provenance working notes, benc

Table of Contents

1. Goal of this present work 1
2. Running your own provenance database 1
3. swift-about-* commands 3
4. What this work does not address 4
5. Data model 6
6. Prototype Implementations 7
7. Comparison with related work that our group has done before 17
8. Questions/Discussion points 17
9. Open Provenance Model (OPM) 23
10. Processing i2u2 cosmic metadata 24
11. processing fMRI metadata 28
12. random unsorted notes 29
13. Provenance Challenge 1 examples 29
14. Representation of dataset containment and procedure execution in r2681 and how it could change. 33

`$Id$`

1. Goal of this present work

The goal of the work described in this document is to investigate *retrospective provenance* and *metadata handling* in Swift, with an emphasis on effective querying of the data, rather than on collection of the data.

The motivating examples are queries of the kinds discussed in section 4 of 'Applying the Virtual Data Provenance Model'[1]; the queries and metadata in the First Provenance Challenge[2]; and the metadata database used by i2u2 cosmic.

I am attempting to scope this so that it can be implemented in a few months; more expensive features, though desirable, are relegated the the 'What this work does not address' section. Features which appear fairly orthogonal to the main aims are also omitted.

This document is a combination of working notes and on-going status report regarding my provenance work; as such its got quite a lot of opinion in it, some of it not justified in the text.

2. Running your own provenance database

This section details running your own SQL-based provenance database on servers of your own control.

2.1. Check out the latest SVN code

Use the following command to check out the `provenancedb` module:

[1] http://www.ci.uchicago.edu/swift/papers/VirtualDataProvenance.pdf
[2] http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge

```
svn co https://svn.ci.uchicago.edu/svn/vdl2/provenancedb
```

2.2. Configuring your SQL database

Follow the instructions in one of the following sections, to configure your database either for sqlite3 or for postgres.

2.2.1. Configuring your sqlite3 SQL database

This section describes configuring the SQL scripts to use sqlite[3], which is appropriate for a single-user installation.

Install or find sqlite3. On `communicado.ci.uchicago.edu`, it is installed and can be accessed by adding the line `+sqlite3` to your ~/.soft file and typing `resoft`. Alternatively, on OS X with MacPorts, this command works:

```
$ sudo port install sqlite3
```

Similar commands using `apt` or `yum` will probably work under Linux.

In the next section, you will create a `provenance.config` file. In that, you should configure the use of sqlite3 by specifying:

```
export SQLCMD="sqlite3 provdb "
```

(note the trailing space before the closing quote)

2.2.2. Configuring your own postgres 8.3 SQL database

This section describes configuring a postgres 8.3 database, which is appropriate for a large installation (where large means lots of log files or multiple users)

First install and start postgres as appropriate for your platform (using **apt-get** or **port** for example).

As user `postgres`, create a database:

```
$ /opt/local/lib/postgresql83/bin/createdb provtest1
```

Check that you can connect and see the empty database:

```
$ psql83 -d provtest1 -U postgres
Welcome to psql83 8.3.6, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help with psql commands
       \g or terminate with semicolon to execute query
       \q to quit

provtest1=# \dt
No relations found.
provtest1=# \q
```

[3] http://www.sqlite.org/

In the next section, when configuring `provenance.config`, specify the use of postgres like this:

```
export SQLCMD="psql83 -d provtest1 -U postgres "
```

Note the trailing space before the final quote. Also, note that if you fiddled the above test command line to make it work, you will have to make similar fiddles in the `SQLCMD` configuration line.

2.3. Import your logs

Now create a `etc/provenance.config` file to define local configuration. An example that I use on my laptop is present in `provenance.config.soju`. The `SQLCMD` indicates which command to run for SQL access. This is used by other scripts to access the database. The `LOGREPO` and `IDIR` variables should point to the directory under which you collect your Swift logs.

Now import your logs for the first time like this:

```
$ ./swift-prov-import-all-logs rebuild
```

2.4. Querying the newly generated database

You can use **swift-about-*** commands, described in the commands section.

If you're using the SQLite database, you can get an interactive SQL session to query your new provenance database like this:

```
$ sqlite3 provdb
SQLite version 3.6.11
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

3. swift-about-* commands

There are several swift-about- commands:

swift-about-filename - returns the global dataset IDs for the specified filename. Several runs may have output the same filename; the provenance database cannot tell which run (if any) any file with that name that exists now came from.

Example: this looks for information about `001-echo.out` which is the output of the first test in the language-behaviour test suite:

```
$ ./swift-about-filename 001-echo.out
Dataset IDs for files that have name file://localhost/001-echo.out
 tag:benc@ci.uchicago.edu,2008:swift:dataset:20080114-1353-g1y3moc0:720000000001
 tag:benc@ci.uchicago.edu,2008:swift:dataset:20080107-1440-67vursv4:720000000001
 tag:benc@ci.uchicago.edu,2008:swift:dataset:20080107-2146-ja2r2z5f:720000000001
 tag:benc@ci.uchicago.edu,2008:swift:dataset:20080107-1608-itdd69l6:720000000001
 tag:benc@ci.uchicago.edu,2008:swift:dataset:20080303-1011-krz4g2y0:720000000001
 tag:benc@ci.uchicago.edu,2008:swift:dataset:20080303-1100-4in9a325:720000000001
```

Six different datasets in the provenance database have had that filename (because six language behaviour test runs

have been uploaded to the database).

swift-about-dataset - returns information about a dataset, given that dataset's uri. Returned information includes the IDs of a containing dataset, datasets contained within this dataset, and IDs for executions that used this dataset as input or output.

Example:

```
$ ./swift-about-dataset tag:benc@ci.uchicago.edu,2008:swift:dataset:20080114-1353-g1y3moc0
About dataset tag:benc@ci.uchicago.edu,2008:swift:dataset:20080114-1353-g1y3moc0:720000000
That dataset has these filename(s):
 file://localhost/001-echo.out

That dataset is part of these datasets:

That dataset contains these datasets:

That dataset was input to the following executions (as the specified named parameter):

That dataset was output from the following executions (as the specified return parameter):
 tag:benc@ci.uchicago.edu,2008:swiftlogs:execute:001-echo-20080114-1353-n7puv429:0
```

This shows that this dataset is not part of a more complicated dataset structure, and was produced as an output parameter t from an execution.

swift-about-execution - gives information about an execution, given an execution ID

```
$ ./swift-about-execution tag:benc@ci.uchicago.edu,2008:swiftlogs:execute:001-echo-2008011
About execution tag:benc@ci.uchicago.edu,2008:swiftlogs:execute:001-echo-20080114-1353-n7p
                                                                id
-------------------------------------------------------------------------------------------
 tag:benc@ci.uchicago.edu,2008:swiftlogs:execute:001-echo-20080114-1353-n7puv429:0
(1 row)
```

This shows some basic information about the execution - the start time, the duration, the name of the application, the final status.

4. What this work does not address

This work explicitly excludes a number of uses which traditionally have been associated with the VDS1 Virtual Data Catalog - either as real or as imagined functionality.

Much of this is open to debate; especially regarding which features are the most important to implement after the first round of implementation has occurred.

Namespaces and versioning	the need for these is somewhat orthogonal to the work here. Namespaces and versions provide a richer identifier but don't fundamentally change the nature of the identifier. so for now I mostly ignore as they are (I think) fairly straightforward drudge-work to implement, rather than being fundamentally part of how queries are formed. Global namespaces are used a little bit for identifying datasets between runs (see tag URI section)
Prospective provenance	SwiftScript source programs don't have as close a similarity to their retrospective structure as in VDL1, so a bunch of thought required here. Is this required? Is it different from the SwiftScript program-library point?

A database of all logged information	though it would be interesting to see what could be done there. straightforward to import eg. .event and/or .transition files from log parsers into the DB.
Replica management	No specific replica location or management support. However see sections on general metadata handling (in as much as general metadata can support replica location as a specific metadata usecase); and also the section on global naming in the to-be-discussed section. This ties in with the Logical File Names concept somehow.
A library for SwiftScript code	need better uses for this and some indication that a more conventional version control system is not more appropriate. Also included in this exclusion is storage of type definitions. Its straightforward to store type names; but the definitions are per-execution. More usecases would be useful here to figure out what sort of query people want to make.
Live status updates of in-progress workflows	this may happen if data goes into the DB during run rather than at end (which may or may not happen). also need to deal with slightly different data - for example, execute2s that ran but failed (which is not direct lineage provenance?) so - one successful invocation has: one execute, one execute2 (the most recent), and maybe one kickstart record. it doesn't track execute2s and kickstarts for failed execution attempts (nor, perhaps, for failed workflows at all...)
Deleting or otherwise modifying provenance data	Deleting or otherwise modifying provenance data. Though deleting/modifying other metadata should be taken into account.
Security	There are several approaches here. The most native approach is to use the security model of the underlying database (which will vary depending on which database is used). This is a non-trivial area, especially to do with any richness. Trust relationships between the various parties accessing the database should be taken into account.
A new metadata or provenance query language	Designing a (useful - i.e. usable and performing) database and query language is a non-trivial exercise (on the order of years). For now, use existing query languages and their implementations. Boilerplate queries can be developed around those languages. One property of this is that there will not be a uniform query language for all prototypes. This is contrast to the VDS1 VDC which had a language which was then mapped to at least SQL and perhaps some XML query language too. An intermediate / alternative to something language-like is a much more tightly constrained set of library / template queries with a very constrained set of parameters. Related to this is the avoidance as much as possible of mixing models; so that one query language is needed for part of a query, and another query language is needed for another part of a query. An example of this in practice is the storage of XML kickstart records as blobs inside a relational database in the VDS1 VDC. SQL could be used to query the containing records, whilst an XML query language had to be used to query inner records. No benefit could be derived there from query language level joining and query optimisation; instead the join had to be implemented poorly by hand.

for provenance or metadata	The prototypes here collect their information through log stripping. This may or may not be the best way to collect the data. For example, hooks inside the code might be a better way.

5. Data model

5.1. Introduction to the data model

All of the prototypes use a basic data model that is strongly related to the structure of data in the log files; much of the naming here comes from names in the log files, which in turn often comes from source code procedure names.

The data model consists of the following data objects:

execute - an execute represents a procedure call in a SwiftScript program.

execute2 - an execute2 is an attempt to actually execute an 'execute' object.

dataset - a dataset is data used by a Swift program. this might be a file, an array, a structure, or a simple value.

workflow - a workflow is an execution of an entire SwiftScript program

5.2. execute

execute - an 'execute' is an execution of a procedure call in a SwiftScript program. Every procedure call in a SwiftScript program corresponds to either one execute (if the execution was attempted) or zero (if the workflow was abandoned before an execution was attempted). An 'execute' may encompass a number of attempts to run the appropriate procedure, possibly on differnt sites. Those attempts are contained within an execute as execute2 entities. Each execute is related to zero or more datasets - those passed as inputs and those that are produced as outputs.

5.3. execute2

execute2 - an 'execute2' is an attempt to run a program on some grid site. It consists of staging in input files, running the program, and staging out the output files. Each execute2 belongs to exactly one execute. If the database is storing only successful workflows and successful executions, then each execute will be associated with exactly one execute2. If storing data about unsuccessful workflows or executions, then each execute may have zero or more execute2s.

5.4. dataset

A dataset represents data within a SwiftScript program. A dataset can be an array, a structure, a file or a simple value. Depending on the nature of the dataset it may have some of the following attributes: a value (for example, if the dataset represents an integer); a filename (if the dataset represents a file); child datasets (if the dataset represents a structure or an array); and parent dataset (if the dataset is contained with a structure or an array).

At present, each dataset corresponds to exactly one in-memory DSHandle object in the Swift runtime environment; however this might not continue to be the case - see the discussion section on cross-dataset run identification.

Datasets may be related to executes, either as datasets taken as inputs by an execute, or as datasets produced by an execute. A dataset may be produced as an output by at most one execute. If it is not produced by any execute, it is an *input to the workflow* and has been produced through some other mechanism. Multiple datasets may have the same filename - for example, at present, each time the same file is used as an input in different workflows, a different dataset appears in the database. this might change. multiple workflows might (and commonly do) output files with the same name. at present, these are different datasets, but are likely to remain that way to some extent - if the contents of files is different then the datasets should be regarded as distinct.

5.5. workflow

workflow - a 'workflow' is an execution of an entire SwiftScript program. Each execute belongs to exactly one workflow. At present, each dataset also belongs to exactly one workflow (though the discussion section talks about how that should not necessarily be the case).

TODO: diagram of the dataset model (similar to the one in the provenance paper but probably different). design so that in the XML model, the element containment hierarchies can be easily marked in a different colour

6. Prototype Implementations

I have made a few prototype implementations to explore ways of storing and querying provenance data.

The basic approach is: build on the log-processing code, which knows how to pull out lots of information from the log files and store it in a structured text format; extend Swift to log additional information as needed; write import code which knows how to take the log-processing structured files and put them into whatever database/format is needed by the particular prototype.

If it is desirable to support more than one of these storage/query mechanisms (perhaps because they have unordered values of usability vs query expessibility) then perhaps should be core provenance output code which is somewhat agnostic to storage system (equivalent to the post-log-processing text files at the moment) and then some relatively straightforward set of importers which are doing little more than syntax change (cf. it was easy to adapt the SQL import code to make prolog code instead)

The script **import-all** will import into the basic SQL and eXist XML databases.

6.1. Relational, using SQL

There are a couple of approaches based around relational databases using SQL. The plain SQL approach allows many queries to be answered, but does provide particularly easy querying for the transitive relations (such as the 'preceeds' relation mentioned elsewhere); ameliorating this problem is point of the second model.

6.1.1. Plain SQL

In this model, the provenance model is mapped to a relational schema, stored in sqlite3 and queried with SQL.

This prototype uses sqlite3 on my laptop. The **import-all** will initialise and import into this database (and also into the XML DB).

example query - counts how many of each procedure have been called.

```
sqlite> select procedure_name, count(procedure_name) from executes, invocation_procedure_n
align|4
average|1
convert|3
slicer|3
```

needs an SQL database. sqlite is easy to get (from standard OS software repos, and from globus toolkit) so this is not as horrible as it seems. setup requirements for sqlite are minimal.

metadata: one way is to handle them as SQL relations. this allows them to be queried using SQL quite nicely, and to be indexed and joined on quite easily.

prov query 1:Find the process that led to Atlas X Graphic / everything that caused Atlas X Graphic to be as it is. This should tell us the new brain images from which the averaged atlas was generated, the warping performed etc.

6.1.1.1. Description of tables

Executions are stored in a table called 'executes'. Each execution has the fields: id - a globally unique ID for that execution; starttime - the start time of the execution attempt, in seconds since the unix epoch (this is roughly the time that swift decides to submit the task, *not* the time that a worker node started executing the task); duration - in seconds (time from start time to swift finally finishing the execution, not the actual on-worker execution time); final state (is likely to always be END_SUCCESS as the present import code ignores failed tasks, but in future may include details of failures; app - the symbolic name of the application

Details of datasets are stored in three tables: dataset_filenames, dataset_usage and dataset_containment.

dataset_filenames maps filenames (or more generally URIs) to unique dataset identifiers.

dataset_usage maps from unique dataset identifiers to the execution unique identifiers for executions that take those datasets as inputs and outputs. execute_id and dataset_id identify the execution and the procedure which are related. direction indicates whether this dataset was used as an input or an output. param_name is the name of the parameter in the SwiftScript source file.

dataset_containment indicates which datasets are contained within others, for example within a structure or array. An array or structure is a dataset with its own unique identifier; and each member of the array or structure is again a dataset with its own unique identifier. The outer_dataset_id and inner_dataset_id fields in each row indicate respectively the containing and contained dataset.

6.1.2. SQL with Pre-generated Transitive Closures

SQL does not allow expression of transitive relations. This causes a problem for some of the queries.

Work has previously been done (cite) to work on pre-generating transitive closures over relations. This is similar in concept to the pregenerated indices that SQL databases traditionally provide.

In the pre-generated transitive closure model, a transitive closure table is pregenerated (and can be incrementally maintained as data is added to the database). Queries are then made against this table instead of against the ground table.

All of the data available in the earlier SQL model is available, in addition to the additional closures generated here.

Prototype code: There is a script called `prov-sql-generate-transitive-closures.sh` to generate the close of the preceeds relation and places it in a table called `trans`:

```
$ prov-sql-generate-transitive-closures.sh
Previous: 0 Now: 869
Previous: 869 Now: 1077
Previous: 1077 Now: 1251
Previous: 1251 Now: 1430
Previous: 1430 Now: 1614
Previous: 1614 Now: 1848
Previous: 1848 Now: 2063
Previous: 2063 Now: 2235
Previous: 2235 Now: 2340
Previous: 2340 Now: 2385
Previous: 2385 Now: 2396
Previous: 2396 Now: 2398
Previous: 2398 Now: 2398
```

A note on timing - constructing the closure of 869 base relations, leading to 2398 relations in the closure takes 48s with no indices; adding an index on a column in the transitive relations table takes this time down to 1.6s. This is interesting as an example of how some decent understanding of the data structure to produce properly optimised queries and the like is very helpful in scaling up, and an argument against implementing a poor 'inner system'.

Now we can reformulate some of the queries from the SQL section making use of this table.

There's some papers around about transitive closures in SQL: 'Maintaining transitive closure of graphs in SQL'[4] and http://willets.org/sqlgraphs.html

how expensive is doing this? how cheaper queries? how more expensive is adding data? and how scales (in both time and size (eg row count)) as we put in more rows (eg. i2u2 scale?) exponential, perhaps? though the theoretical limit is going to be influenced by our usage pattern which I think for the most part will be lots of disjoint graphs (I think). we get to index the transitive closure table, which we don't get to do when making the closure at run time.

We don't have the path(s) between nodes but we could store that in the closure table too if we wanted (though multiple paths would then be more expensive as there are now more unique rows to go in the closure table)

This is a violation of normalisation which the traditional relational people would say is bad, but OLAP people would say is ok.

how much easier does it make queries? for queries to root, should be much easier (query over transitive table almost as if over base table). but queries such as 'go back n-far then stop' and the like harder to query.

keyword: 'incremental evaluation system' (to maintain transitive closure)

The difference between plain SQL and SQL-with-transitive-closures is that in SQL mode, construction occurs at query time and the query needs to specify that construction. In the transitive-close mode, construction occurs at data insertion time, with increased expense there and in size of db, but cheaper queries (I think).

sample numbers: fmri example has 50 rows in base causal relation table. 757 in table with transitive close.

If running entirely separate workflows, both those numbers will scale linearly with the number of workflows; however, if there is some crossover between subsequent workflows in terms of shared data files then the transitive graph will grow super-linearly.

6.2. XML

In this XML approach, provenance data and metadata is represented as a set of XML documents.

Each document is stored in some kind of document store. Two different document stores are used: the posix filesystem and eXist. XPath and XQuery are investigated as query languages.

semi-structuredness allows structured metadata without having to necessarily declare its schema (which I think is one of the desired properties that turns people off using plain SQL tables to reflect the metadata schema). but won't get indexing without some configuration of structure so whilst that will be nice for small DBs it may be necessary to scale up (though that in itself isn't a problem - it allows gentle start without schema declaration and to scale up, add schema declarations later on - fits in with the scripting style). semi-structured form of XML lines up very will with the desire to have semi-structured metadata. compare ease of converting other things (eg fmri showheader output) to loose XML - field relabelling without having to know what the fields actually are - to how this needs to be done in SQL.

The hierarchical structure of XML perhaps better for dataset containment because we can use // operator which is transitive down the tree for dataset containment.

XML provides a more convenient export format than SQL or the other formats in terms of an easily parseable file format. There are lots of tools around for processing XML files in various different ways (for example, treating as text-like documents; deserialising into Java in-memory objects based on an XML Schema definition), and XML is one of the most familiar structured data file formats.

[4] http://coblitz.codeen.org:3125/citeseer.ist.psu.edu/cache/papers/cs/554/http:zSzzSzsdmc.krdl.org.sgzSzkleislizSzpsZzSzdlsw-ijit97-9.pdf/dong99maintaining.pdf

Not sure what DAG representation would look like here? many (one per arc) small documents? is that a problem for the DBs? many small elements, more likely, rather than many small documents - roughly one document per workflow.

6.2.1. xml metadata

in the XML model, two different ways of putting in metadata: as descendents of the appropriate objects (eg. dataset metadata under the relevant datasets). this is most xml-like in the sense that its strongly hierarchical. as separate elements at a higher level (eg. separate documents in xml db). the two ways are compatible to the extent that some metadata can be stored one way, some the other way, although the way of querying each will be different.

way i: at time of converting provenance data into XML, insert metadata at appropriate slots (though if XML storage medium allows, it could be inserted later on).

modified **prov-to-xml.sh** to put that info in for the appropriate datasets (identified using the below descripted false-filename method

can now make queries such as 'tell me the datasets which have header metadata':

```
cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '//dataset[hea
```

way ii: need to figure out what the dataset IDs for the volumes are. At the moment, the filename field for (some) mapped dataset parents still has a filename even though that file never exists, like below. This depends on the mapper being able to invent a filename for such. Mappers aren't guaranteed to be able to do that - eg where filenames are not formed as a function of the parameters and path, but rely on eg. whats in the directory at initialisation (like the filesystem mapper).

```
<dataset identifier="10682109">
<filename>file://localhost/0001.in</filename>
<dataset identifier="12735302">
<filename>file://localhost/0001.h.in</filename>
</dataset>
<dataset identifier="7080341">
<filename>file://localhost/0001.v.in</filename>
</dataset>
```

so we can perhaps use that. The mapped filename here is providing a dataset identification (by chance, not by design) so we can take advantage of it:

```
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance//datas
10682109
```

I think metadata in XML is more flexible than metadata in relational, in terms of not having to define schema and not having to stick to schema. However, how will it stand up to the challenge of scalability? Need to get a big DB. Its ok to say that indices need to be made - I don't dispute that. What's nice is that you can operate at the low end without such. So need to get this stuff being imported into eg eXist (maybe the prototype XML processing should look like -> XML doc(s) on disk -> whatever xmldb in order to facilitate prototyping and pluggability.)

6.2.2. XPath query language

XPath queries can be run either against the posix file system store or against the eXist database. When using eXist, the opportunity exists for more optimised query processing (and indeed, the eXist query processing model appears to evaluate queries in an initially surprising and unintuitive way to get speed); compared to on the filesystem, where XML is stored in serialised form and must be parsed for each query.

xml generation:

```
./prov-to-xml.sh > /tmp/prov.xml
```

and basic querying with xpathtool (http://www.semicomplete.com/projects/xpathtool/)

```
cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '/provenance/e
```

q1:

```
cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '/provenance//
<toplevel>
  <dataset identifier="14976260">
    <filename>file://localhost/0001.jpeg</filename>
  </dataset>
</toplevel>
```

or can get the identifier like this:

```
 cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance//datase
14976260
```

can also request eg IDs for multiple, like this:

```
cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance//dataset
```

can find the threads that use this dataset like this:

```
 cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '/provenance/
<toplevel>
  <tie>
    <thread>0-4-3</thread>
    <direction>output</direction>
    <dataset>14976260</dataset>
    <param>j</param>
    <value>org.griphyn.vdl.mapping.DataNode hashCode 14976260 with no value at dataset=fin
  </tie>
```

now we can iterate as in the SQL example:

```
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/tie[th
4845856
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/tie[da
0-3-3
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/tie[th
3354850
6033476
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/tie[da
0-2
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/tie[th
4436324
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/tie[da
```

so now we've exhausted the tie relation - dataset 4436324 comes from elsewhere...

so we say this:

```
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/datase
11153746
7202698
12705705
7202698
12705705
655223
2088036
13671126
2088036
13671126
5169861
14285084
12896050
14285084
12896050
6487148
5772360
4910675
5772360
4910675
```

which gives us (non-unique) datasets contained within dataset 4436324. We can uniquify outside of the language:

```
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/datase
11153746
12705705
12896050
13671126
14285084
2088036
4910675
5169861
5772360
6487148
655223
7202698
```

and now need to find what produced all of those... iterate everything again. probably we can do it integrated with the previous query so that we don't have to iterate externally:

```
$ cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '/provenance
<?xml version="1.0"?>
<toplevel>
  <tie>
    <thread>0-1-3</thread>
    <direction>output</direction>
    <dataset>5169861</dataset>
    <param>o</param>
    <value>org.griphyn.vdl.mapping.DataNode hashCode 5169861 with no value at dataset=alig
  </tie>
  <tie>
    <thread>0-1-4</thread>
    <direction>output</direction>
    <dataset>6487148</dataset>
    <param>o</param>
```

```
    <value>org.griphyn.vdl.mapping.DataNode hashCode 6487148 with no value at dataset=alig
  </tie>
  <tie>
    <thread>0-1-2</thread>
    <direction>output</direction>
    <dataset>655223</dataset>
    <param>o</param>
    <value>org.griphyn.vdl.mapping.DataNode hashCode 655223 with no value at dataset=align
  </tie>
  <tie>
    <thread>0-1-1</thread>
    <direction>output</direction>
    <dataset>11153746</dataset>
    <param>o</param>
    <value>org.griphyn.vdl.mapping.DataNode hashCode 11153746 with no value at dataset=ali
  </tie>
</toplevel>
```

which reveals only 4 ties to procedures from those datasets - the elements of the aligned array. We can get just the thread IDs for that by adding /thread onto the end:

```
 cat /tmp/prov.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh '/provenance/tie[dat
0-1-3
0-1-4
0-1-2
0-1-1
```

so now we need to iterate over those four threads as before using same process.

so we will ask 'which datasets does this contain?' because at present, a composite dataset will ultimately be produced by its component datasets (though I think perhaps we'll end up with apps producing datasets that are composites, eg when a file is output that then maps into some structure - eg file contains (1,2) and this maps to struct { int x; int y;}. TODO move this para into section on issues-for-future-discussion.

so xpath here doesn't really seem too different in expressive ability from the SQL approach - it still needs external implementation of transitivity for some of the transitive relations (though not for dataset containment). and that's a big complicating factor for ad-hoc queries...

6.2.2.1. notes on using eXist

command line client doc[5]

run in command line shell with local embedded DB (not running inside a server, so analogous to using sqlite rather than postgres):

```
~/work/eXist/bin/client.sh -s -ouri=xmldb:exist://
```

import a file:

```
~/work/eXist/bin/client.sh -m /db/prov -p `pwd`/tmp.xml  -ouri=xmldb:exist://
```

note that the -p document path is relative to exist root directory, not to the pwd, hence the explicit pwd.

xpath query from commandline:

[5] http://exist.sourceforge.net/client.html

```
 echo '//tie' |  ~/work/eXist/bin/client.sh -ouri=xmldb:exist:// -x
```

6.2.3. XSLT

very much like when we revieved xpath, xslt and xquery for MDS data - these are the three I'll consider for the XML data model? does XSLT add anything? not sure. for now I think not so ignore, or at least comment that it does not add anything.

```
./prov-to-xml.sh > /tmp/prov.xml
xsltproc ./prov-xml-stylesheet.xslt /tmp/prov.xml
```

with no rules will generate plain text output that is not much use.

Two potential uses: i) as a formatting language for stuff coming out of some other (perhaps also XSLT, or perhaps other language) query process. and ii) as that other language doing semantic rather than presentation level querying (better names for those levels?)

6.2.4. XQuery query language

Build query results for this using probably the same database as the above XPath section, but indicating where things could be better expressed using XPath.

Using XQuery with eXists:

```
$ cat xq.xq
//tie
$ ~/work/eXist/bin/client.sh -ouri=xmldb:exist:// -F `pwd`/xq.xq
```

A more advanced query:

```
for $t in //tie
  let $dataset := //dataset[@identifier=$t/dataset]
  let $exec := //execute[thread=$t/thread]
  where $t/direction="input"
  return <r>An invocation of {$exec/trname} took input {$dataset/filename}</r>
```

6.3. RDF and SPARQL

This can probably also be extended to SPARQL-with-transitive-closures using the same methods as 1; or see OWL note below.

Pegasus/WINGS queries could be interesting to look at here - they are from the same tradition as Swift. However, the don't deal very well with transitivity.

OWL mentions transitivity as something that can be expressed in an OWL ontology but are there any query languages around that can make use of that kind of information?

See prolog section on RDF querying with prolog.

There's an RDF-in-XML format for exposing information in serialised form. Same discussion applies to this as to the discussion in XML above.

6.4. GraphGrep

```
 - download link see email
graphgrep[6]
graphgrep install notes: port install db3
some hack patches to get it to build with db3
```

Got a version of graph grep with interesting graph language apparently in it. Haven't tried it yet though.

6.5. prolog

Perhaps interesting querying ability here. Probably slow? but not really sure - SWI Prolog talks about indexing its database (and allowing the indexing to be customised) and about supporting very large databases. So this sounds hopeful.

convert database into SWI prolog. make queries based on that.

Can make library to handle things like transitive relations - should be easy to express the transitivity in various different ways (dataset containment, procedure-ordering, whatever) - far more clear there than in any other query language.

SWI Prolog has some RDF interfacing, so this is clearly a realm that is being investigated by some other people. For example:

http://www.xml.com/pub/a/2001/04/25/prologrdf/index.html [7]

prolog can be used over RDF or over any other tuples. stuff in SQL tables should map neatly too. Stuff in XML hiearchy perhaps not so easily but should still be doable.

indeed, SPARQL queries have a very prolog-like feel to them superficially.

prolog db is a program at the moment - want something that looks more like a persistent modifiable database. not sure what the prolog approach to doing that is.

so maybe prolog makes an interesting place to do future research on query language? not used by this immediate work but a direction to do query expressibility research (building on top of whatever DB is used for this round?)

q1 incremental:

```
?- dataset_filenames(Dataset,'file://localhost/0001.in').

Dataset = '10682109' ;
```

Now with lib.pl:

```
dataset_trans_preceeds(Product, Source) :-
   dataset_usage(Thread, 'O', Product, _, _),
   dataset_usage(Thread, 'I', Source, _, _).


dataset_trans_preceeds(Product, Source) :-
   dataset_usage(Thread, 'O', Product, _, _),
   dataset_usage(Thread, 'I', Inter, _, _),
```

[6] http://www.cs.nyu.edu/shasha/papers/graphgrep/
[7] http://www.xml.com/pub/a/2001/04/25/prologrdf/index.html

```
   dataset_trans_preceeds(Inter, Source).
```

then we can ask:

```
?- dataset_trans_preceeds('14976260',S).

S = '4845856' ;

S = '3354850' ;

S = '6033476' ;

S = '4436324' ;

No
```

which is all the dataset IDs up until the point that we get into array construction. This is the same iterative problem we have in the SQL section too - however, it should be solvable in the prolog case within prolog in the same way that the recursion is. so now:

```
base_dataset_trans_preceeds(Product, Source, Derivation) :-
   dataset_usage(Thread, 'O', Product, _, _),
   dataset_usage(Thread, 'I', Source, _, _),
   Derivation = f(one).

base_dataset_trans_preceeds(Product, Source, Derivation) :-
   dataset_containment(Product, Source),
   Derivation = f(two).

dataset_trans_preceeds(Product, Source, Derivation) :-
    base_dataset_trans_preceeds(Product, Source, DBase),
    Derivation = [DBase].

dataset_trans_preceeds(Product, Source, Derivation) :-
   base_dataset_trans_preceeds(Product, Inter, DA),
   dataset_trans_preceeds(Inter, Source, DB),
   Derivation = [DA|DB].
```

q4:

```
invocation_procedure_names(Thread, 'align_warp'), dataset_usage(Thread, Direction, Dataset
TODO format this multiline, perhaps remove unused bindings
```

6.6. amazon simpledb

restricted beta access... dunno if i will get any access - i have none so far, though I have applied.

From reading a bit about it, my impressions are that this will prove to be a key->value lookup mechanism with poor support for going the other way (eg. value or value pattern or predicate-on-value -> key) or for doing joins (so rather like a hash table - which then makes me say 'why not also investigate last year's buzzword of DHTs?'. I think that these additional lookup mechanisms are probably necessary for a lot of the query patterns.

For some set of queries, though, key -> value lookup is sufficient; and likely the set of queries that is appropriate to this model varies depending on how the key -> value model is laid out (i.e. what gets to be a key and what is its value? do we form a hierarchy from workflow downwards?)

6.7. graphviz

This is a very different approach that is on the boundaries of relevance.

goal: produce an annotated graph showing the procedures and the datasets, with appropriate annotation of identifiers and descriptive text (eg filenames, procedure names, executable names) that for small (eg. fmri sized workflows) its easy to get a visual view of whats going on.

don't target anything much bigger than the fmri example for this. (though there is maybe some desire to produce larger visualisations for this - perhaps as a separate piece of work. eg could combine foreach into single node, datasets into single node)

perhaps make subgraphs by the various containment relationships: datasets in same subgraph as their top level parent; app procedure invocations in the same subgraph as their compound procedure invocation.

7. Comparison with related work that our group has done before

7.1. vs VDS1 VDC

gendax - VDS1 has a tool **gendax** which provides various ways of accessing data from the command line. Eg. prov challenge question 1 very easily answered by this.

two points I don't like that should discuss here: i) the metadata schema (I claim there doesn't need to be a generic metadata schema at all - when applications decide they want to store certain metadata, they declare it in the database); and ii) the mixed-model - this is discussed a bit in the 'no general query language' section. consolidate/cross-link.

7.2. vs VDL provenance paper figure 1 schema

The significant differences are: (TODO perhaps produce a diagram for comparison. could use same diagram differently annotated to indicate trees in the XML section and also in the transitivity discussion section)

the 'annotation' model - screw that, go native

the dataset containment model, which doesn't exist in the virtual dataset model.

workflow object has a fromDV and toDV field. what are those meant to mean? In present model, there isn't any base data for a workflow at the moment - everything can be found in the descriptions of its components (such as files used, start time, etc). (see notes on compound procedure containment with model of a workflow as a compound procedure)

invocation to call to procedure chain. this chain looks different. there are executes (which look like invocations/calls) and procedure names (which do not exist as primary objects because I am not storing program code). kickstart records and execute2 records would be more like the annotations you'd get from the annotation part, with the call being more directly associated with the execute object.

8. Questions/Discussion points

8.1. metadata

discourse analysis: Perhaps the word 'metadata' should be banned in this document - it implies that there is some special property that distinguishes it sufficiently from normal data such that it must be treated differently from dif-

ferent data. I don't believe this to be the case.

script **prov-mfd-meta-to-xml** that generates (fake) metadata record in XML like this:

```
$ ./prov-mfd-meta-to-xml 123
<headermeta>
  <dataset>123</dataset>
  <bitspixel>16</bitspixel>
  <xdim>256</xdim>
  <ydim>256</ydim>
  <zdim>128</zdim>
  <xsize>1.000000e+00</xsize>
  <ysize>1.000000e+00</ysize>
  <zsize>1.250000e+00</zsize>
  <globalmaximum>4095</globalmaximum>
  <globalminimum>0</globalminimum>
</headermeta>
```

8.1.1. metadata random notes

metadata: there's a model of arbitrary metadata pairs being associated with arbitrary objects.

there's another model (that I tend to favour) in that the metadata schema is more defined than this - eg in i2u2 for any particular elab, the schema for metadata is fairly well defined.

eg in cosmic, there are strongly typed fields such as "blessed" or "detector number" that are hard-coded throughout the elab. whilst the VDS1 VDC can deal with arbitrary typing, that's not the model that i2u2/cosmic is using. need to be careful to avoid the inner-platform effect here especially - "we need a system that can do arbitrarily typed metadata pairs" is not actually a requirement in this case as the schema is known at application build time. (note that this matters for SQL a lot, not so much for plain XML data model, though if we want to specify things like 'is-transitive' properties then in any model things like that need to be better defined)

fMRI provenance challenge metadata (extracted using scanheader) looks like this:

```
$ /Users/benc/work/fmri-tutorial/AIR5.2.5/bin/scanheader ./anatomy0001.hdr
bits/pixel=16
x_dim=256
y_dim=256
z_dim=128
x_size=1.000000e+00
y_size=1.000000e+00
z_size=1.250000e+00

global maximum=4095
global minimum=0
```

8.2. The 'preceeds' relation

8.2.1. Provenance of hierarchical datasets

One of the main provenance queries is whether some entity (a data file or a procedure) was influenced by some other entity.

In VDS1 a workflow is represented by a bipartite DAG where one vertex partition is files and the other is procedures.

The more complex data structures in Swift make the provenance graph not so straightforward. Procedures input and

output datasets that may be composed of smaller datasets and may in turn be composed into larger datasets.

For example, a dataset D coming out of a procedure P may form a part of a larger dataset E. Dataset E may then be an input to procedure Q. The ordering is then:

```
 P --output--> D --contained-by-> E --input--> Q
```

Conversely, a dataset D coming out of a procedure P may contain a smaller dataset E. Dataset E may then be used as an input to procedure Q.

```
 P --output--> D --contains--> E --input--> Q
```

So the contains relation and its reverse, the contained-by relation, do not in the general case seem to give an appropriate preceeds relation.

```
so: i) should Q1<->Q be a bidirection dependency (in which case we
  no longer have a DAG, which causes trouble)

or

    ii) the dependency direction between Q1 and Q depends on how Q and Q1
were constructed. I think this is the better approach, because I think
there really is some natural dependency order.

If A writes to Q1 and Q1 is part of Q then A->Q1->Q
If A writes to Q and Q1 is part of Q then A->Q->Q1

So when we write to a dataset, we need to propagate out the dependency
from there (both upwards and downwards, I think).

eg. if Q1X is part of Q1 is part of Q
and A writes to Q1, then Q1X depends on Q1 and Q depends on Q1.
```

Various ways of doing closure - we have various relations in the graph such as dataset containment and procedure input/output. Need to figure out how this relates to predecessor/successors in the provenance sense.

```
A(
Also there are multiple levels of tracking (see the section on that):

If an app procedure produces eg
volume v, consisting of two files v.img and v.hdr (the fmri example)
then what is the dependency here? I guess v.img and v.hdr is the
output... (so in the present model there will never be
downward propagation as every produced dataset will be produced out of
base files. however its worth noting that this perhaps not always the
case...)

Alternatively we can model at the level of the app procedure, which in
the above case returns a volume v.

I guess this is similar to the case of the compound procedures vs
contained app procedures...

If we model at the level of files, then we don't really need to know
```

```
about higher datasets much?

Perhaps for now should model at level of procedure calls
)A

List A()A above as an issue and pick one choice - for now, lowest=file
production, so that all intermediate and output datasets will end up
with a strictly upward dependency

This rule does not deal with input-only datasets (that is, datasets
which we do not know where they came from). It would be fairly natural
with the above choice to again make dependencies from files upward.

So for now, dataset dependency rule is:

  * parent datasets depend on their children.

Perhaps?
```

8.2.2. Transitivity of relations in query language

One of my biggest concerns in query languages such as SQL and XPath is lack of decent transitive query ability.

I think we need a main relation, the *preceeds* relation. None of the relations defined in the source provenance data provides this relation.

The relation needs to be such that if any dataset or program Y that contributed to the production of any other dataset or program Y, then X preceeds Y.

We can construct pieces of this relation from the existing relations:

- There are fairly simple rules for procedure inputs and outputs: A dataset passed as an input to a procedure preceeds that procedure. Similarly, a procedure that outputs a dataset preceeds that dataset.
- Hierarchical datasets are straightforward to describe in the present implementation. Composite data structures are always described in terms of their members, so the members of a data structure always preceed the structures that contain them. [not true, i think - we can pass a struct into a procedure and have that procedure populate multiple contained files... bleugh]
- The relation is transitive, so the presence of some relations by the above rules will imply the presence of other relations to ensure transitivity.

8.3. Unique identification of provenance objects

A few issues - what are the objects that should be identified? (semantics); and how should the objects be identified? (syntax).

8.3.1. provenence object identifier syntax

For syntax, I favour a URI-based approach and this is what I have implemented in the prototypes. URIs rovide a ready made system for identifying different kinds of objects in different ways within the same syntax. which should be useful for the querys that want to do that. file, gsiftp URIs for filenames. probably should be normalising file URIs to refer to a specific hostname? otherwise they're fairly meaningless outside of one host... also, these name files but files are mutable.

its also fairly straightforward to subsume other identifier schemes into URIs (for example, that is already done for

UUIDs, in RFC4122).

for other IDs, such as workflow IDs, a tag or uuid URI would be nice.

cite: RFC4151[8]

cite: RFC4122[9]

8.3.1.1. tag URIs

tag URIs for identifiers of provenance objects:

all URIs allocated according to this section are labelled beginning with one of:

```
tag:benc@ci.uchicago.edu,2007:swift:
tag:benc@ci.uchicago.edu,2008:
```

for datasets identified only within a run (that is, for example, anything that doesn't have a filename): tag:benc@ci.uchicago.edu,2007:swift:dataset:TIMESTAMP:SEQ with TIMESTAMP being a timestamp of sometime near the start of the run, intending to be a unique workflow id (probably better to use the run-id) and SEQ being a sequence number. However, shouldn't really be pulling any information out of these time and seq fields.

for executes - this is based on the karajan thread ID and the log base filename (which is assumed to be a globally unique identifying string): tag:benc@ci.uchicago.edu,2007:swiftlogs:execute:WFID:THREAD with, as for datasets, WFID is a workflow-id-like entity.

8.3.2. Dataset identifier semantics

At present, dataset identifiers are formed uniquely for every dataset object created in the swift runtime (unique across JVMs as well as within a JVM).

This provides an overly sensitive(?) identity - datasets which are the same will be given different dataset identifiers at different times/places; although two different datasets will never be given the same identifier.

A different approach would be to say 'datasets are made of files, so we want to identify files, and files already have identiers called filenames'.

I think this approach is also insufficient.

The assertion 'datasets are made of files' is not correct. Datasets come in several forms: typed files, typed simple values, and typed collections of other datasets. Each of these needs a way to identify it.

Simple values are probably the easiest to identify. They can be identified by their own value and embedded within a suitable URI scheme. For example, a dataset representing the integer 7 could be identified as:

```
tag:benc@ci.uchicago.edu,2008:swift:types:int:7
```

This would have the property that all datasets representing the integer 7 would be identical (that is, have the same identifier).

Collections of datasets are more complicated. One interesting example of something that feels to me quite similar is the treatment of directories in hash-based file systems, such as git. In this model, a collection of datasets would be

[8] http://www.rfc-editor.org/rfc/rfc4151.txt
[9] http://www.rfc-editor.org/rfc/rfc4122.txt

represented by a hash of a canonical representation of its contenst, for example, a dataset consisting of a three element array of three files in this order: "x-foo:red", "x-foo:green" and "x-foo:blue" might be represented as:

```
tag:benc@ci.uchicago.edu,2008:collection:QHASH
```

where:

```
QHASH := sha1sum(QLONG)
```

```
QLONG := "[0] x-foo:red [1] x-foo:green [2] x-foo:blue"
```

This allows a repeatable computation of dataset identifiers given only knowledge of the contents of the dataset. Specifically it does not rely on a shared database to map content to identifier. However, it can only be computed when the content of the dataset is fully known (roughly equivalent to when the dataset is closed in the Swift runtime)

For identifying a dataset that is a file, there are various properties. Filename is one property. File content is another property. It seems desirable to distinguish between datasets that have the same name yet have different content, whilst identifying datasets that have the same content. To this end, an identifier might be constructed from both the filename and a hash of the content.

for prototype could deal only with files staged to local system, so that we can easily compute a hash over the content.

related to taking md5sums, kickstart provides the first few bytes of certain files (the executable and specified input and output files); whilst useful for basic sanity checks, there are very strong correlations with magic numbers and common headers that make this a poor content identifying function. perhaps it should be absorbed as dataset metadata if its available?

TODO the following para needs to rephrase as justification for having identities for dataset collections ::: at run-time when can we pick up the identities from other runs? pretty much we want identity to be expressed in some way so that we can get cross-run linkup. how do we label a dataset such that we can annotate it - eg in fmri example, how do we identify the input datasets (as file pairs) rather than the individual files?

Its desirable to give the same dataset the same identifier in multiple runs; and be able to figure out that dataset identifier outside of a run, for example for the purposes of dealing with metadata that is annotating a dataset.

8.3.3. File content tracking

identify file contents with md5sum (or other hash) - this is somewhat expensive, but without it we have (significantly) lessened belief in what the contents of a file are - we would otherwise, I think, be using only names and relying on the fact that those names are primary keys to file content (which is not true in general). so this should be perhaps optional. plus where to do it? various places... in wrapper.sh?

References here for using content-hashes: git, many of the DHTs (freenet, for example - amusing to cite the classic freenet gpl.txt example)

8.4. Type representation

how to represent types in this? for now use names, but that doesn't go cross-program because we can define a different type with the same name in every different program. hashtree of type definitions?

8.5. representation of workflows

perhaps need a workflow object that acts something like a namespace but implicitly definted rather than being user labelled (hence capturing the actual runtime space rather than what the user claims). that's the runID, I guess.

Also tracking of workflow source file. Above-mentioned reference to tracking file contents applies to this file too.

8.6. metadata extraction

provenance challenge I question 5 reports about pulling fields out of the headers of one of the input files. There's a program, scanheader, that extracts this info. Related but not actually useful, I think, for this question is that header fields could be mapped into SwiftScript if we allowed value+file simultaneous data structures.

8.7. source code recreation

should the output of the queries be sufficient to regenerate the data? the most difficult thing here seems to be handling data sets - we have the mapping tree for a dataset, but what is the right way to specify that in swift syntax? maybe need mapper that takes a literal datastructure and maps the filenames from it. though that doesn't account for file contents (so this bit of this point is the file contents issue, which should perhaps be its own chapter in this file)

8.8. Input parameters

Should also work on workflows which take an input parameter, so that we end up with the same output file generated several times with different output values - eg pass a string as a parameter and write that to 'output.txt' - every time we run it, the file will be different, and we'll have multiple provenance reports indicating how it was made, with different parameters. that's a simple demonstration of the content-tracking which could be useful.

If we're tracking datasets for simple values, I think we get this automatically. The input parameters are input datasets in the same way that input files are input datasets; and so fit into the model in the same way.

9. Open Provenance Model (OPM)

9.1. OPM-defined terms and their relation to Swift

OPM defines a number of terms. This section describes how those terms relate to Swift.

artifact: This OPM term maps well onto the internal Swift representation of `DSHandle`s. Each DSHandle in a Swift run is an OPM artifact, and each OPM artifact in a graph is a DSHandle.

collection: OPM collections are a specific kind of artifact, containing other artifacts. This corresponds with DSHandles for composite data types (structs and arrays). OPM has collection accessors and collection constructors which correspond to the `[]` and `.` operators (for accessors) and various assignment forms for constructors.

process: An OPM process corresponds to a number of Swift concepts (although they are slowly converging in Swift to a single concept). Those concepts are: procedure invocations, function calls, and operators.

agent: There are several entities which can act as an agent. At the highest level where only Swift is involved, a run of the `swift` commandline client is an agent which drives everything. Some other components of Swift may be regarded as agents, such as the client side wrapper script. For present OPM work, the only agent will be the Swift command line client invocation.

account: For present OPM work, there will be one account per workflow run. In future, different levels of granularity that could be expressed through different accounts might include representing compound procedure calls as processes vs representing atomic procedures calls explicitly.

OPM graph: there are two kinds of OPM graph that appear interesting and straightforward to export: i) of entire provenance database (thus containing multiple workflow runs); ii) of a single run

9.2. OPM links

Open Provenance Model at ipaw.info[10]

9.3. Swift specific OPM considerations

non-strictness: Swift sometimes lazily constructs collections (leading to the notion in Swift of an array being closed, which means that we know no more contents will be created, somewhat like knowing we've reached the end of a list). It may be that an array is never closed during a run, but that we still have sufficient provenance information to answer useful queries (for example, if we specify a list [1:100000] and only refer to the 5th element in that array, we probably never generate most of the DSHandles... so an explicit representation of that array in terms of datasets cannot be expressed - though a higher level representation of it in terms of its constructor parameters can be made) (?)

aliasing: (this is related to some similar ambiguity in other parts of Swift, to do with dataset roots - not provenance related). It is possible to construct arrays by explicitly listing their members:

```
int i = 8;
int j = 100;
int a[] = [i,j];
int k = a[1];
// here, k = 8
```

The dataset contained in `i` is an artifact (a literal, so some input artifact that has no creating process). The array `a` is an artifact created by the explicit array construction syntax `[memberlist]` (which is an OPM process). If we then model the array accessor syntax `a[1]` as an OPM process, what artifact does it return? The same one or a different one? In OPM, we want it to return a different artifact; but in Swift we want this to be the same dataset... (perhaps explaining this with `int` type variables is not the best way - using file-mapped data might be better) TODO: what are the reasons we want files to have a single dataset representation in Swift? dependency ordering - definitely. cache management? Does this lead to a stronger notion of aliasing in Swift?

Provenance of array indices: It seems fairly natural to represent arrays as OPM collections, with array element extraction being a process. However, in OPM, the index of an array is indicated with a role (with suggestions that it might be a simple number or an XPath expression). In Swift arrays, the index is a number, but it has its own provenance, so by recording only an integer there, we lose provenance information about where that integer came from - that integer is a Swift dataset in its own right, which has its own provenance. It would be nice to be able to represent that (even if its not standardised in OPM). I think that needs re-ification of roles so that they can be described; or it needs treatment of [] as being like any other binary operator (which is what happens inside swift) - where the LHS and RHS are artifacts, and the role is not used for identifying the member (which would also be an argument for making array element extraction be treated more like a plain binary operator inside the Swift compiler and runtime)

provenance of references vs provenance of the data in them: the array and structure access operators can be used to acquire `DSHandle`s which have no value yet, and which are then subsequently assigned. In this usage, the provenance of the containing structure should perhaps be that it is constructed from the assignments made to its members, rather than the other way round. There is some subtlety here that I have not fully figured out.

Piecewise construction of collections: arrays and structs can be constructed piecewise using `. =` and `[] =`. how is this to be represented in OPM? perhaps the closing operation maps to the OPM process that creates the array, so that it ends up looking like an explicit array construction, happening at the time of the close?

Provenance of mapper parameters: mapper parameters are artifacts. We can represent references to those in a Swift-specific part of an artifacts value, perhaps. Probably not something OPM-generalisable.

10. Processing i2u2 cosmic metadata

i2u2 cosmic metadata is extracted from a VDS1 VDC.

[10] http://twiki.ipaw.info/bin/view/Challenge/OPM

TODO some notes here about how I dislike the inner-plaform effect in the metadata part of the VDS1 VDC.

to launch postgres on soju.hawaga.org.uk:

```
sudo -u postgres /opt/local/lib/postgresql82/bin/postgres -D  /opt/local/var/db/postgresql
```

and then to import i2u2 vdc data as VDC1 vdc:

```
$ /opt/local/lib/postgresql82/bin/createdb -U postgres i2u2vdc1
CREATE DATABASE
$ psql82 -U postgres -d i2u2vdc1 < work/i2u2.vdc
gives lots of errors like this:
ERROR:  role "portal2006_1022" does not exist
because indeed that role doesn't exist
but I think that doesn't matter for these purposes - everything will end
up being owned by the postgres user which suffices for what I want to do.
```

annotation tables are:

```
 public | anno_bool       | table | postgres   29214 rows
  this is boolean values

 public | anno_call       | table | postgres   0 rows
- this is a subject table. also has did

 public | anno_date       | table | postgres   52644 rows
   this is date values

 public | anno_definition | table | postgres   1849 rows
    this is XML-embedded derivations (values / objects)

 public | anno_dv         | table | postgres   0 rows
- this is a subject table. also has did

 public | anno_float      | table | postgres   27966 rows
    this is float values

 public | anno_int        | table | postgres   58879 rows
    this is int values

 public | anno_lfn        | table | postgres   411490 rows
    this is the subject record for LFN subjects - subjects have an
    mkey (predicate) column

 public | anno_lfn_b      | table | postgres
this appears to be keyed by did field - ties dids to what looks like
LFNs

 public | anno_lfn_i      | table | postgres
 public | anno_lfn_o      | table | postgres
likewise these two

 public | anno_targ       | table | postgres
is this a subject table? it has an mkey value that always appears to be
'description' and then has a name column which lists invocation parameter
names and ties them to dids.

 public | anno_text       | table | postgres   242824 rows
text values (objects)
```

```
 public | anno_tr         | table | postgres
```

most of the interesting data starts in anno_lfn because data is mostly annotating LFNs:

```
i2u2vdc1=# select * from anno_lfn limit 1;
 id |        name         |   mkey
----+---------------------+----------
  2 | 180.2004.0819.0.raw | origname
```

There are 63 different mkeys (predicates in RDF-speak):

```
i2u2vdc1=# select distinct mkey from anno_lfn;
             mkey
-------------------------------
 alpha
 alpha_error
 author
 avgaltitude
 avglatitude
 avglongitude
 background_constant
 background_constant_error
 bins
 blessed
 caption
 chan1
 chan2
 chan3
 chan4
 channel
 city
 coincidence
 comments
 cpldfrequency
 creationdate
 date
 description
 detectorcoincidence
 detectorid
 dvname
 enddate
 energycheck
 eventcoincidence
 eventnum
 expire
 filename
 gate
 gatewidth
 group
 height
 julianstartdate
 lifetime(microseconds)
 lifetime_error(microseconds)
 name
 nondatalines
 numBins
 origname
 plotURL
 project
```

```
 provenance
 radius
 rawanalyze
 rawdate
 school
 source
 stacked
 startdate
 state
 study
 teacher
 thumbnail
 time
 title
 totalevents
 transformation
 type
 year
(63 rows)
```

so work on a metadata importer for i2u2 cosmic that will initially deal with only the lfn records.

There are 19040 annotated LFNs, with 411490 annotations in total, so about 21 annotations per LFN.

The typing of the i2u2 data doesn't support metadata objects that aren't swift workflow entities - for example high schools as objects in their own right - the same text string is stored as a value over and over in many anno_text rows. A more generalised Subject-Predicate-Object model in RDF would have perhaps a URI for the high school, with metadata on files tying files to a high school and metadata on the high school object. In SQL, that same could be modelled in a relational schema.

Conversion of i2u2 VDS1 VDC LFN/text annotations into an XML document using quick hack script took 32mins on soju, my laptop. resulting XML is 8mb. needed some manual massage to remove malformed embedded xml and things like that.

```
./i2u2-to-xml.sh >lfn-text-anno.xml
```

so we end up with a lot of records that look like this:

```
<lfn name="43.2007.0619.0.raw">
<origname>rgnew.txt</origname>
<group>riogrande</group>
<teacher>Luis Torres Rosa</teacher>
<school>Escuelo Superior Pedro Falu</school>
<city>Rio Grande</city>
<state>PR</state>
<year>AY2007</year>
<project>cosmic</project>
<comments></comments>
<detectorid>43</detectorid>
<type>raw</type>
<avglatitude>18.22.8264</avglatitude>
<avglongitude>-65.50.1975</avglongitude>
<avgaltitude>-30</avgaltitude>
</lfn>
```

The translation here is not cosmic-aware - the XML tag is the mkey name from vdc and the content is the value. So we get all the different metadata (informal) schemas that appear to have been used, translated.

Output the entire provenance database:

```
$ time cat lfn-text-anno.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '/c
 10178037

real    0m2.624s
user    0m2.612s
sys     0m0.348s
```

Select all LFN objects (which on this dataset means everything one layer down):

```
$ time cat lfn-text-anno.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '/c
 9618818

real    0m2.692s
user    0m2.703s
sys     0m0.337s
```

Try to select an LFN that doesn't exist, by specifying a filename that is not there:

```
$ time cat lfn-text-anno.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '/c
<?xml version="1.0"?>
<toplevel/>

real    0m0.867s
user    0m0.740s
sys     0m0.143s
```

Similar query for a filename that does exist:

```
$ time cat lfn-text-anno.xml | ~/work/xpathtool-20071102/xpathtool/xpathtool.sh --oxml '/c
<?xml version="1.0"?>
<toplevel>
  <lfn name="1.2005.0801.0">
    <origname>C:\Documents and Settings\zsaleh\My Documents\Tera stuff\Qnet\Qnet Data\All_
    <group>TERA</group>
    <teacher>Marcus Hohlmann</teacher>
    <school>Florida Institute of Technology</school>
    <city>Melbourne</city>
    <state>FL</state>
    <year>AY2004</year>
    <project>cosmic</project>
    <comments/>
    <source>1.2005.0801.0</source>
    <detectorid>1</detectorid>
    <type>split</type>
  </lfn>
</toplevel>

real    0m0.875s
user    0m0.745s
sys     0m0.154s
```

11. processing fMRI metadata

for fmri, we can extract embedded image metadata using the scanheader utility.

associate that with the 'volume' dataset, not with the actual image data files. for now that means we need the datasets to have been labelled with their ID already, which is at the moment after execution has completed. that's fine for now with the retrospective provenance restriction of this immediate work. see the 'cross-run dataset ID' section, for which this also applies - we are generating dataset IDs outside of a particular run.

12. random unsorted notes

to put provdb in postgres instead of sqlite3: start as per i2u2 instructions, then / **opt/local/lib/postgresql82/bin/createdb -U postgres provdb** then: **psql82 -U postgres -d provdb < prov-init.sql** to initialise the db.

on terminable, made new database that is not the default system db install, by using existing postgres but running under my user id:

```
  131  mkdir pgplay
  133  chmod 0700 pgplay/
  135  initdb -D ~/pgplay/
  138  postmaster -D ~/pgplay/ -p 5435
$ createdb -p 5435 provdb
CREATE DATABASE
```

now can access like this:

```
$ psql -p 5435 -d provdb
provdb=# \dt
No relations found.
```

osg/gratia - how does this data tie in?

cedps logging - potential for info there but there doesn't seem anything particularly substantial at the moment

13. Provenance Challenge 1 examples

13.1. Basic SQL

13.1.1. provch q1

```
get the dataset id for the relevant final dataset:
sqlite> select * from dataset_filenames where filename like '%0001.jpeg';
14976260|file://localhost/0001.jpeg

get containment info for that file:
sqlite> select * from dataset_containment where inner_dataset_id = 14976260;
7316236|14976260
sqlite> select * from dataset_containment where inner_dataset_id = 7316236;
[no answer]

now need to find what contributed to those...

> select * from dataset_usage where dataset_id=14976260;
0-4-3|O|14976260

> select * from dataset_usage where execute_id='0-4-3' and direction='I';
0-4-3|I|4845856
```

```
qlite> select * from dataset_usage where dataset_id=4845856 and direction='O';
0-3-3|O|4845856

sqlite> select * from dataset_usage where execute_id='0-3-3' and direction='I';
0-3-3|I|3354850
0-3-3|I|6033476

qlite> select * from dataset_usage where (dataset_id=3354850 or dataset_id=6033476) and di
0-2|O|3354850

sqlite> select * from dataset_usage where execute_id='0-2' and direction='I';0-2|I|4436324

sqlite> select * from dataset_usage where dataset_id=4436324 and direction='O';
[no answer]

so here we have run out of places to keep going. however, I think this 4436324
is not an input - its related to another dataset. so we need another rule for
inference here...
```

13.1.2. prov ch q4

prov ch q4 incremental solutions:

first cut: this will select align_warp procedures and their start times. does not select based on parameters, and does not select based on day of week. (the former we don't have the information for; the latter maybe don't have the information in sqlite3 to do - or maybe need SQL date ops and SQL dates rather than unix timestamps)

```
sqlite> select id, starttime from invocation_procedure_names, executes where executes.id =
```

Next, this will display the day of week for an invocation:

```
select id, strftime('%w',starttime, 'unixepoch') from executes,invocation_procedure_names
0-0-3|5
0-0-4|5
0-0-1|5
0-0-2|5
```

And this will match day of week (sample data is on day 5, which is a Friday, not the day requested in the question):

```
sqlite> select id from executes,invocation_procedure_names where procedure_name='align_war
0-0-3
0-0-4
0-0-1
0-0-2
```

Now we bring in input data binding: we query which datasets were passed in as the model parameter for each of the above found invocations:

```
sqlite> select executes.id, dataset_usage.dataset_id from executes,invocation_procedure_na
0-0-3|11032210
0-0-4|13014156
```

```
0-0-1|14537849
0-0-2|16166946
```

though at the moment this doesn't give us the value of the parameter.

so now pull in the parameter value:

```
sqlite> select executes.id, dataset_usage.dataset_id, dataset_usage.value from executes,in
0-0-3|11032210|12
0-0-4|13014156|12
0-0-1|14537849|12
0-0-2|16166946|12
```

Now we can select on the parameter value and get our final answer:

```
sqlite> select executes.id from executes,invocation_procedure_names, dataset_usage where p
0-0-3
0-0-4
0-0-1
0-0-2
```

Note that in SQL in general, we *don't* get typing of the parameter value here so can't do anything more than string comparison. For example, we couldn't check for the parameter being greater than 12 or similar. In sqlite, it happens that its typing is dynamic enough to allow the use of relational operators like > on fields no matter what their declared type, because declared type is ignored. This would stop working if stuff was run on eg postgres or mysql, I think.

13.1.3. prov ch metadata

metadata: in the prov challenge, we annotate (some) files with their header info. in the provenance paper, we want annotations on more than just files.

for prov ch metadata, define a scanheader table with the result of scanheader on each input dataset, but do it *after* we've done the run (because we're then aware of dataset IDs)

There's a representation question here - the metadata is about a volume dataset which is a pair of files, not about a header or image file separately. how to represent this? we need to know the dataset ID for the volume. at the moment, we can know that after a run. but this ties into the identification of datasets outside of an individual run point - move this paragraph into that questions/discussions section.

should probably for each storage method show the inner-platform style of doing metadata too; associated queries to allow comparison with the different styles; speeds of metadata query for large metadata collections (eg. dump i2u2 cosmic metadata for real cosmic VDC)

13.2. SQL with transitive closures

13.2.1. prov ch question 1:

```
$ sqlite3 provdb
SQLite version 3.3.17
Enter ".help" for instructions
sqlite> select * from dataset_filenames where filename like '%0001.jpeg';
14976260|file://localhost/0001.jpeg
-- can query keeping relations
sqlite> select * from trans where after=14976260;
```

```
0-4-3|14976260
4845856|14976260
0-3-3|14976260
3354850|14976260
6033476|14976260
4825541|14976260
7061626|14976260
0-2|14976260
4436324|14976260
11153746|14976260
655223|14976260
5169861|14976260
6487148|14976260
5772360|14976260
4910675|14976260
7202698|14976260
12705705|14976260
2088036|14976260
13671126|14976260
14285084|14976260
12896050|14976260
0-1-3|14976260
0-1-4|14976260
0-1-2|14976260
0-1-1|14976260
2673619|14976260
9339756|14976260
10682109|14976260
8426950|14976260
16032673|14976260
2274050|14976260
1461238|14976260
13975694|14976260
9282438|14976260
12766963|14976260
8344105|14976260
9190543|14976260
14055055|14976260
2942918|14976260
12735302|14976260
7080341|14976260
0-0-3|14976260
0-0-4|14976260
0-0-2|14976260
0-0-1|14976260
2307300|14976260
11032210|14976260
16166946|14976260
14537849|14976260
13014156|14976260
6435309|14976260
6646123|14976260
-- or can query without relations:
sqlite> select before from trans where after=14976260;
0-4-3
4845856
0-3-3
3354850
6033476
4825541
7061626
0-2
4436324
11153746
```

```
655223
5169861
6487148
5772360
4910675
7202698
12705705
2088036
13671126
14285084
12896050
0-1-3
0-1-4
0-1-2
0-1-1
2673619
9339756
10682109
8426950
16032673
2274050
1461238
13975694
9282438
12766963
8344105
9190543
14055055
2942918
12735302
7080341
0-0-3
0-0-4
0-0-2
0-0-1
2307300
11032210
16166946
14537849
13014156
6435309
6646123
```

14. Representation of dataset containment and procedure execution in r2681 and how it could change.

Representation of processes that transform one dataset into another dataset at present only occurs for `app` procedures, in logging of `vdl:execute` invocations, in lines like this:

```
2009-03-12 12:20:29,772+0100 INFO  vdl:parameterlog PARAM thread=0-10-1 direction=input va
```

and dataset containment is represented at closing of the containing DSHandle by this:

```
2009-03-12 12:20:30,205+0100 INFO  AbstractDataNode CONTAINMENT parent=tag:benc@ci.uchicag
2009-03-12 12:20:30,205+0100 INFO  AbstractDataNode ROOTPATH dataset=tag:benc@ci.uchicago.
```

This representation does not represent the relationship between datasets when they are related by @functions or operators. Nor does it represent causal relationships between collections and their members - instead it represents containment.

Adding representation of operators (including array construction) and of @function invocations would give substantially more information about the provenance of many more datasets.