Provenance Data Storage

Peter Macko
pmacko@fas.harvard.edu

Nicolas Ward
nward@fas.harvard.edu

Abstract

Provenance research has generally focused on issues with data collection and organization. Most approaches represent stored provenance data as a directed acyclic graph (DAG), where objects such as files and processes are nodes in the graph and directed edges specify ancestry relationships between them. While there has been some work addressing logical compression of these provenance graphs, efficient physical storage of provenance data remains unaddressed. In approaching this problem, we implemented and evaluated several techniques tailored for provenance storage, which were inspired by existing representations of general semi-structured data. We considered variants of vertical partitioning, PASS, and RDF, varying two kinds of compression. We compared query runtime, disk usage, and data load time across these storage methods. Our results indicate that vertical partitioning performs best in most cases, while the benefit of compression varies by query.

1 Introduction

In computer science research, provenance refers to the origins of, and transformations applied to, a specific segment of data. Current provenance research is focused in several areas, including collecting provenance [5, 22], improving storage methods [8], and querying provenance data [13]. Many of these efforts are motivated by other fields of scientific inquiry, where determining whether a data point is an interesting outlier or experimental error determines the validity of a result. This result auditing is an important step in reproducibility and is enabled by collecting and storing provenance during data processing.

In addition to the sciences, provenance is being used to improve human-computer interaction (HCI) research, by enabling a study to better track a user's interface actions over time, and in particular, the higher-level operations performed by a user. For the end user, adding web browser provenance can provide additional context to improve web search results [18].

Both of these areas of potential application of provenance require efficient storage of the provenance data. For the physical sciences, astronomical sky surveys or protein structures quickly surpass terabyte scales [3]; even with increasingly cheap storage technology, we need to be able to store the provenance of this data efficiently. For HCI, responsiveness and unobtrusiveness are both important to the user experience, so we require fast query runtimes and even faster provenance store inserts when collecting data for user actions.

We quantify efficiency by emphasizing low-latency query run times and fast database loading, while also seeking a reduction in on-disk space usage. When these factors conflict, we prefer improved user interactivity. The semi-structured provenance data is typically represented as a directed acyclic graph (DAG). Existing work in provenance efficiency has focused on logical improvements, such as reducing replication of subsets of the graphs [8], or applying general graph compression techniques [13]. Disk space is in general very cheap, but even small datasets grow by an order of magnitude or more when producing a provenance graph [8]. For example, the MiMI dataset is 270 MB of protein structure data, while the provenance for that data occupies 6 GB [17].

In considering all of these factors for efficient provenance processing, storage, and querying, we note that the space is as yet largely unexplored. We will demonstrate a few generalizable techniques for improving provenance storage performance as per the above metric, which we hope will serve as a "jumping off point" for further research supporting provenance.

The paper is organized as follows. We describe the general problem of provenance storage in Section 2, and focus on the storage approaches we have implemented in Section 3. Then in Section 4, we describe our experimental set up, whose results are presented in Section 5. Section 6 contains a summary of related work. We conclude the paper in Section 7.

2 Background

2.1 Provenance Data Model

Our work focuses on the data model used by the provenance research group at Harvard University [21]. Provenance is conceptually represented as a directed acyclic graph (DAG), where each node corresponds to some object, or a version of an object. For example, when tracking provenance at the system call level, such as in PASS [22] and ES3 [5], the objects are files, processes, and Unix pipes. At the workflow level, the objects of interests are files, operators, ports, etc. Every node has its own set of properties, most importantly, `NAME` and `TYPE`. Specific examples of additional properties include `PID` (for processes), `INODE` (for files), or `executableFile` (for workflow operators calling external programs).

There are two kinds of edges between the nodes. The first and most important kind are the `INPUT` edges, which define the ancestry relationships between objects. For example, executing the Unix command `cp a.txt b.txt` as in Figure 1, `a.txt` is an input to `/bin/cp`, which in turn produces `b.txt`. The second kind are `PREV` edges, which connect different versions of the same object created as a part of a cycle-breaking algorithm [6]. Cycles in the provenance graph are broken by introducing new versions of objects, such as files or processes. For example, if a file A is an input of file B, and then file B is an input of A, a provenance analyzer breaks the cycle by causing the write from B to A to introduce a new version of A. Consequently, the provenance graph has one node for B and two nodes for each version of A.

Equivalently, a provenance record can be expressed as a set of RDF triples, as shown in Table 1. Each ancestry edge between two nodes A and B is represented as (A.id, INPUT, B.id), and every property of a node A is stored as (A.id, key, value).

2.2 Compression Techniques

The provenance datasets we tested (see 4.2) contain many string literals, which are frequently repeated and are often long (more than 80 characters). Because of this dataset feature, we apply dictionary encoding, which replaces each string literal with an integer ID [1, 2, 23]. In the case of an RDF storage model (see 2.1), this simple approach compresses the size of individual triples by a factor of 20 or more, from hundreds of single-byte characters to only three 32-bit integers.

It is possible to compress the triples further by applying null suppression as is done by Abadi et al. [1]. The fundamental idea is to delete consecutive zero bytes in data and replace them with a description of how many there were and where they existed. This approach allows us to remove leading null bytes in an integer, so instead of using a full 4 bytes, we store the integer in the exact numbers of bytes (1, 2, 3, or 4) necessary and store the number of removed zeros as two bits in the block header. It is possible to achieve a better compression ratio by removing all leading null bits of an integer, but according to Abadi et al. [1], this would cause a major performance hit, as all the queries would quickly become CPU bound. Our initial experiments showed that encoding integers at the bit level is two times slower, and decoding is approximately ten times slower. This performance cost does not justify the minimal space savings we get in return, which we estimated to be between 5% to 10% using statistics gathered during data loading (see 5.1). This effect is most likely due to instruction sets designed to make operations at byte boundaries be the most efficient.

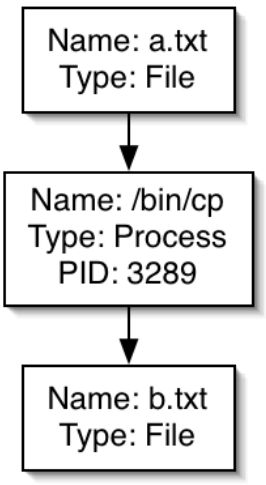

Figure 1: **Provenance Example**. An example of a simple provenance graph for the Unix command `cp a.txt b.txt`.

The average size in bytes of a provenance node varies from dataset to dataset. Representing node IDs as 32-bit integers sets an upper bound on the number of nodes that can be represented; for example, if we assume an average node size of 400 bytes, then a dataset exceeding 1.6 TB would likely have more than 2^{32} provenance nodes, and we would overflow the 32-bit integer node IDs. Many large scientific datasets would exceed this size limitation, so switching to 64-bit integer IDs would be necessary.

(1, NAME, a.txt)
(1, TYPE, File)

(2, INPUT, 1)

(2, NAME, /bin/cp)
(2, TYPE, Process)
(2, PID, 3289)

(3, INPUT, 2)

(3, NAME, b.txt)
(3, TYPE, File)

Table 1: **Provenance RDF Triples**. For each provenance node, we have subject-predicate-object triples storing the integer node ID, the predicate, and the value.

3 Evaluated Storage Methods

There is a large space of approaches for storing semi-structured data, which we could not possibly cover entirely in a single research project. Storing semi-structured data in B-trees appears to be one of the most popular approaches that do not use a traditional database system as a storage backend. Even within this space, there are many decisions to be made:

- **Predicate Partitioning**: It is possible to store all key/value pairs that describe a provenance record in a single B-tree, with one B-tree for each predicate, or a mixed approach where some predicates have their own B-trees, while the rest are grouped in a single B-tree.
- **Compression**: Previous work [1, 2, 23] has shown the value of compression in both reducing the footprint of the database on disk and more importantly, in improving query performance. Abadi et al. [1] provide a good discussion of the various compression and encoding schemes applicable to database systems.
- **Indexes**: The query performance obviously also depends on the availability of indexes, but there is a trade-off between the disk footprint and query performance.

We have chosen several storage methods and their variations that occupy different points in this decision space.

We initially also experimented with storing the provenance graph in XML. It is possible to treat each node as an object with properties such as ancestor node, node properties, arguments, and so on. This hierarchical arrangement can be stored as an XML document, which can in turn be queried using standard technologies such as XQuery. We experimented with storing an entire graph as a single document containing node elements, but this suffered from scalability problems. In addition to the I/O cost of reading and parsing an XML document, the parsed objects suffer a large space inflation from the on-disk document size, which quickly consumes gigabytes of main memory. This makes loading a test set such as the NetBSD data infeasible using a single XML store. Consequently, we decided to stop pursuing this path.

3.1 RDF-Inspired Models

We represent the provenance graph as RDF and store all triples in a compressed clustered B-Tree, as inspired by Neumann et al. [23]. The triples are sorted lexicographically by subject, predicate, and object strings. This allows easy lookup of values by the tuple `(node id, property name)`, as well as efficient retrieving of all properties of a particular node. In addition, we use another index, in which we store the triples in a different sort order (object, predicate, subject), which is especially useful for looking up provenance nodes by their property values. This approach does not allow for efficient retrieval of all subject-object pairs which have a given property defined.

We also experimented with using delta encoding as advocated by Neumann et al. [23]. This compression method encodes an RDF triple as the difference from the previous triple in the database, producing three delta values, one for each component of the encoded triple. These delta values tend to be small (an offset of less than 256, which can be stored in 8 bits or fewer), so combining this approach with null suppression can produce significant space savings. We eventually decided to abandon this idea, because our preliminary experiments showed that the cost of incremental updates to a database is prohibitively slow, causing most insert statements to be CPU bound.

3.2 PASSv1

The provenance research group at Harvard University stores the collected provenance in several BerkeleyDB databases (in relational database terminology, these would be called tables). Both versions of their system use different database schema. We implemented the BerkeleyDB scheme used by the first version of their system (PASSv1) and adapted it as needed to support our datasets.

The first version is particularly interesting to study, since it stores provenance in a denormalized

Database	Key	Values
Map	object name	p-node number
Provenance	p-node number	provenance record (table 3)
Argument Data	record id	command-line text
Argument Index	argument	p-node numbers
Properties	p-node, key	property value
Provenance Index	p-node	p-nodes of children

Table 2: **BerkeleyDB databases for PASSv1**. The first four databases were part of the original implementation by the PASSv1 group. We introduced the last two databases in order to adapt the storage system for our datasets.

Field	Description
Name	the object name
Type	the object type
Input	the p-node of a parent
Prev	the p-node of the previous version of the object
Env	*Argument Data* record id for environment
Args	*Argument Data* record id for arguments

Table 3: **PASSv1 provenance record**. A slightly modified version of a provenance record from the PASSv1 model.

form: each ancestor-descendant edge is stored as a record in a database together with all the properties of the descendant. Thus, the vertices of the graph are not stored separately, but they are already prejoined with some edges (vertices that do not have any ancestors are stored as edges with parent `NULL`). The original implementation by the provenance group at Harvard stored only the provenance of persistent objects, such as files. Given our selected input data, our implementation must also store transient objects like processes and workflow operators.

The system also uses dictionary encoding on command-line arguments and environment variables, which are usually long and frequently repeating. Furthermore, there is an index that maps individual command-line arguments to provenance node IDs.

We had to make several modification to this schema in order to enable it to store our two test datasets. Most importantly, we added a database for node properties from the input dataset that were not representable in the original schema. We also added an index that allows efficient traversal of the graph in parent-to-children fashion. The schema together with our modifications is summarized in Tables 2 and 3.

3.3 PASSv2

The second generation of the PASS storage backend uses a more typical approach that does not use denormalization as in PASSv1. This system lies somewhere in the middle of the predicate partitioning continuum. `NAME`, `INPUT`, and `PREV` properties are stored in separate BerkeleyDB B-trees, and then there is one large table for all the other properties. There are two indexes by default: an index on `INPUT` and an index on command-line arguments.

PASSv2 has a unique method for encoding command-line arguments. Each word is encoded separately using a dictionary, and then the entire command line is expressed as an ordered list of such numbers. For example, consider the Unix command `make -w all`. If `make` is encoded as 24, `-w` as 11, and `all` as 49, the entire string is then stored as a triple (24, 11, 49). An analogous approach is used to encode a list of environment variables, where each key-value pair is encoded separately.

3.4 Vertically Partitioned Store

Storing large datasets in columns rather than rows has become increasingly popular in the data warehouse community last few years, and it was only a matter of time when this approach is applied to semi-structured data. Abadi et al. [2] create a table for each RDF predicate and then populate it with the corresponding (subject, object) pairs. For example, the provenance graph in Figure 1 would be stored in the following four tables:

- `NAME`: (1, a.txt), (2, /bin/cp), (3, b.txt)
- `TYPE`: (1, File), (2, Process), (3, File)
- `INPUT`: (2, 1), (3, 2)

- PID: (2, 3289)

This table can be stored either in two B-trees (as in the RDF model) or in a column-store table. We have decided to implement the first approach, because then our results would be directly comparable to the aforementioned RDF approaches (see 3.1). We have implemented two variants of this approach: the first uses dictionary encoding only for command-line arguments and environment variables (as in the PASS models), while the second uses dictionary encoding and null suppression for all strings. We use the PASSv2 method for encoding command-line arguments and environment variables.

4 Experiments

In our experiments, we evaluate the different approaches for storing semi-structured data. We measure the query performance and the system footprint (on-disk storage size and database load time) and compare these across storage methods.

4.1 Experimental Setup

We are using a dataset generated by the provenance research group at Harvard, which contains the provenance of compiling parts of NetBSD [21]. We additionally include a scientific dataset derived from D2K data published by the National Center for Supercomputing Applications (NCSA) [11], which was used by one of the teams at the First Provenance Challenge [20]. We designed our own query sets (see 4.2), because we could not find any standard queries for NetBSD dataset, and the provided D2K queries execute very quickly, rendering them unsuitable for our comparative benchmarking. Instead, we selected heavy query workloads that exercise different parts of the storage implementation. To the best of our knowledge, there are no other standard provenance datasets or queries.

Our work uses Path Query Language (PQL) [14, 15], which is the current interface to PASS [21]. This engine has an interface for interaction with a backend database or a storage engine. It consists of functions for initialization of the backend, clean up, inserting new provenance records, and accessing and modifying existing records. All of our storage backends implement a PQL-compatible interface. We designed our benchmark queries in PQL, but we created the query plans ourselves in order to better exploit the features of the storage backends. The PQL interface does not currently implement these customizations.

We compress the provenance data using structural inheritance as described in Chapman et al. [8], which helps us to avoid unnecessary space overhead when thawing new versions of objects. "Thawing" refers to incrementing the version number of a file when opening it for modification, and each thaw creates a new instance of the provenance node and its attributes [6]. The datasets we examined use cycle-breaking on their provenance graphs, which results in new versions of some nodes. Creating a new version of a node is expensive, since all of its attributes must be copied over to the new version, which could be avoided entirely by structural inheritance. We found that the extra overhead in query execution due to inheritance is negligible when compared to the query performance on the bloated datasets without structural inheritance. The way we used structural inheritance is very similar to the approach in the PASSv2 system [21].

We utilized a single desktop tower machine running CentOS GNU/Linux 2.6.18 on an Intel Core 2 Duo E6550 running at 2.33 GHz with 2 GB of RAM and a single server-grade hard drive. We compiled our own instance of BerkeleyDB version 4.7.25 (using GCC 4.1.2).

For all experimental runs, we set the BerkeleyDB cache size to 512 MB. We forced a cold cache by restarting the test machine between each run. This ensures that repeated accesses of the same input dataset or database file do not speed up by getting stored in the I/O cache.

4.2 Datasets and Queries

We tested the performance of each of the storage engines on two datasets by measuring the time it takes to answer our queries, which were selected in order to test different aspects of the system. We hand-crafted a query execution plan for each query for each system using the following optimization heuristics:

1. Use access paths (indexes) whenever possible to avoid a full database scan
2. Batch reads of records within one table (BerkeleyDB database) and execute them in order of the access keys, which minimizes the disk seeks
3. When using dictionary encoding, delay translating the ID numbers to strings as much as possible

4.2.1 D2K Dataset

The original D2K dataset contains a provenance of one scientific workflow execution that processes scans

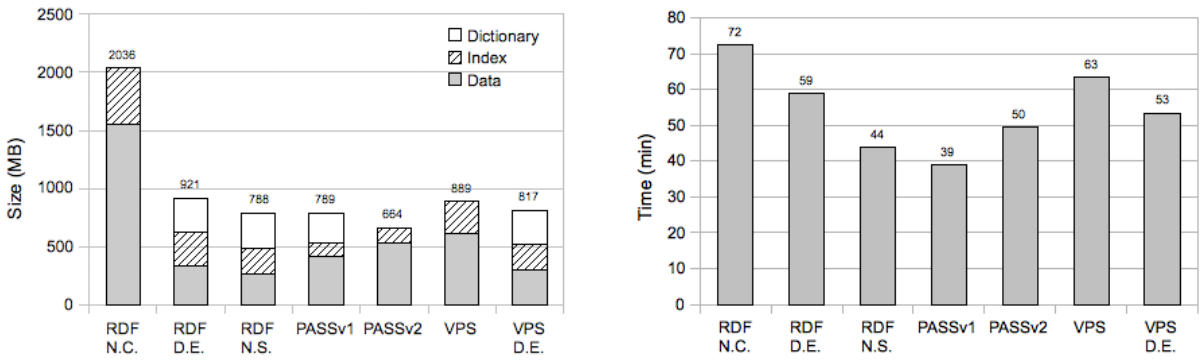

Figure 2: **Loading 1 GB of the NetBSD dataset**. The two figures compare the physical size of the database and query times for our approaches, RDF with no compression of attribute values (RDF N.C.), RDF with dictionary encoding (RDF D.E.), RDF with dictionary encoding and null suppression (RDF N.S.), a modified first version of PASS (PASSv1), the current version of PASS (PASSv2), a vertically partitioned store (VPS), and a vertically partitioned store with dictionary encoding (VPS D.E.).

of a brain [11]. In many scientific applications, scientists use the same workflow a large number of times on different input datasets, producing thousands of results. In order to simulate this, we took the D2K dataset and derived from it an equivalent dataset that contains provenance of ten thousand independent executions of the workflow. The total size of the dataset is 391 MB. This test set expansion was generated by replicating the provenance workflow with random text appended to selected string literals, so that it is not merely repetition of identical data.

We tested the dataset using the following two queries:

Query 1: Find all result files "atlas-y.gif" (possibly in different directories) that were produced as a consequence of executing "align_warp" with arguments "-m 91 -q".

```
SELECT DISTINCT C.outputFile
FROM Provenance.% as M, M.input-of+ as C
WHERE M.executableProgram
   = "~mcgrath/AIR/AIR5.2.5/bin/align_warp"
  AND C.executableProgram
   = "~mcgrath/D2Kb/runconvert.sh"
  AND C.outputFile glob "*/atlas-y.gif"
  AND M.arguments = "-m 91 -q";
```

Query 2: Find all buddy result files in datasets with prefixes "data_12" and "data_14" that were produced using the same arguments of "align_warp".

```
SELECT W1.arguments, C1.outputFile,
       C2.outputFile
FROM Provenance.% as C1, Provenance.% as C2,
     C1.input+ as W1, C2.input+ as W2
WHERE C1.executableProgram
   = "~mcgrath/D2Kb/runconvert.sh"
  AND C2.executableProgram
   = "~mcgrath/D2Kb/runconvert.sh"
  AND W1.executableProgram
   = "~mcgrath/AIR/AIR5.2.5/bin/align_warp"
  AND W2.executableProgram
   = "~mcgrath/AIR/AIR5.2.5/bin/align_warp"
  AND C1.outputFile glob "data_12*"
  AND C2.outputFile glob "data_14*"
  AND W1.arguments = W2.arguments;
```

4.2.2 NetBSD Compile Dataset

The second dataset we used for testing was the first 1 GB of the provenance of a NetBSD compile, collected by the provenance group at Harvard University. Unlike the D2K dataset, this dataset is characterized by having a large number of repeated string literals and having only few attributes per provenance node without much variety.

Query 1: Find the names of all processes that had "-Wall" (i.e. enable all warnings) as one of the command-line arguments.

```
SELECT X.name
FROM Provenance.% as X
WHERE X.arg% = "-Wall";
```

Query 2: Find the names of all shared libraries within three levels of ancestry hierarchy of the processes that had "-Wall" as one of their command-line arguments.

```
SELECT I.name
FROM Provenance.% as X,
     X(.input)?(.input)?(.input)? as I
```

```
WHERE X.arg% = "-Wall"
  AND I.type = "NP_FILE"
  AND I.name glob "*.so";
```

Query 3: Get the list of names of all referenced files stored on a volume on which the provenance collection was not enabled.

```
SELECT X.name
FROM Provenance.% as X
WHERE X.type = "NP_FILE";
```

5 Results

5.1 Loading Data

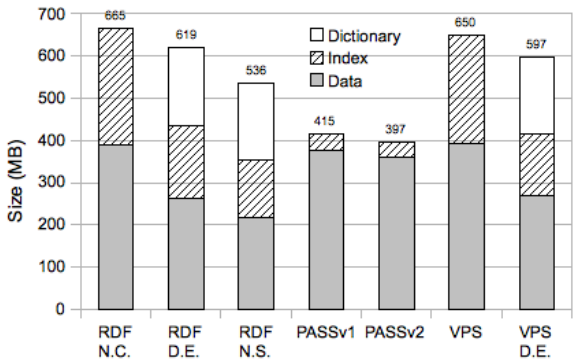

Figure 3: **D2K Space Usage**. On-disk size of each database storage model, broken down into raw data, index, and dictionary.

Figure 2 summarizes the loading times of the first 1 GB of the NetBSD dataset together with the physical sizes of the database. Figure 3 summarizes the space occupied after loading 391 MB of D2K data. The database size is broken into its three main components:

- **Data**: the part of the database that can be used to lookup all property values of provenance nodes given the (node id, property name) pairs
- **Index**: the part that can be used to find node ID numbers given values of properties
- **Dictionary**: the dictionary used for encoding all string literals, such as in the two RDF variants, or just for encoding the command-line arguments in PASSv1 (the dictionary used to encode individual command-line words in PASSv2 and VPS is too small to appear on the graph – less than 10 MB)

In the RDF and Vertical Partitioning approaches, one could expect that the size of the data and the size of the index would be proportional, because these methods provide an index for every property. However, observe that especially the index for RDF with no compression is disproportionately smaller. The reason for this is that the data component stores repeated property values multiple times – once per node, while the index stores each value only once.

The major outlier in Figure 2 is the RDF approach with no compression, which takes twice the space of the other methods. This is not surprising, since the dataset contains a large number of repeated long string literals, which are typically command-line arguments and especially dumps of environment variables. The other models have significantly smaller footprint because they all use dictionary encoding in these two cases. In the D2K dataset, the difference between the sizes of the RDF variants is not that significant, because it does not contain any environment variables, and the arguments are typically very short (less than 10 bytes).

5.1.1 RDF and Vertical Partitioning

The RDF models with dictionary encoding take about the same amount of space, although it is no surprise that the version with additional compression is more space efficient. The difference is however not significant. In our current implementation, we encode all integers as 32-bit values, and since the integers in the database range from 0 to approximately 16 million, a 24-bit integer, the average space savings per one integer are only slightly bigger than one byte. On the other hand, we expect that the database would have to use 64-bit integers in applications that require storage of terabytes of data, which would make the space savings due to null suppression more significant (see 2.2).

We performed one test load of the NetBSD data using 64-bit integer node IDs using the RDF with dictionary encoding method. When null suppression was applied, the total size shrank from 1.3 GB to 843 MB, with a linearly related reduction in load time.

Both variants of the Vertically Partitioned Store occupy approximately the same space. They both compress command-line arguments and environment variables, which suggests that these two properties could be the only ones worth compressing in order to save space. We will revisit this issue during our discussion of query performance, when we will see how it is affected by compressing all string literals.

In the D2K dataset, all arguments in the dataset are smaller than 10 bytes, so encoding them produces insignificant space savings. Consequently, the extra space occupied by extra indexes in RDF and Verti-

cally Partitioned Stores is no longer compensated for. Exactly as in the NetBSD dataset, RDF without any form of compression takes most space and the version with both dictionary encoding and null suppression performs best, and the space occupied by the VPS variants and the RDF variants is also comparable.

5.1.2 PASSv1 and PASSv2

According to the results, the second version of PASS is most space efficient. This is not surprising, since the storage method provides only few indexes and the command-line arguments and environment variables are compressed.

The first version of PASS is surprisingly space efficient, despite storing object name and type of every object multiple times in the the database. In fact, it is not that much larger than PASSv2. In the NetBSD dataset, command-line arguments and environment variables frequently contain two or even three digit number of words, so while PASSv2 compresses one to few hundred bytes (e.g. 100 words, encoded to 4 bytes each produces 400 bytes), PASSv1 encodes it using one integer. We did not yet experiment with using null suppression in PASSv2, but we expect that applying it would produce noteworthy space savings. Consequently, relative to PASSv1, the extra space occupied by PASSv1 by repeating names and types of objects is compensated by longer entries for arguments and environment variables in PASSv2. In the D2K dataset, nodes have a very small number of `INPUT` edges – more precisely, one `INPUT` edge for the majority of nodes. PASSv1 therefore rarely stores redundant values.

In the D2K dataset, both PASSv1 and PASSv2 outperform the other models. However, if we disable indexes in all storage approaches that we tested and reloaded the D2K dataset, all of them would occupy approximately the same amount of space.

5.1.3 Loading Times

We found the database loading time to be dominated by the I/O time, with CPU usage ranging between 30% and 50%. The time it takes to load the data is thus correlated with the space occupied by the database plus the extra I/O necessary for the encoding of command-line arguments and environment variables in two RDF models, both versions of PASS, and VPS (writing every string literal requires a lookup in a hash table stored on disk). Considering only the physical size of the database and ignoring any other kinds of I/O, one might expect RDF without dictionary encoding to underperform the other models by a factor of two; the load time is the reason this is not the case.

In the case of the Vertically Partitioned Store and the second version of PASS, the database loading is somewhat slower, because their method of encoding command-line arguments is more CPU intensive. Encoding a command line or a set of environment variables requires a fairly large number of dictionary lookups (on the order of hundreds in the NetBSD dataset), so this costs CPU time but no I/O. The dictionary becomes the hotspot, and because of its small size, it remains pinned in the main memory for the entire duration of the loading process. Consequently, virtually all dictionary lookups are cache hits.

5.2 Query Performance

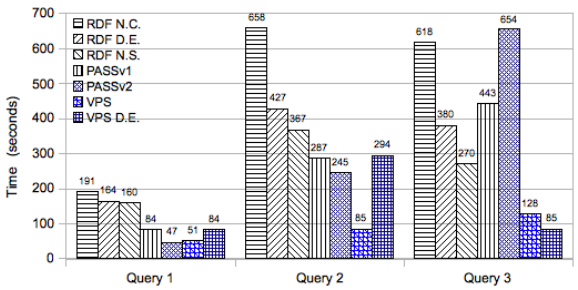

Figure 4: **NetBSD Dataset Queries**.

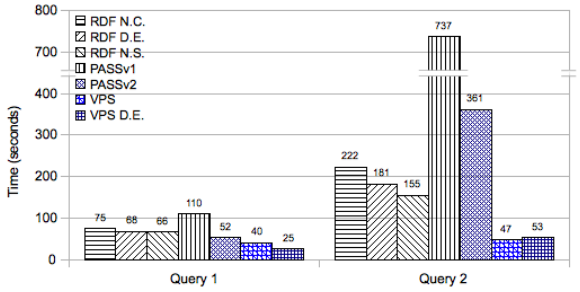

Figure 5: **D2K Dataset Queries**.

Users are very concerned with fast query response time. Figures 4 and 5 summarize the query execution times on the NetBSD and D2K datasets, respectively.

The third NetBSD query and the second D2K query show the value of indexes. Vertically Partitioned Store and RDF could use indexes to answer the queries, and thus they outperformed PASS, which do not have required indexes. The only outlier is RDF with no compression, which performed poorly

on the NetBSD query, because its B-tree that contains the index was even larger than the entire part of the PASSv1 and PASSv2 databases that were sequentially scanned.

In the next two subsections, we will discuss effects of two more interesting aspects: compression and predicate partitioning.

5.2.1 Compression

In all cases RDF without compression performs worse than when the compression is enabled, and that RDF with null suppression performs better than when only dictionary encoding is used. These RDF approaches differ only in the way they use compression, and since we found all queries to be I/O-bound, the query performance is roughly correlated to the storage size. Dictionary encoding costs extra I/O to translate strings in the query predicates to their integer IDs and vice versa, but in all cases, we found this I/O cost to be small in comparison with the I/O savings due to the use of compression.

On the other hand, the scenario is not as clear-cut in the case of Vertically Partitioned Store, which differ in whether dictionary encoding is used for all string literals or selectively only for command-line arguments and environment variables. This suggests that dictionary encoding can significantly hurt performance of some queries, but at the same time, improve performance of others.

The first NetBSD query requires a lookup of provenance nodes using an index on command-line arguments (present in all of the tested systems) and printing their names. In all cases, the query executor begins by retrieving the node IDs of all nodes with the specified argument. In RDF without compression, PASS, and the first variant of VPS, retrieving a name of an objects then costs only one database lookup per a node ID. But when a dictionary is involved, two lookups per node ID are needed to retrieve the name: one to get the string ID and the other to translate it to a string. Consequently, the methods that use dictionary encoding perform worse than those that do not.

The only outlier is RDF without compression, which is very slow because of its large footprint on disk. Since it stores all data in one B-tree, it has to scan significantly more data than all the other approaches in order to answer this query. Consequently, we would ignore this storage approach in the most of our subsequent discussion.

On the other hand, Vertically Partitioned Store with dictionary encoding performs better in the third NetBSD query. These two queries differ only in which index is used and thus also in the size of the result set. While the result set of the second query is small, the third query returns almost a tenth of the database. To translate the node ID's retrieved from the index to the node names (or their dictionary values), VPS has to scan every page of the B-tree in which it stores the NAME property. With dictionary encoding, this is an almost instantaneous scan of a 15 MB file followed by a dictionary scan. In fact, even for the first query, the B-tree is so small that the database has to read every page of the file, which is why the query times for the two queries are very similar. When dictionary encoding is not used, the first query touches only several pages in the name B-tree, while the third query has to scan more than 100 MB worth of a B-tree. Consequently, when answering the third query, the VPS variant without compression does more I/O than the first variant.

In the second NetBSD query, the node IDs are also first looked up using an index on arguments, but then the graph is traversed up to three levels deep, considering a name and a type of each encountered node. In this case, dictionary encoding seriously hurt the performance of the second variant of Vertical Partitioned Store. The difference in latency is greater for this query than for the first one. The second query requires more dictionary lookups. Furthermore, while the first query was able to batch the lookups into one dictionary scan, the second query performs multiple scans.

Dictionary encoding is however an advantage in the first D2K query. The query executor must find a "runconvert" node for every "align_warp" node returned by the argument index. On every node visited during the traversal, the executor checks the value of the "executableProgram" attribute. The desired value of this attribute (as specified in the query predicate) is first translated to its string ID. This saves the extra I/O that would result in having to translate the ID retrieved from the database on every visited node. The approaches with dictionary encoding have smaller disk footprints, so it costs them less I/O to answer the query than for the approaches that do not use compression.

Bottom Line Dictionary encoding is a must for encoding repeated long string literals, such as environment variables, and depending on the dataset and the queries, it can be beneficial to encode all string literals. This is a grey area, as in some cases, compression reduces I/O, but in others, dictionary lookups can dominate the query execution time. We found null suppression to be beneficial in all cases, which can be

seen from the fact that RDF with null suppression always outperforms RDF without null suppression.

5.2.2 Predicate Partitioning

When multiple properties of a node are stored in one B-tree, they are frequently stored on disk; if not on the same page, then at least on neighboring pages. Consequently, reading multiple properties of one node is only slightly more expensive than reading one of them. Reading another property of the same node is almost guaranteed to result in a cache hit, which costs a small amount of CPU time. The denormalized approach of PASSv1 brought this to an extreme by eliminating even this round trip to the cache. The reduction in CPU time is negligible, but comes at the cost of extra I/O, because PASSv1 not only stores multiple copies of attributes such as `NAME` or `TYPE`, but then it also has to read these multiple copies when, for example, it is retrieving a list of all values of `INPUT` for a particular node.

RDF and Vertically Partitioned Store are the two opposite extremes of the predicate partitioning continuum. One variant of RDF and one variant of VPS use both dictionary encoding and null suppression, so their query times are directly comparable. In all cases, VPS outperforms RDF. Our queries reference only a handful of properties, so the combined size of the B-trees that VPS uses to answer the query is often significantly smaller than the one or two B-trees used by RDF. On the other hand, we expect that RDF would be more suitable for queries that reference a large number of predicates, exactly as was shown by Neumann et al. [23].

Similarly, the query performance of the VPS variant without dictionary encoding is directly comparable to that of PASSv2. In all but once case, VPS outperforms PASSv2 (the only exception is the first NetBSD query, but the difference minimal). The difference between these two approaches is most remarkable in the second NetBSD query. In both cases, `NAME` and `INPUT` properties are stored in their individual B-trees, but unlike PASSv2, VPS has also a separate B-tree for `TYPE`. This is the only difference between VPS and PASSv2 with respect to this query, and yet it is alone sufficient to make VPS to run three times faster.

Bottom Line In all of our experiments, the more predicate partitioning the better. However, if the queries reference a large number predicates, partitioning can hurt performance as demonstrated by Neumann et al. [23].

6 Related Work

There is no significant past work on efficient physical storage of provenance data. Previous work has focused on logical compression of provenance graphs [8] and efficient creation of indexes [13]. Most of the active research in efficient storage considers semi-structured data, but it does not necessarily focus on provenance, which has some very distinct characteristics. There are many existing methods for storing semi-structured data, which we could not possibly explore in their entirety. Some of them are described below. We chose a few of them and evaluated them in the context of our provenance model.

Our work is complementary to that of Azar [4], who discusses several other provenance storage methods. Both his and our work are actively investigating the problem space of which data model is best suited for efficient storage (low space usage) and querying (fast response) of provenance graphs.

The first version of PASS [22] stores ancestor-descendant edges of the provenance graph as records in a BerkeleyDB database. Each record of an edge contains all properties of its descendant. The vertices of the graph are not stored separately, and those that do not have any ancestors are thus stored as edges with parent NULL. The second version of PASS [21] stores vertices and edges separately. The names of nodes are stored in a separate B-tree, while other properties are stored in a large property table.

Natix [10] and Timber [16] are methods for natively storing XML documents. They cannot directly represent an arbitrary DAG, but the lessons learned may be applicable to provenance storage (as demonstrated in a different domain by Heinis and Alonso [13]). When inserting an XML document to Timber, the system first creates a parse tree of the document. The system then traverses the tree in the depth-first manner, storing the XML tags in this depth-first order. This storage method is especially effective for workloads that frequently request sub-elements of an XML element. Natix semantically splits the XML parse tree into subtrees, each of which fits on one page. It also uses dictionary encoding to replace non-leaf nodes in the trees, such as XML tags or attribute names, by small integer values.

Lore [19] is a database that natively stores general semi-structured data. It uses a simple Object Exchange Model [24], in which the data can be thought of as a labeled directed graph. Since Lore's query execution is heavily based on a Scan operator that performs depth-first search, it stores the data as if the graph were traversed in the depth-first manner. It

also uses indexes to quickly navigate the graph along the edges.

The majority of existing RDF data storage solutions use relational databases, such as Jena [25], Oracle [9], Sesame [7], and 3store [12], just to name a few. We focused most of our study on native, non-RDBMS representations, so we did not consider these as a part of our exploration.

RDF-3X [23], a native RDF database, compresses the RDF triples using dictionary encoding and delta encoding, and then stores them as leaf nodes in multiple B-trees. The main database consists of six B-trees, one for each permutation of (subject, predicate, object). Additionally, there are several pre-aggregated B-trees, which store triples of the form (subject, predicate, number of unique objects). The authors were able to use this approach to efficiently answer queries that involve a large number of RDF predicates.

Abadi et al. explored two alternate methods of storing RDF data that were inspired by column-oriented databases [2]. In the data is vertically partitioned, so that there is one database table and one index for every RDF predicate. In one approach, the table is stored as a B-tree compressed using dictionary encoding. In the other approach, the tables are stored in a column-store database.

7 Conclusion

We have delimited a small area of a very large space of possible data models and database physical layouts. In exploring the handful of test cases above, we extracted the underlying features and performance trends for these approaches. We assert that some of these principles are applicable to future studies of efficient provenance storage, among them dictionary compression of keys and values, using indexing available in a given database system, and predicate partitioning. The users will have to choose a model that best fits the size and growth trend of their provenance data. There is a trade-off between query responsiveness and on-disk storage size. Depending on a given provenance application, where user interactivity (browser interface tracking) or long-term data storage (scientific datasets) might be emphasized, the tuning must be selected properly.

Our established foothold in the provenance space provides a suggested foundation for building efficient provenance data storage systems for a wide range of conceivable applications.

8 Acknowledgements

The authors wish to thank the Fall 2008 CS265 class for their assistance in reviewing this paper, Prof. Margo Seltzer for her feedback on the research, David Holland for his design of PQL, the provenance group at Harvard for the NetBSD compilation dataset, and the NCSA for the D2K dataset.

References

[1] Abadi, D. J., Madden, S. R., and Ferreira, M. Integrating compression and execution in column-oriented database systems. In *SIGMOD* (Chicago, IL, USA, 2006), pp. 671–682.

[2] Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. Scalable semantic web data management using vertical partitioning. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 411–422.

[3] Annis, J., Zhao, Y., Voeckler, J., Wilde, M., Kent, S., and Foster, I. Applying Chimera virtual data concepts to cluster finding in the Sloan Sky Survey. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing* (Los Alamitos, CA, USA, 2002), IEEE Computer Society Press, pp. 1–14.

[4] Azar, P. Data models for provenance: An empirical evaluation. CS265 Final Project, 2008.

[5] Bose, R., and Frew, J. Composing lineage metadata with XML for custom satellite-derived data products. In *Proceedings of the Sixteenth International Conference on Scientific and Statistical Database Management* (2004).

[6] Braun, U., Holland, D., Muniswamy-Reddy, K.-K., and Seltzer, M. Coping with cycles in provenance. Tech. rep., Harvard University, 2006.

[7] Broekstra, J., Kampman, A., and van Harmelen, F. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web* (London, UK, 2002), Springer-Verlag, pp. 54–68.

[8] Chapman, A. P., Jagadish, H. V., and Ramanan, P. Efficient provenance storage. In

SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data (New York, NY, USA, 2008), ACM, pp. 993–1006.

[9] Chong, E. I., Das, S., Eadon, G., and Srinivasan, J. An efficient sql-based rdf querying scheme. In *VLDB* (2005), ACM, pp. 1216–1227.

[10] Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele, R., and Westmann, T. Anatomy of a native XML base management system. *VLDB J. 11*, 4 (2002), 292–314.

[11] Futrelle, J., Myers, J., Auvil, L., McGrath, R. E., and Clutter, D. NCSA provenance challenge, D2K. `http://twiki.ipaw.info/pub/Challenge/NcsaD2k/d2k-provenance.txt`. National Center for Supercomputing Applications, 2006.

[12] Harris, S., and Gibbins, N. 3store: Efficient bulk rdf storage. In *PSSS* (2003), R. Volz, S. Decker, and I. F. Cruz, Eds., vol. 89 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[13] Heinis, T., and Alonso, G. Efficient lineage tracking for scientific workflows. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 1007–1018.

[14] Holland, D. A. PQL language guide and reference. `http://www.eecs.harvard.edu/syrah/pql/docs/`. Harvard University, 2009.

[15] Holland, D. A., Braun, U., Maclean, D., Muniswamy-Reddy, K.-K., and Seltzer, M. Choosing a data model and query language for provenance. In *Proceedings of the 2nd International Provenance and Annotation Workshop* (Salt Lake City, UT, USA, 2008).

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

[17] Jayapandian, M., Chapman, A., Tarcea, V. G., Yu, C., Elkiss, A., Ianni, A., Liu, B., Nandi, A., Santos, C., Andrews, P., Athey, B., States, D. J., and Jagadish, H. V. Michigan molecular interactions (mimi): putting the jigsaw puzzle together. *Nucleic Acids Research 35*, Database-Issue (2007), 566–571.

[18] Margo, D. W., and Seltzer, M. The case for browser provenance. Submitted to TaPP 2009, 2008.

[19] McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J. Lore: A database management system for semistructured data. *SIGMOD Record 26* (1997), 54–66.

[20] Moreau, L., et al. The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*. Published online. DOI 10.1002/cpe.1233, April 2008.

[21] Muniswamy-Reddy, K.-K., Barillari, J., Braun, U., Holland, D. A., Maclean, D., Seltzer, M., and Holland, S. D. Layering in provenance-aware storage systems. Tech. Rep. Computer Science Technical Report TR-04-08, Harvard University, 2008.

[22] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. Provenance-aware storage systems. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference* (Berkeley, CA, USA, 2006), USENIX Association, pp. 4–4.

[23] Neumann, T., and Weikum, G. RDF-3X: a RISC-style engine for RDF. In *International Conference on Very Large Data Bases (VLDB'08)* (Auckland, New Zealand, August 2008).

[24] Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. Object exchange across heterogeneous information sources. In *In Proceedings of the Eleventh International Conference on Data Engineering* (1995), pp. 251–260.

[25] Wilkinson, K., Sayers, C., Kuno, H., and Reynolds, D. Efficient RDF storage and retrieval in Jena2. In *Proc. First International Workshop on Semantic Web and Databases* (2003).