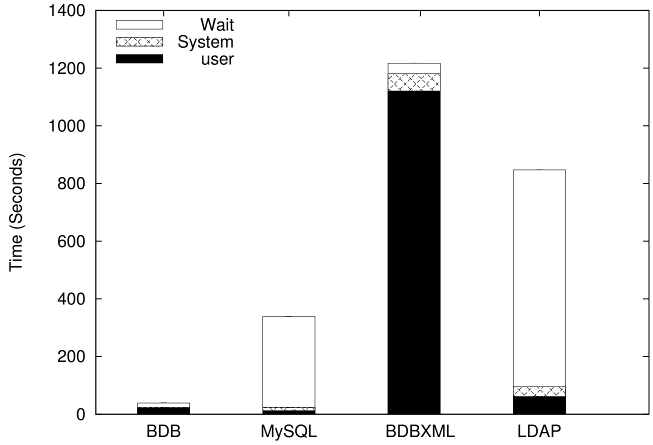

Figure 4: **Provenance Insertion Benchmark for 100k files: Elapsed times.**

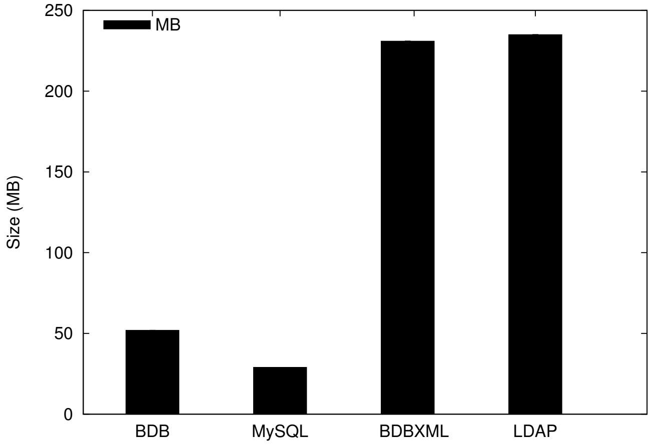

Figure 5: **Provenance Insertion Benchmark for 25k files: Space utilization.**

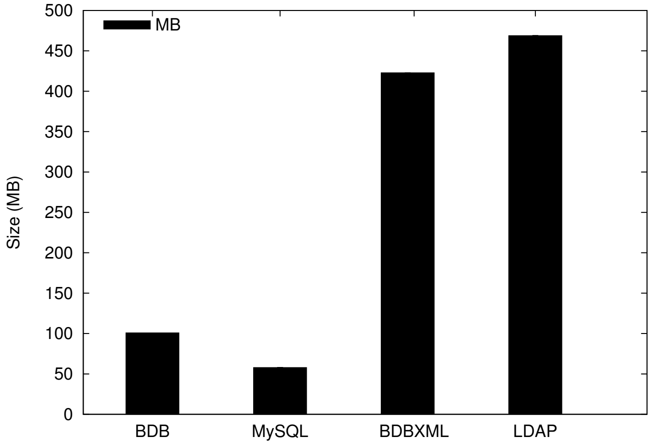

Figure 6: **Provenance Insertion Benchmark for 50k files: Space utilization.**

formance. We expected the BDBXML performance to be close to that of MySQL, but this is not the case. The performance of BDBXML can be explained by the fact that BDBXML elapsed time is CPU bound. The overhead is due to XML parsing. LDAP, surprisingly, does better than BDBXML. This is because we configured LDAP to have a large cache, large log, and made transactions to be asynchronous. We observed that while LDAP is running, the database files are small, but cache files (__db.*) are very large. On shutting down LDAP, it takes several minutes to move the data from `__db.*` files to the database files. So we suspect that LDAP does better than BDBXML, despite having transactions, as most of its data is in a cache.

Figures 2, 3, and 4 show the elapsed times for the 50k, 75k, and 100k file cases respectively. The 50k case generates 273,805 dependency records, 137,922 String annotations and 137,454 Integer annotations. The 75k case generates 411,704 dependency records, 207,135 String annotations and 206,757 Integer annotations. The 100k case generates 548,657 dependency records, 275,515 String annotations and 275,805 Integer annotations. The elapsed times for the 50k, 75k, and 100k are very similar to the 25k files case with only the scale changing. These results seem to imply that increasing the number of records does not change the relative elapsed times of the databases for the insertion benchmark.

Space utilization comparison: Figures 5, 6, 7, and 8 show the space utilization for the 25k, 50k, 75k, and 100k file cases respectively. MySQL, as expected, has the least space utilization in all cases. BDB, does better than BDBXML and LDAP. LDAP excepting for the 75k(Figure 7), has the worst space utilization. In the 75k case, LDAP occupies more space than BDBXML. At this point, we don't know the reason for this.

4.5.2 Am-Utils compile provenance insertion benchmark

Figure 9 shows the elapsed time for the Am-Utils compile provenance insertion benchmark. The relative ordering between the elapsed times of various databases is similar to the previous benchmark: BDB is the fastest, MySQL is second, LDAP third

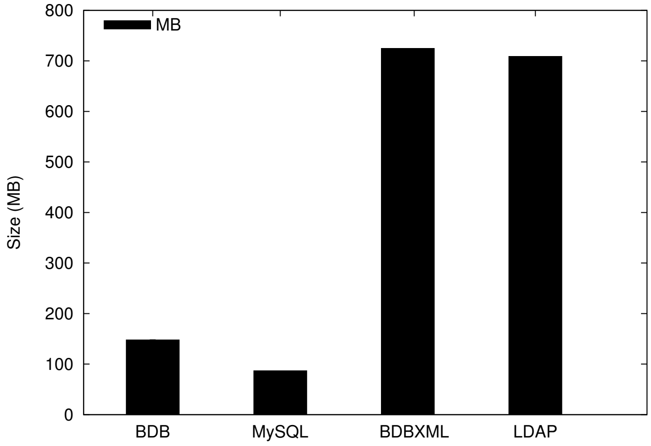

Figure 7: **Provenance Insertion Benchmark for 75k files: Space utilization.**

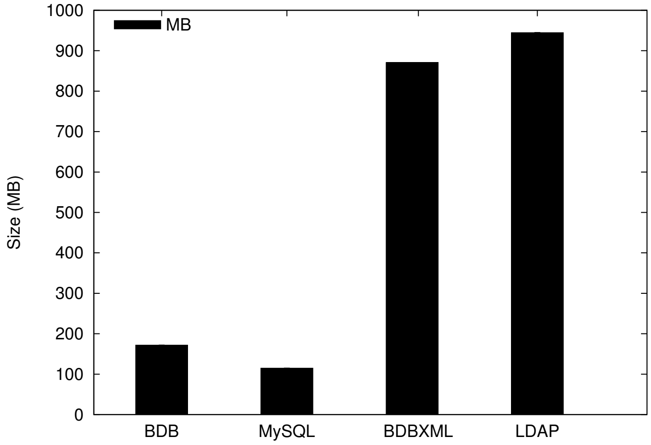

Figure 8: **Provenance Insertion Benchmark for 100k files: Space utilization.**

and BDBXML the slowest. In fact, the Figure 9 looks like a scaled down version of the elapsed time figures of the insertion benchmark. This shows that BDB is the fastest even on real workloads.

Figure 10 shows the space utilization for the Am-Utils compile provenance insertion benchmark. The relative ordering between the space utilization is similar to the previous benchmark: MySQL has the least space utilization, BDB is second, BDBXML third, and LDAP has the highest space utilization. However, the difference between the space utilization of MySQL (1.1MB) and BDB (1.2MB) is much smaller than the insert benchmark. While the data present in this benchmark is not sufficient make a conclusion, it might imply that for real workloads, the space utilization of BDB might be much closer to MySQL. This should be explored by running more real workloads.

4.5.3 *Provenance tree query benchmark:*

Figure 11 shows the elapsed time for provenance tree query. BDB has the best performance and it takes 5 seconds to complete the benchmark. MySQL is second and takes 1 minute and 44 seconds to complete the query. BDBXML is third behind MySQL and takes 1hr and 18minutes. BDBXML is very CPU bound and spends all its time doing CPU computations. We don't have LDAP numbers in the graph as LDAP did not finish the benchmark even after 12 hours. LDAP spends most of its time doing I/O. At this point, we are not sure why LDAP is so slow. We think that it could be due to protocol overhead and the fact that LDAP uses transactions even for read queries. However, we cannot rule out a bug in the LDAP configuration.

4.5.4 *Attribute query*

Figure 12 shows the results of the attribute query. BDB and MySQL perform the best on this query and take 3.05 and 2.94 seconds respectively to complete the benchmark. BDBXML and LDAP are an order of magnitude slower and take 33 minutes and 43 minutes respectively to complete the query. Again BDBXML is very CPU intensive and LDAP is I/O intensive. However, we could not think of a reason why LDAP does well on this query

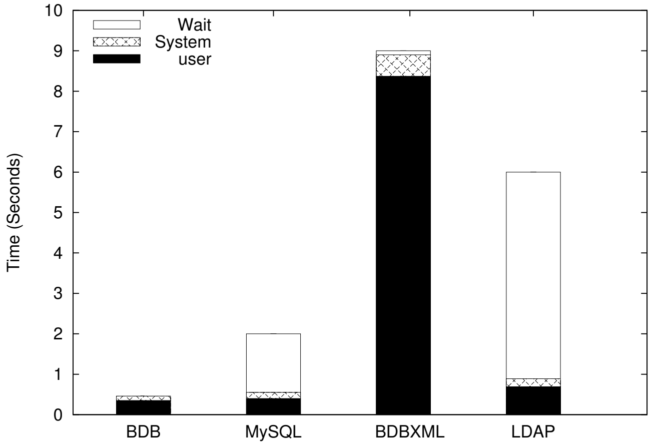

Figure 9: **Am-Utils compile provenance insertion benchmark: Elapsed times.**

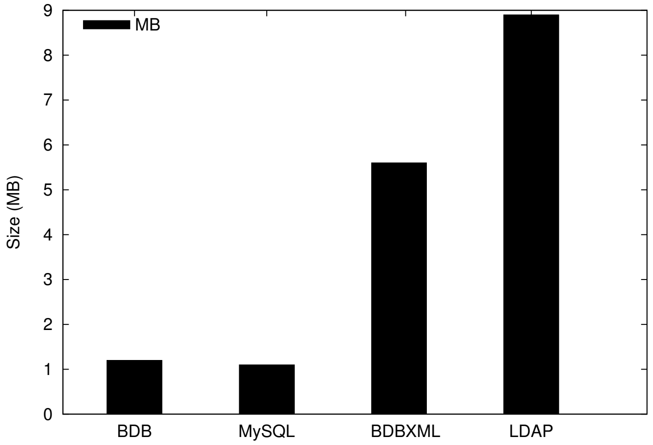

Figure 10: **Am-Utils compile provenance insertion benchmark: Space utilization.**

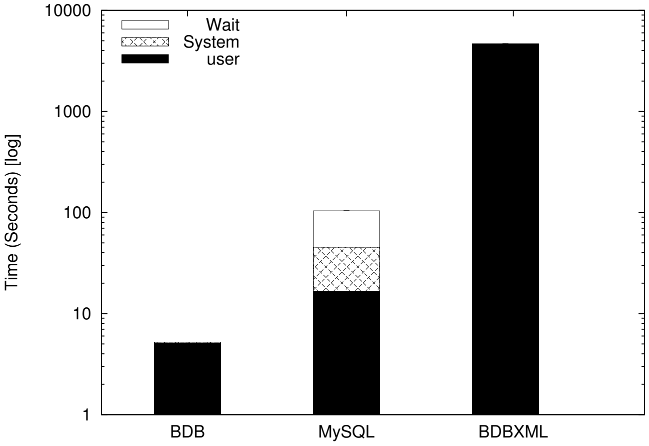

Figure 11: **Provenance tree query benchmark results: Elapsed times.**

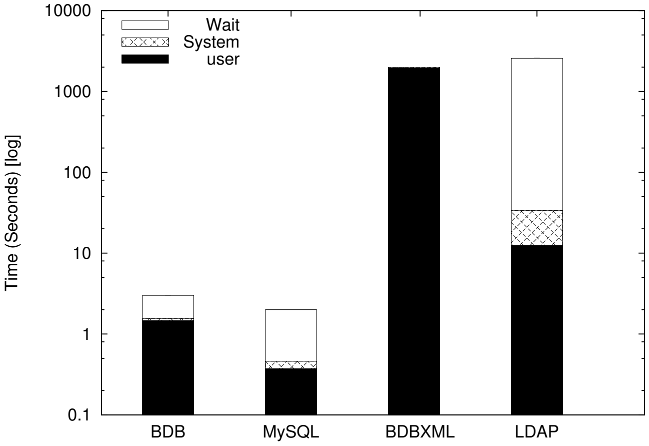

Figure 12: **Attribute query results: Elapsed times.**

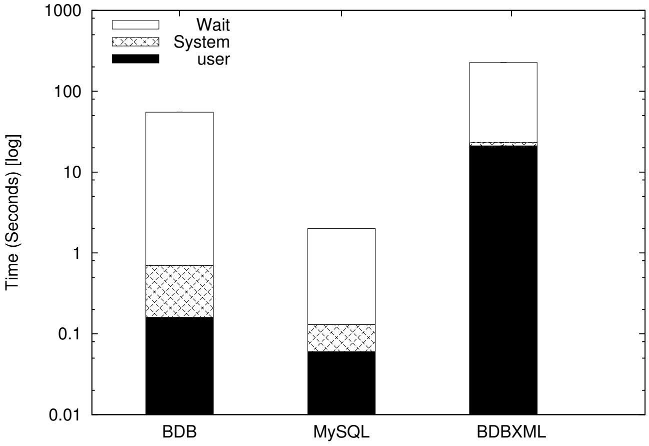

Figure 13: **Complete provenance query results with 2,000 files: Elapsed times.**

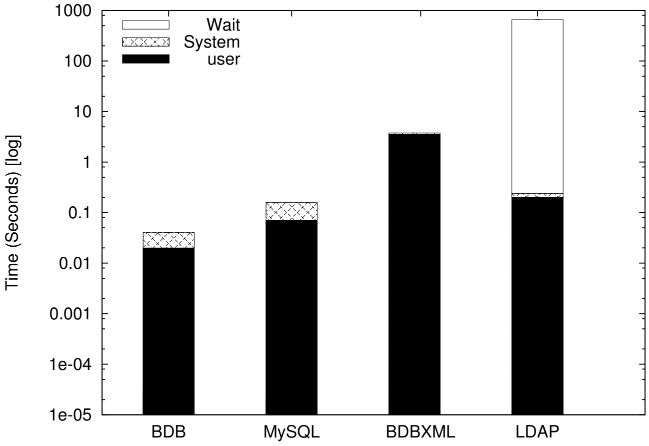

Figure 14: **Complete provenance query results with 300 files: Elapsed times.**

while it does so poorly on the others.

4.5.5 Complete provenance query

Figure 13 shows the results of the complete provenance query when querying for the complete provenance of 2000 files. We ran this benchmark for all databases except LDAP. This is because LDAP was taking too long for 2000 files. MySQL has the best time in this benchmark (2s). BDB takes 55s to complete the benchmark. The reason that MySQL does better than BDB is because we used a partial key search (DB_SET_RANGE) in BDB. For BDB, the key for this database is of the form "file-id+attribute". Re-organizing the data schema so that the key is "file-id" will make the performance of BDB comparable or better than MySQL. BDBXML is CPU intensive due to all the XML parsing and is slower than MySQL and BDB. BDBXML takes 227s to complete the benchmark.

Figure 14 shows the results of the complete provenance query when querying for the complete provenance of 300 files. We ran this scaled down version of the benchmark as LDAP completed in a reasonable the benchmark in a reasonable amount of time. MySQL has the best time in this benchmark (0.16s). BDB takes 0.31s to complete the benchmark. The reason that MySQL does better than BDB is because we used a partial key search (DB_SET_RANGE) in BDB. Re-organizing the data schema will make the performance of BDB comparable or better than MySQL. BDBXML is CPU intensive due to all the XML parsing and is slower than MySQL and BDB. BDBXML takes 3.87s to complete the benchmark. LDAP again is I/O intensive and is an order of magnitude slower. LDAP takes 11 minutes to complete the benchmark. This again could be because LDAP uses transactions or due to protocol overheads.

In Summary, BDB has the best query performance in the experiments that we conducted. MySQL has the best space utilization in most cases except for real workloads where BDB is comparable to MySQL. However, before passing a final verdict, we need to reorganize our data and run more realistic workload benchmarks. We also need to evaluate the databases for two more query workloads: time based queries and partial string name queries.

5 Related Work

In this section, we first provide a broad overview of the database architectures and schemas used by provenance systems in other domains. We then discuss existing database benchmarks and the applicability of their results.

5.1 Databases used in current provenance systems

A summary of the databases used by current provenance systems along with some other properties are listed in Table 1. Most existing provenance systems can be broadly classified into two categories: Workflow-based provenance systems and Annotation-based provenance systems. ESSW [17] and Lineage file system [18] are two systems that don't fall under either of these categories.

Workflow based provenance systems: In many existing Grid/Service workflow based systems, the recorded provenance is generally a variant of the following: when a service is invoked, the workflow manager stores the input, output and the service name in a provenance repository. Also, logically related service invocations are grouped by using a common ID.

The disadvantage with this schema is that it is sub-optimal for running provenance tree based queries. This schema doubles the depth of the provenance tree, as tree traversals first have to lookup the process, then lookup the inputs to the process to get to the parent file.

Systems that have a workflow based provenance include: PASOA [19], Chimera [16], and myGrid [25]. The database backends used by these systems and their query interfaces are listed in Table 1.

Annotation based provenance systems: Systems like the Metadata Catalog Service (MCS) [14], the Storage Resource Broker (SRB) [22, 1], and GeoDise [13, 23] store provenance in name-value attribute pairs. While this schema can be used to build provenance trees, the construction is again inefficient for the same reasons as in workflow-based provenance systems. However, MCS proposes and evaluates two schemas that we explore in Section 2. Again, the database backends used by these systems and their query interfaces are listed in Table 1.

ESSW: Earth System Science Workbench (ESSW) [17] is a data management infrastructure that is used to keep track of the processing of locally received satellite imagery. ESSW has an XML frontend. The parent/child relationship is recorded by storing the ID's of the two objects in a binary relation. ESSW uses Informix Dynamic Server Object-relational DBMS. It has also been developed for MySQL. The lineage of an object can be accessed using a web-browser by specifying its ID or just browsing to it. A user can also specify the number of levels of forward or backward lineage to be retrieved. lineage is retrieved using recursive SQL queries on the lineage parent/child RDBMS table. The lineage can be displayed as a graph of the objects.

Lineage File System: The lineage file system [18] tracks provenance at file system granularity. Lineage file system tracks provenance by logging all process creation and file-related system calls in the `printk` buffer. A user-level daemon processes the log and inserts the records into a MySQL database. The database has two tables; one for storing process command lines, pid, uid, and euid; and second, for storing the name of a file, pid of process that opened it, the open and close timestamp, and the file open mode. The pid consists of the pid of the program concatenated with the timestamp. The lineage is retrieved by doing a join on the pid in the two tables. Ancestry queries are again inefficient with this schema.

5.2 Database provenance systems

A number of research projects including Trio [24], Buneman's work [12], and Tioga [21] have focused on providing provenance for tuples in a database system. Since our work is focused on recording provenance at a file level granularity, these works ultimately complement our work.

5.3 Database Benchmarks

There are numerous database benchmark packages available that are used by database vendors to benchmark their system. Some of the benchmark suits are: Nile [15] benchmark used to compare database performance for ecommerce applications, SysBench [9] designed to simulate generic OLTP workload, Open Source database benchmark (OSDB) [10] is yet another benchmark used to measure performance. Unfortunately, most of these benchmark suits cannot to be used to reliably predict the performance of the system for particular applications like provenance storage and query.

The benchmark that is closest to that required for this project is the TPC-H [11]. The TPC-H consists of a suite of business oriented ad-hoc queries and concurrent data modifications. TPC-H is used to evaluate systems with large volumes of data and systems that execute queries with a high degree of complexity. The performance metric reported by TPC-H is the TPC-H Composite Query-per-Hour Performance Metric. Open source development labs Database Test Suite(DBT) [6], consists of a set of workloads inspired by the TPC benchmarks, the difference being that it is free. The DBT-3 workload corresponds to the TPC-H workload.

6 Conclusions

In this paper, we discussed some possible schemas for storing provenance and discuss the tradeoffs in choosing each of them. We also discussed possible database architectures and the advantages and disadvantages of storing provenance in them. Our evaluation indicates BDB has the best elapsed time performance under provenance workloads and that MySQL has the best space utilization.

6.1 Future work

Resource Description Framework (RDF) [7] RDF is gaining popularity in the provenance world as a means to store provenance. Data is stored in RDF triples, of the form subject-predicate-object, which can be used to naturally express dependencies. We plan to evaluate RDF as a possible option for storing provenance in the future.

	System	Backend	Representation Schema	Query Interface	Workflow based	Annotation based	Tree Queries
1	MCS	RDBMS	Tables	Client API		✔	
2	SRB	RBDMS	Tables	Browser		✔	
3	PASOA	Berkeley DB JE	XML	Xquery	✔		
4	Lineage FS	RDBMS	Tables	SQL queries	✔		
5	Chimera	Virtual Data Catalog / RDBMS	Virtual Data Language annotations	Queries	✔		
6	MyGrid	MIR repository / RDBMS	XML/RDF	Browser	✔		
7	ESSW	Lineage Server / RDBMS	XML/RDF	Browser	[a]		✔
8	GeoDise	RBDMS	XML	Web-services			

Table 1: **Summary of database architecture and interface used by existing provenance systems.**

[a] Records dependencies similar to PASS.

A check mark in the last three 3 columns indicates that the column is true/supported, otherwise it is not.

LDAP has been internally optimized to store trees but it does not export this to the external world. The internal structure of LDAP should be studied to see if provides an advantage for storing dependencies.

A Hybrid approach where dependency information is stored in one architecture and annotation in another would be interesting to explore.

Last, we need to do a more rigorous evaluation of the system by running some more queries. For example, run wildcard queries on annotation attributes and values, time based queries, etc.

References

[1] http://www.npaci.edu/DICE/SRB/README.zones.

[2] Berkeley DB XML. http://www.sleepycat.com/products/bdbxml.html.

[3] Linux Extended Attributes and ACLs. http://acl.bestbits.at/.

[4] myXMLDB. http://myxmldb.sourceforge.net/.

[5] OpenLDAP. http://www.openldap.org/.

[6] OSDL Database Test Suite. http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/.

[7] Resource Description Framework (RDF) . www.w3.org/RDF.

[8] Sleepycat Software. http://www.sleepycat.com.

[9] SysBench: a system performanace benchmark. http://sysbench.sourceforge.net/.

[10] The Open Source Database Benchmark. http://osdb.sourceforge.net/.

[11] TPC-H. http://www.tpc.org/tpch/default.asp.

[12] P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *International Conference on Database Theory*, London, UK, Jan. 2001.

[13] S. Cox, Z. Jiao, and J. Wason. Data management services for engineering. 2002.

[14] E. Deelman, G. Singh, M. P. Atkinson, A. Chervenak, N. P. C. Hong, C. Kesselman, S. Patil, L. Pearlman, and M.-H. Su. Grid-Based Metadata Services. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM04)*, June 2004.

[15] e-week's Nile benchmark. http://www.eweek.com/article2/0,4149,293,00.asp.

[16] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR*, Asilomar, CA, Jan. 2003.

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

[18] Lineage File System. http://crypto.stanford.edu/~cao/lineage.html.

[19] Provenance aware service oriented architecture. http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome.

[20] M. Seltzer, K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and J. Ledlie. Provenance-Aware Storage Systems. Technical report, Harvard University, July 2005. Computer Science Technical Report TR-18-05.

[21] M. Stonebraker, J. Chen, N. Nathan, C. Parson, A. Su, and J. Wu. Tioga: A database-oriented visualization tool. pages 86–93.

[22] M. Wan, A. Rajasekar, and W. Schroeder. An Overview of the SRB 3.0: the Federated MCAT. http://www.npaci.edu/DICE/SRB/FedMcat.html, September 2003.

[23] J. Wason, M. Molinari, Z. Jiao, and S. Cox. Delivering data management for engineers on the grid.

[24] J. Widom. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *Conference on Innovative Data Systems Research*, Asilomar, CA, January 2005.

[25] J. Zhao, M. Goble, C.and Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-science.