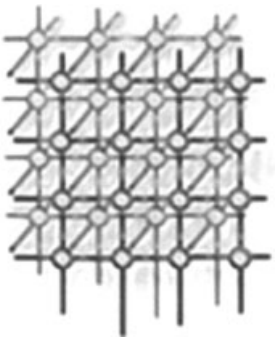
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **20**:531–540
Published online 11 September 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1227



PASSing the provenance challenge

David A. Holland*,†, Margo I. Seltzer, Uri Braun and Kiran-Kumar Muniswamy-Reddy

Harvard University, Cambridge, MA, U.S.A.

SUMMARY

Provenance-aware storage systems (PASS) are a new class of storage system treating provenance as a first-class object, providing automatic collection, storage, and management of provenance as well as query capabilities. We developed the first PASS prototype between 2005 and 2006, targeting scientific end users. Prior to undertaking the provenance challenge, we had focused on provenance collection and storage, without much emphasis on a query model or language. The challenge forced us to (quickly) develop a query model and infrastructure implementing this model. We present a brief overview of the PASS prototype and a discussion of the evolution of the query model that we developed for the challenge. Copyright © 2007 John Wiley & Sons, Ltd.

Received 29 November 2006; Revised 23 April 2007; Accepted 1 May 2007

KEY WORDS: provenance; automatic collection; file system

INTRODUCTION

A provenance-aware storage system, or PASS, is a storage system in which provenance is handled seamlessly and transparently to the user. This handling includes *collection*, *indexing*, *storage*, and *retrieval*. Handling provenance directly in the storage system offers certain advantages beyond transparency, most notably consistency: the provenance cannot be lost or separated from the data it describes. It also allows *automatic collection* [1].

Our first prototype PASS [2], developed in 2005, was intended as a vehicle to allow us to explore automatic provenance collection and the uses of provenance at the system level. This prototype consists of a Linux 2.4.29 kernel modified for automatic collection, a kernel-level port of Berkeley DB [3,4] for indexing, and a stackable [5] file system layer called PASTA. It also includes a command-line query tool and a graphical provenance browser.

*Correspondence to: David A. Holland, Maxwell-Dworkin Bldg., 33 Oxford St., Cambridge, MA 02138, U.S.A.
†E-mail: dholland@eecs.harvard.edu

This prototype was intended to demonstrate automatic provenance collection and storage of provenance within the file system. Until we tackled the provenance challenge, our query model was 'make the data available via Berkeley DB'. We had some tools for inspecting the database, but we had not developed a proper query model or paid much attention to querying. The provenance challenge forced us to choose a query model and write a query tool; however, both the model and the tool are rather *ad hoc* and should not be considered definitive.

In the remainder of this paper, we first discuss the PASS approach in terms of the categories presented in the introduction to this issue [6]; we then outline the query system we built and discuss its limitations. Following this, we discuss what we learned from the challenge in the context of the challenge queries themselves, and then we conclude.

THE PASS APPROACH

System characteristics

Execution environment: The PASS execution environment is the ordinary Unix execution environment, and users can take advantage of the full power of Unix for executing workloads in whatever form they choose. For the challenge, we simply ran the workload using the sample shell script provided as part of the challenge description.

Representation technology: We included an in-kernel port of Berkeley DB [3] and used a set of five Berkeley DB databases (tables in relational parlance) to store all provenance data and indexes. We chose Berkeley DB as our back-end storage because it was an off-the-shelf solution for indexed lookup, and because it was feasible to integrate with the operating system kernel. Berkeley DB is an embedded database library that provides efficient keyed lookup on key/value pairs. Since Berkeley DB is embedded, data are accessed via a simple programmatic put/get interface. There is no schema or query model.

Query language: Our query language is an *ad hoc* custom tool, described more fully later in the paper. This was a pragmatic trade-off between functionality and development time.

Research emphasis: Our research is about recording (R) and storage (S). We execute unmodified user workloads using unmodified user applications. We accomplish this by modifying the operating system kernel, so that it records and stores provenance.

Implementation: We ran the challenge shell script. While we did use a fake version of one component (`slicer`), this was because of potential licensing concerns and not for any technical reason. The fake `slicer` was written such that its operation (though not its identity) was indistinguishable from the real `slicer` from the point of view of the PASS kernel.

Data representation

We represent provenance as name/value attribute pairs, where names are things like INPUT and ENVIRONMENT and values are things like `/bin/sh` and `EDITOR=emacs`. We collect provenance for various objects: files on PASS volumes, processes, pipes, and files from non-PASS volumes; however, we persist provenance only for files on PASS volumes. For a single PASS file, we may track multiple versions; in most cases, we create a new version for a file when an

DOI: 10.1002/cpe

Table I. Database schema.

Database	Key	Values
PROVENANCE	Pnode number, record type	Subobject number, provenance data
ARGDATA	Sequence number	Command line text
ARGREVERSE	Command line text	Sequence number
ARGINDEX	Words from `argv`	Pnode number

application opens, writes, and then closes a file. (In reality, the precise details of version creation are much more subtle [2].) We maintain provenance for each version of a file and chain those versions together, creating the entire history of the file.

Conceptually, we refer to the provenance for a particular version of a file as a *pnode* and use a *pnode number* to uniquely identify each version. The database stores pnodes as collections of name/value pairs. As mentioned above, we collect provenance for transient objects, such as processes, but we do not persist that provenance. However, the provenance of a process, for example, is also part of the provenance for any file written by that process. Therefore, we must incorporate the process' provenance in the provenance for those files. In the discussion that follows, we use a process as an example of a transient, provenanced object whose provenance appears in PASS files, but the same approach is used for pipes and other transient objects as well. Let us say that process P create three files: A, B, and C. We copy the name/value pairs that comprise the provenance of P into the provenance of each of A, B, and C and identify that set of name/value pairs with a *subobject number* (a small integer), unique to the object containing these attributes. An unfortunate side effect of this technique is that these transient entities are not *uniquely* named; process P in our example appears as three different subobjects in the provenance of A, B, and C. The assorted shortcomings of this representation will be discussed later in the paper.

The provenance data are physically stored in four Berkeley DB databases (tables in relational parlance), listed in Table I. (A fifth is used to maintain the linkage between provenance and data.)

The ARGDATA and ARGREVERSE databases allow sharing repeated argument and environment vectors, as a space-saving measure; the ARGINDEX database is meant to allow fast searches for argument substrings.

Representation characteristics

Includes workflow representation: Our system does not make use of any sort of explicit workflow representation. It allows users to run workloads using whatever Unix tools they choose; some of these (e.g. `make`) involve canonical workflow representations that can be found using the provenance afterwards, but others do not.

Data derivation vs causal flow of events: Our system is based on data derivation. It observes events (`writes` to files, `exec` operations, and so forth), converting them to either to name/value pairs that indicate object identity information or data flow information. We remove duplicate name/value pairs in a single pnode, because they contribute only redundant information. We also remove events that do not contribute to data flow, as retaining every kernel-observable event would consume an excessive amount of disk space, even by today's standards.

DOI: 10.1002/cpe

Annotations: For PASS, annotations are both in scope and implemented. The distinction between annotations (information from outside the provenance system) and provenance (information from within the provenance system) is a false dichotomy, because various entities may 'participate' in 'the provenance system' to different degrees or in different ways depending on circumstances. Our system is intended for an environment where the user, the applications, any support libraries, and the operating system all willingly participate and provide the best information reasonably possible within the limits of human error.

Time: Our system records a timestamp for each version of each file. These timestamps can be used for query purposes. Time information is not used in the provenance collection, so there is no inherent need for the timestamps to be unique or even accurate.

Naming: Internally, our system names objects with a pnode number and a subobject number, as described above. Since it exists at the operating system level, other names exist in addition: file names, inode numbers, and process ids. All of these can be used in queries.

Granularity: We maintain provenance on files. We also track processes; we consider the virtual memory space of a process to be equivalent to a file.

Abstraction mechanisms: We currently have no support for provenance abstraction. We are investigating abstraction mechanisms in connection with our second-generation system.

Comparison with other approaches

PASS is quite different from most of the other challenge systems, such as job provenance (JP) [7], the virtual data grid (VDL) [8], and Taverna [9], which embed provenance collection in workflow engines. It is most similar to ES3 [10], which uses passive tracing (e.g. `strace`) to collect provenance from unmodified applications. We collect provenance at a finer level of granularity, which might be considered a burden to application users, but enables a level of system debugging that is difficult to achieve with higher level systems. The main advantage that systems like ES3 and PASS offer is that they capture provenance for all executions, not just those performed in the context of the workflow engine. Therefore, local experimentation and incidental computation are fully documented; users need not distinguish between *real* experiments and *toy* ones. Whenever something important or interesting happens, the system contains the provenance of that interesting event. The greatest disadvantage of these approaches is that they may capture provenance that is of no interest to the user. Both systems identify this problem and have filtering techniques to address it. Ultimately, we believe that complete solutions will employ a combination of system-level monitoring and workflow-based systems.

QUERY CAPABILITIES

As a storage systems group, we took a storage-centric perspective when we designed the first PASS prototype. Our initial approach to querying was that we provided the data in a simple format via Berkeley DB; application writers or HCI researchers or other third parties would be able to build more complex structures on top of what we provided.

We had observed that ancestry and descendent queries would be important, and we knew that querying by argument strings was likely to be useful. Consequently, the system was built with

DOI: 10.1002/cpe

the ability to handle ancestry queries and with a secondary index for argument strings. Indexing for descendent query support was designed but never implemented. Consistent with our research mission, we spent far more time on provenance storage and collection than on query. While we had some simple tools for performing what we believed were typical queries, we had no query model or useful query processor, nor did we have a complete set of representative sample queries.

We developed a simple query tool, called `nq` (new query), to answer the challenge queries. It and its query model represent a trade-off between generality and implementation time; the final version described here should be interpreted as what we were able to implement with minimal effort.

The `nq` query model treats the provenance as a single relational table, where each provenanced entity appears as a row and each provenance attribute appears as a column. Each `nq` query consists of first a recursive search, which collects a subset of the table based on ancestry relationships, followed by flat filtering, sorting, and projection operations. It also allows a choice of several reporting formats for its output. It does not support joins.

The syntax is SQL-like and is summarized in Figure 1. Not every query that can be written is actually implemented; in particular, `chain` queries (find the route connecting two objects) were never implemented and neither were the `all`/`any`/`no` expression quantifiers.

Additional explanatory notes:

- The `sunrise`, `sunset`, `depth`, and `count` filters constrain the recursive search. `nq` lacks the query planner that would allow inferring these from filter expressions.
- An `anchor` is an object at which the recursive search stops, similar to the `-prune` operation in the Unix `find` utility. Any files matching the expression are treated as anchors.
- The `concat` operator applies to vector data-like argument strings. This allows, for example, writing searches for patterns that span words in argument lists.
- The `nameof` and `typeof` operators apply only to cross-reference attributes (those that point to other objects), and fetch the object name and object type (file, process, pipe, etc.), respectively, from the object pointed to.

$query$ ::= $select_{opt}$ $search$ $filter$ $sort_{opt}$ $report$
$select$ ::= `select` $field$...|`*` `from`
$sort$ ::= `order by` $field$ [`ascending`|`descending`]
$report$ ::= `dump` | `report text` | `report html` | `table` | `graph` | `script` | `makefile`
$search$ ::= `ancestors` $objectlist$ | `descendents` $objectlist$ | `chain` $object$ $object$ | `everything`
$filter$ ::= `sunrise` $earliest\_time$ | `sunset` $latest\_time$ | `depth` $depth\_limit$ | `count` $count\_limit$
| `anchor` $expr$ | `hide` $object\_type$ | `where` $expr$
$expr$ ::= `existing` | `nonexisting` | $expr$ op $expr$ | `all`|`any`|`no` $expr$ | `(` $expr$ `)`
| `concat(`$field$`)` | `nameof(`$field$`)` | `typeof(`$field$`)` | $field$
$field$ ::= $attribute$ | `$`$user\_annotation$
op ::= `&&` | `||` | `==` | `!=` | `<` | `>` | `<=` | `>=` | `~` | `!~`

Figure 1. `nq` syntax summary.

DOI: 10.1002/cpe

- The operators are as one would expect from C or `awk`; however, the tilde denotes the simpler Unix shell 'glob' matching rather than full regular expression matching.

Note that there is no support for multiple recursive searches; one cannot directly take the result from one query and use it as the starting point for another recursive search. However, as `nq` is a command-line tool, the Unix shell can be used to invoke it repeatedly as needed. This technique was used for several of the challenge queries.

THE CHALLENGE

Query 1: `nq 'ancestors atlas-x.gif report'`

Query 1 asks for the complete provenance of one of the workload outputs. This is a simple query that we could handle with our original tools. Unfortunately, our output is extremely large: somewhat over five megabytes of text describing nearly 5000 objects.

PASS records and reports system-level provenance, so the query returns the complete system-level provenance for the file in question, not just the material that relates to the workflow itself. For example, this includes the compilation of the AIR tool suite used in the workload.

On one hand, the query asked for this level of detail; in some cases, compilation specifics (e.g., compile-time options) directly affect the output of that tool. On the other hand, frequently, this level of detail is not what the user wanted. Furthermore, even for this relatively short and simple workload, the full provenance dump is too large to be useful.

We have in the past considered the possibility that it may be desirable to be able to both summarize and abstract the material reported so that the user can reasonably inspect and comprehend it. We have learned from the challenge that such functionality is not only desirable but urgently necessary.

Note that summarizing the material by ignoring everything not part of a workflow specification (as many/most workflow-based systems do implicitly) loses information that should not be ignored. For example, the exact identities of the shared libraries used by the programs in the workload are important for reproducibility; the shared library mechanisms in most OSes are fragile, and mysterious application problems can often be traced to changes in shared libraries.

The advantage of the automatic collection method over methods that require workload specification is that these otherwise hidden dependencies are revealed. The disadvantage is that if one is not interested in them, one still has to wade through them or filter them out.

It is important to note, however, that the 5000 objects in query output have already been filtered and pruned. As mentioned above, objects that are not PASS files do not necessarily have unique names; this is because they can appear more than once in the database. Specifically, they appear once in the provenance of every file to which they contribute directly. Code in `nq` finds and eliminates these duplicates. There is also code that hides versions (primarily versions of processes) whose existence in the report contributes no information. These two steps remove 66 026 and 2308 objects, respectively, and thus represent a considerable increment in output comprehensibility over the very first version of `nq`.

DOI: 10.1002/cpe

Query 2

```
nq 'ancestors atlas-x.gif
    anchor (type == "proc" && name == "AIR5.2.5/bin/softmean") report'
```

Query 2 asks for a subset of that requested in query 1. This is exactly the functionality that `anchor` provides, as described above. The anchor does not, however, remove the full ancestry of the tools used in the stages not pruned away, so the output is still large.

The query matches any process that was run under the name `AIR5.2.5/bin/softmean`, whether or not it was the same version of the softmean program, or even the same program at all, and also whether or not the process was related to the workloads in which we were interested. These uncertainties can be clarified if necessary by expanding the expression or by searching for the anchor points explicitly beforehand.

Query 3

This query asks for a report on certain stages of the workload. Because our system does not have explicit specification of workloads, there is also no explicit specification of stages. So there is no way to name stages as such; instead one can search by bounding the query, which becomes the same as query 2.

One could write three queries, one bounding the workload for each of the stages requested, and then take the union of the results; however, while support for set operations on search results would not be difficult to add, it does not currently exist.

Query 4

```
nq 'everything where basename == "align_warp" && concat(argv) ~ "*-m12*"
    && freezetime ~ "*Mon*" report'
```

Query 4 is a search by attribute, so we need no recursive search; the query is entirely a filter expression. The `freezetime` attribute says when a particular version of an object was 'finished'.

Note that `freezetime` is an attribute of a specific version of an object, whereas many of the other attributes in the system are, properly speaking, attributes of the object itself. For example, a process has only one `argv` vector, no matter how many distinct versions of it are considered to exist over the time that it executes. Our schema failed to take this into account; all attributes are stored on a per-version basis and the per-object attributes are scattered into whatever versions happened to be current when they were recognized and stored. This is often, but not always, the first version. Consequently, `nq` must inspect all versions of an object to know definitively if an attribute is present or to find its value.

This problem is exacerbated because our system does not store the types of objects; the type must be inferred using the presence or value of other attributes. Several other analyses are required within `nq` to cover for assorted shortcomings in data collection.

Inspecting all versions is in turn problematic because there is no index for the versions of a particular object; they can only be collected by recursively following the 'previous-version' records. Such a recursive search is required for essentially every object that every query touches. This proved most unfortunate.

DOI: 10.1002/cpe

Query 5

```
ALIGN_WARPS=`nq 'select ident from everything
     where type == "proc" && basename == "align_warp" table'`
nq 'select name from ancestors { '"\$ALIGN_WARPS"' } depth 1
  where basename ~ "*.hdr" table'
 :
nq 'descendents {
    anatomy1.hdr anatomy2.hdr anatomy3.hdr anatomy4.hdr reference.hdr
  }
  where basename ~ "atlas*.gif" || basename ~ "atlas*.jpg" report'
```

This query is a search on application annotations in the input files. Our first PASS prototype was never intended to be able to interoperate directly with application-specific provenance systems, but it does allow user-level annotations. Such annotations are stored directly in the PASS database.

The first half of the query is two consecutive invocations of `nq`. We first need to find the `align_warp` processes that might have read the header files. Then, we need to find the header files they did read. The use of the 'ident' attribute (which gives the internal unique identifier for an object) and the 'table' output format allows using the shell to chain the queries together in a reasonably natural way.

The use of the 'depth 1' restriction limits the output to immediate inputs of the `align_warp` processes; otherwise, the recursive search would find any such file anywhere in the ancestry, which might or might not be one we wanted.

The second half of the query is a descendent query; this is logically the same as an ancestry query except that it searches the graph in the opposite direction. Internally it is quite different, because our database is organized in terms of ancestry records that point only from an entity to the entities on which it depends. We planned but never implemented indexing for descendent queries; consequently, they must scan the whole database and are rather slow.

Query 6

```
ALIGN_WARPS=`nq 'select ident from everything where type == "proc" &&
     basename == "align_warp" && concat(argv) ~ "*-m 12*" table'`
SOFTMEANS=`nq 'select ident from descendents { '"\$ALIGN_WARPS"' }
      where type == "proc" && basename == "softmean" table'`
nq 'select name from descendents { '"\$SOFTMEANS"' } depth 1
  where type == "passfile" && basename ~ "*.img" report'
```

This query, which has two ancestry predicates in the statement, requires three steps. The first step finds the starting points; the second and third handle the two ancestry predicates.

Note that there are two ways to handle the first two steps: one can start from the `align_warp` processes and find the descendent `softmean` processes, as above; or one can find the `softmean` processes and find the ancestor `align_warp` processes.

DOI: 10.1002/cpe

Query 7

For this query, we run a slightly modified form of the workload using the netpbm toolkit instead of ImageMagick for the final image conversion; the goal is to report the differences in the provenance of the output images.

We considered implementing explicit diff queries, but did not do so in the time allotted. Instead, we collected two complete reports (the same form as query 1) and ran `diff` on them.

This produced output that was considerably larger than one would desire: about 11 000 lines of diff. This could be worse; for perspective, the reports are about 110 000 lines each. The diff includes not just the altered workload, but also the `emacs` process used to alter the workload script and various system-level differences arising from the use of different tools. It also includes noise.

The diff output shows that most of the textual changes are the result of the two reports using different pnode and subobject numbers for the same non-PASS files. This arises because the duplicate-elimination code cannot always choose the same pnode and subobject number when examining different sets of files.

Query 8

```
INPUTS=`nq 'select ident from everything where \$center == "UChicago"
table'`
WARPS=`nq 'select ident from descendents { '"\$INPUTS"' } depth 1
    where type == "proc" && basename == "align_warp" table'`
nq 'descendents { '"\$WARPS"' } anchor type == "passfile"
  where type == "passfile" report'
```

Query 8 is a query on explicit user annotations used as the roots of a descendents search. After running the workload we added the pertinent annotation to two of the input files and not the others.

Note that we added the annotation after the workload on purpose; the file is still the same version, and we want to be able to annotate files after the fact and still search on the annotations. (This is useful, for example, for marking files bad.)

The `$` allows referring to a user annotation. This prevents name conflicts between user annotations and built-in fields.

We begin by finding the annotated file, then finding the immediate (depth 1) descendent processes of a suitable form; then we find the output files from those processes—the anchor stops the search at the first file down, and the type restriction prevents anything in between from appearing. This form allows for possible pipes and output filter processes in between.

Query 9

```
nq 'select annotations from everything
  where (basename ~ "atlas*.gif" || basename ~ "atlas*.jpg") && (
     \$studyModality == "speech" || \$studyModality == "visual" ||
     \$studyModality == "audio") report'
```


DOI: 10.1002/cpe

This query is purely a query on annotations. To make it possible we annotated all six output images (from both the regular and Q7-variant workloads), one set with study modality 'mindreading' and the other with study modality 'visual'.

CONCLUSION

The challenge reminded us that we had not thought hard enough about the details of the queries that users might want to ask. As a result, we had three major problems: first, we had no query model and had to develop an *ad hoc* one in a hurry; second, our database schema and data representation had not been designed with the query model in mind, and came up short; and finally, we do not do as good a job as we should of filtering the query output.

There are two broader lessons to be learned from this. First, it makes no sense to optimize for query performance without a clear understanding of the expected queries. Second, design mistakes often do not appear until one has completed the whole implementation, including the parts one expected to be straightforward.

The provenance challenge was an excellent testbed for our system. We were pleased that we were able to respond to all the queries, and valued the experience we gained.

REFERENCES

1. Braun U, Garfinkel S, Holland DA, Muniswamy-Reddy K-K, Seltzer M. Issues in automatic provenance collection. *Proceedings of the 2006 International Provenance and Annotation Workshop*, Chicago, IL, May 2006.
2. Muniswamy-Reddy K-K, Holland DA, Braun U, Seltzer M. Provenance-aware storage systems. *Proceedings of the 2006 USENIX Annual Technical Conference*, Boston, MA, June 2006.
3. Olson MA, Bostic K, Seltzer MI, Berkeley DB. *USENIX Annual Technical Conference*, *FREENIX Track*, Monterey, CA, June 1999.
4. Kashyap A. File system extensibility and reliability using an in-kernel database. *Master's Thesis*, Stony Brook University, *Technical Report FSL-04-06*, December 2004.
5. Zadok E, Bădulescu I, Shender A. Extending file systems using stackable templates. *USENIX Technical Conference*, Monterey, CA, June 1999.
6. Moreau L, Ludäscher B, Altintas I, Barga RS, Bowers S, Callahan S, Chin Jr G, Clifford B, Cohen S, Cohen-Boulakia S, Davidson S, Deelman E, Digiampietri L, Foster I, Freire J, Frew J, Futrelle J, Gibson T, Gil Y, Goble C, Golbeck J, Groth P, Holland DA, Jiang S, Kim J, Koop D, Krenek A, McPhillips T, Mehta G, Miles S, Metzger D, Munroe S, Myers J, Plale B, Podhorszki B, Ratnakar V, Santos E, Scheidegger C, Schuchardt K, Seltzer M, Simmhan YL, Silva C, Slaughter P, Stephan E, Stevens R, Turi D, Vo H, Wilde M, Zhao J, Zhao Y. The First Provenance Challenge. *Concurrency and Computation*: *Practice and Experience* 2007; DOI: 10.1002/cpe.1233.
7. Krenek A, Sitera J, Matyska L, Dvorak F, Mulac M, Ruda M, Salvet Z. gLite Job Provenance—a job-centric view. *Concurrency and Computation*: *Practice and Experience* 2007; DOI: 10.1002/cpe.1252.
8. Clifford B, Foster I, Hategan M, Stef-Praun T, Wilde M, Zhao Y. Tracking provenance in a virtual data grid. *Concurrency and Computation*: *Practice and Experience* 2007; DOI: 10.1002/cpe.1256.
9. Zhao J, Goble C, Stevens R, Turi D. Mining Taverna's semantic web of provenance. *Concurrency and Computation*: *Practice and Experience* 2007; DOI: 10.1002/cpe.1231.
10. Frew J, Metzger D, Slaughter P. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience* 2007; DOI: 10.1002/cpe.1247.