Swift Provenance Database

Contents

1 Introduction

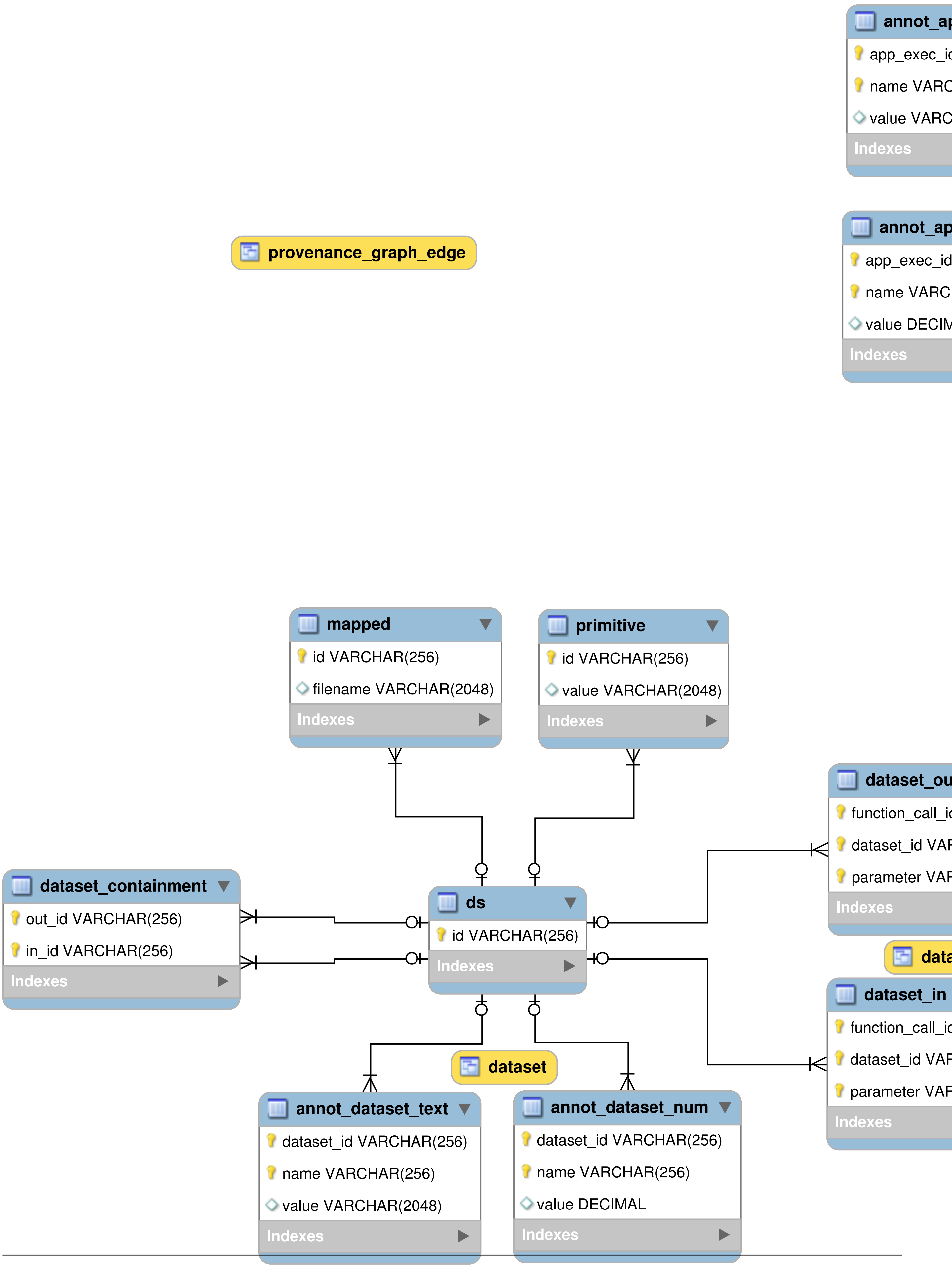
Swift can be configured to gather and store provenance information about script executions. The following tools are available:

1. A set of scripts for extracting provenance information from Swift's log files. The extracted data is imported into a relational database, currently PostgreSQL, where it can queried.
2. A query interface for provenance with a built-in query language called SPQL (Swift Provenance Query Language). SPQL is similar to SQL except for not having `FROM`-clauses and join expressions on the `WHERE`-clause, which are automatically computed for the user. A number of functions and stored procedures that abstract common provenance query patterns are available in both SPQL and SQL.

The tools for managing provenance information in Swift have the following features:

- Gathering of producer-consumer relationships between data sets and processes.
- Gathering of hierarchical relationships between data sets.
- Gathering of script source code used in each execution.
- Allows users to enrich their provenance records with annotations.
- Gathering of runtime information about application executions.
- Provides a usable and useful query interface for provenance information.

A UML diagram of this provenance model is presented in figure [?informalfigure]. We simplify the UML notation to abbreviate the information that each annotated entity set (script run, function call, and variable) has one annotation entity set per data type. We define entities that correspond to the Open Provenance Model (OPM) notions of artifact, process, and artifact usage (either being consumed or produced by a process). Annotations, which can be added post-execution, represent information about provenance entities such as object version tags and scientific parameters.


annot_ap
app_exec_ic
name VARC
value VARC
Indexes
annot_ap
app_exec_id
name VARC
value DECIM
Indexes
provenance_graph_edge
mapped
id VARCHAR(256)
filename VARCHAR(2048)
Indexes
primitive
id VARCHAR(256)
value VARCHAR(2048)
Indexes
dataset_ou
function_call_ic
dataset_id VAF
parameter VAF
Indexes
data
dataset_containment
out_id VARCHAR(256)
in_id VARCHAR(256)
Indexes
ds
id VARCHAR(256)
Indexes
dataset_in
function_call_ic
dataset_id VAF
parameter VAF
Indexes
dataset
annot_dataset_text
dataset_id VARCHAR(256)
name VARCHAR(256)
value VARCHAR(2048)
Indexes
annot_dataset_num
dataset_id VARCHAR(256)
name VARCHAR(256)
value DECIMAL
Indexes

`script`: contains the script source code used and its hash value.

`script_run`: refers to the execution (successful or unsuccessful) of a script, with attributes such as start time, source code filename, and Swift's version.

`function_call`: records calls to functions within a script execution. These calls take as input data sets, such as values stored in primitive variables or files referenced by mapped variables; perform some computation specified in the respective function declaration; and produce data sets as output. In Swift, function calls can represent invocations of external applications, built-in functions, and operators; each function call is associated with the script run that invoked it.

`app_fun_call`: represents an invocation of an application function (*app function*). In Swift, it is generated by an invocation to an external application. External applications are listed in an application catalog along with the computational resources on which they can be executed.

`application_execution`: represents execution attempts of an external application. Each application function call triggers one or more execution attempts, where one (or, in the case of retries or replication, several) particular computational resource(s) will be selected to actually execute the application.

`runtime_info`: contains information associated with an application execution, such as resource consumption.

`dataset`: represents data sets that were assigned to variables in a Swift script.

`annot`: is a key-value pair associated with either a `variable`, `function_call`, or `script_run`. The annotations are free-form and can be used, for instance, to record scientific-domain parameters, object versions, and user identities.

The `dataset_in` and `dataset_out` relationships between `function_call` and `variable` define a lineage graph that can be traversed to determine ancestors or descendants of a particular entity. Process dependency and data dependency graphs are derived with transitive queries over these relationships.

2 Design and Implementation of Swift Provenance Database

The Swift Provenance Database design is influenced by our survey of provenance queries in many-task computing. The *multiple-step relationships* (R^*) pattern is implemented by queries that follow the transitive closure of basic provenance relationships, such as data containment hierarchies, and data derivation and consumption. The *run correlation* (RCr) pattern is implemented by queries for correlating attributes from multiple script runs, such as annotation values or the values of function call parameters.

2.1 Provenance Gathering and Storage

Swift can be configured to add both prospective and retrospective provenance information to the log file it creates to track the behavior of each script run. The provenance extraction mechanism processes these log files, filters the entries that contain provenance data, and exports this information to a relational SQL database. Each application execution is launched by a wrapper script that sets up the execution environment. We modified these scripts to also gather runtime information, such as memory consumption and processor load. Additionally, one can define a script that generates annotations in the form of key-value pairs, to be executed immediately before the actual application. These annotations can be exported to the provenance database and associated with the respective application execution. Swift Provenance Database processes the data logged by each wrapper to extract both the runtime information and the annotations, storing them in the provenance database. Additional annotations can be generated per script run using *ad-hoc* annotator scripts. In addition to retrospective provenance, Swift Provenance Database keeps prospective provenance by recording the Swift script source code, the application catalog, and the site catalog used in each script run.

2.2 Query Interface

During the Third Provenance Challenge, we observed that expressing provenance queries in SQL is often cumbersome. For example, such queries require extensive use of complex relational joins, for instance, which are beyond the level of complexity that most domain scientists are willing, or have the time, to master and write. Such usability barriers are increasingly being seen as a critical issue in database management systems. Jagadish et al. propose that ease of use should be a requirement as important as functionality and performance. They observe that, even though general-purpose query languages such as SQL and XQuery allow for the design of powerful queries, they require detailed knowledge of the database schema and rather complex programming to express queries in terms of such schemas. Since databases are often normalized, data is spread through different relations requiring even more extensive use of database join operations when designing queries. Some of the approaches used to improve usability are forms-based query interfaces, visual query builders, and schema summarization.

translated into a SQL query that is processed by the underlying relational database. While the syntax of SPQL is by design similar to SQL, it does not require detailed knowledge of the underlying database schema for designing queries, but rather only of the entities in a simpler, higher-level abstract provenance schema, and their respective attributes.

The basic building block of a SPQL query consists of a selection query with the following format:

```
select (distinct) selectClause
(where            whereClause
(group by         groupByClause
(order by         orderByClause)))
```

This syntax is very similar to a selection query in SQL, with a critical usability benefit: hide the complexity of designing extensive join expressions. One does not need to provide all tables of the from clause. Instead, only the entity name is given and the translator reconstructs the underlying entity that was broken apart to produce the normalized schema. As in the relational data model, every query or built-in function results in a table, to preserve the power of SQL in querying results of another query. Selection queries can be composed using the usual set operations: union, intersect, and difference. A `select` clause is a list with elements of the form `<entity set name>(.<attribute name>)` or `<built-in function name>(.<return attribute name>)`. If attribute names are omitted, the query returns all the existing attributes of the entity set. SPQL supports the same aggregation, grouping, set operation and ordering constructs provided by SQL.

To simplify the schema that the user needs to understand to design queries, we used database views to define the higher-level schema presentation shown in Figure. This abstract, OPM-compliant provenance schema, is a simplified view of the physical database schema detailed in section. It groups information related to a provenance entity set in a single relation. The annotation entity set shown is the union of the annotation entity sets of the underlying database, presented in Figure. To avoid defining one annotation table per data type, we use dynamic expression evaluation in the SPQL to SQL translator to determine the required type-specific annotation table of the underlying provenance database.

- `ancestors(object_id})` returns a table with a single column containing the identifiers of variables and function calls that precede a particular node in a provenance graph stored in the database.
- `data_dependencies(variable_id})`, related to the previous built-in function, returns the identifiers of variables upon which `variable_id` depends.
- `function_call_dependencies(function_call_id})` returns the identifiers of function calls upon which `function_call_id` depends.
- `compare_run(list of <function_parameter=string | annotation_key=string)` shows how process parameters or annotation values vary across the script runs stored in the database.

The underlying SQL implementation of the `ancestor` built-in function, below, uses recursive Common Query Expressions, which are supported in the SQL:1999 standard. It uses the `prov\_graph` database view, which is derived from the `dataset\_in` and `dataset_out` tables, resulting in a table containing the edges of the provenance graph.

```
CREATE FUNCTION ancestors(varchar)  RETURNS SETOF varchar AS $$
WITH RECURSIVE anc(ancestor,descendant) AS
  (
       SELECT parent AS ancestor, child AS descendant
       FROM   prov_graph
       WHERE child=$1
     UNION
       SELECT prov_graph.parent AS ancestor,
              anc.descendant AS descendant
       FROM   anc, prov_graph
       WHERE  anc.ancestor=prov_graph.child
  )
  SELECT ancestor FROM anc $$ ;
```

To further simplify query specification, SPQL uses a generic mechanism for computing the {em from} clauses and the join expressions of the `where` clause for the target SQL query. The SPQL to SQL query translator first scans all the entities present in the SPQL query. A shortest path containing all these entities is computed in the graph defined by the schema of the provenance database. All the entities present in this shortest path are listed in the `from` clause of the target SQL query. The join expressions of the `where` clause of the target query are computed using the edges of the shortest path, where each edge derives an expression that equates the attributes involved in the foreign key constraint of the entities that define the edge. While this automated join computation facilitates query design, it does somewhat reduce the expressivity of SPQL, as one is not able to perform other types of joins, such as self-joins, explicitly. However, many such queries can be expressed using subqueries, which are supported by SPQL. While some of the expressive power of SQL is thus lost, we show in the sections that follow that SPQL is able to express, with far less effort and complexity, most important and useful queries that provenance query patterns require. As a quick taste, this SPQL query returns the identifiers of the script runs that either produced or consumed the file `nr`:

```
select  compare_run(parameter='proteinId').run_id  where  file.name='nr';
```

This SPQL query is translated by Swift Provenance Database to the following SQL query:

```
select compare_run1.run_id
from   select run_id, j1.value AS proteinId
       from compare_run_by_param('proteinId') as compare_run1,
       run, proc, ds_use, ds, file
where  compare_run1.run_id=run.id and ds_use.proc_id=proc.id and
       ds_use.ds_id=ds.id and ds.id=file.id and
       run.id=proc.run_id and file.name='nr';
```

Further queries are illustrated by example in the next section. We note here that the SPQL query interface also lets the user submit standard SQL statements to query the database.

3 Tutorial

Swift Provenance Database is a set of scripts, SQL functions and stored procedures, and a query interface. It extracts provenance information from Swift's log files into a relational database. The tools are downloadable through SVN with the command:

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

3.1 Database Configuration

Swift Provenance Database depends on PostgreSQL, version 9.0 or later, due to the use of *Common Table Expressions* for computing transitive closures of data derivation relationships, supported only on these versions. The file `prov-init.sql` contains the database schema, and the file `pql_functions.sql` contain the function and stored procedure definitions. If the user has not created a provenance database yet, this can be done with the following commands (one may need to add "`-U` *username*" and "`-h` *hostname*" before the database name "`provdb`", depending on the database server configuration):

```
createdb provdb
psql -f prov-init.sql provdb
psql -f pql-functions.sql provdb
```

3.2 Swift Provenance Database Configuration

The file `etc/provenance.config` should be edited to define the database configuration. The location of the directory containing the log files should be defined in the variable `LOGREPO`. For instance:

```
export LOGREPO=~/swift-logs/
```

The command used for connecting to the database should be defined in the variable SQLCMD. For example, to connect to CI's PostgreSQL? database:

```
export SQLCMD="psql -h db.ci.uchicago.edu -U provdb provdb"
```

The script `./swift-prov-import-all-logs` will import provenance information from the log files in `$LOGREPO` into the database. The command line option `-rebuild` will initialize the database before importing provenance information.

3.3 Swift Configuration

To enable the generation of provenance information in Swift's log files the option `provenance.log` should be set to true in `etc/swift.properties`:

```
provenance.log=true
```

If Swift's SVN revision is 3417 or greater, the following options should be set in `etc/log4j.properties`:

```
log4j.logger.swift=DEBUG
log4j.logger.org.griphyn.vdl.karajan.lib=DEBUG
```

3.3.1 Enriching Provenance Data with Runtime Resource Consumption Statistics

A modified version of `_swiftwrap` can be used to gather additional information on runtime resource comsumption, such as processor, memory, I/O, and swap use. One should backup the original `_swiftwrap` script and replace it with the modified one:

```
cp $SWIFT_HOME/libexec/_swiftwrap $SWIFT_HOME/libexec/_swiftwrap-backup
cp swift_mod/_swiftwrap_runtime_snapshots $SWIFT_HOME/libexec/_swiftwrap
```

3.4 Example: MODIS

Run MODIS.

```
swift modis.swift
swift-prov-import-all-logs
```

Connect to the provenance database:

```
psql provdb
```

List runs that were imported to the database:

```
SELECT script_filename, swift_version, cog_version, final_state, start_time, duration
FROM   script_run;

  script_filename   | swift_version | cog_version | final_state |         start_time  ←
              | duration
--------------------+---------------+-------------+-------------+----------------------------+------

 modis.swift        | 5746          | 3371        | FAIL        | 2012-09-19  ←
    17:26:19.221-03 |    2.168
 modis-vortex.swift | 5746          | 3371        | FAIL        | 2012-09-19  ←
    17:28:24.809-03 |  180.542
 modis-vortex.swift | 5746          | 3371        | FAIL        | 2012-09-19  ←
    17:31:55.706-03 |  312.249
```

```
select * from ancestors('dataset:20120919-1731-06svjllb:720000000654');

                          ancestors
 pass:[------------------------------------------------------------]
 modis-vortex-20120919-1731-6fa0kk03:0
 modis-vortex-20120919-1731-6fa0kk03:0-6
 dataset:20120919-1731-06svjllb:720000000335
 dataset:20120919-1731-06svjllb:720000000653
 dataset:20120919-1731-06svjllb:720000000007
 modis-vortex-20120919-1731-6fa0kk03:06svjllb:720000000335
 dataset:20120919-1731-06svjllb:720000000336
 dataset:20120919-1731-06svjllb:720000000337
 ...
 dataset:20120919-1731-06svjllb:720000000042
 dataset:20120919-1731-06svjllb:720000000229
 dataset:20120919-1731-06svjllb:720000000006
(958 rows)
```