Analysis and reporting of service data using the SAFE

Stephen Booth
EPCC
University of Edinburgh
Edinburgh, UK
s.booth@ed.ac.uk

***Abstract*—The SAFE (Service Administration from EPCC) is a user services system developed by EPCC that handles user management and report generation for all our HPC services including the Cray services. An important function of this system is the ingestion of accounting data into the database and the generation of usage reports. This presents the design and implementation of this reporting system.**

Keywords-component; reporting, accounting, database

I. INTRODUCTION

The SAFE (Service Administration from EPCC) is a software system developed by EPCC to administer High Performance Computing services. It is a web based system built using java-servlets[1] and a SQL relational database. It can be responsible for almost all aspects of user management ranging from user registration and helpdesk to resource management and reporting. One of the big advantages of using a single integrated system is that it allows many operations to be delegated to appropriate users. For example a project manager can approve access to, generate reports on, and manage resources within their projects.

The generation of reports and the analysis of system use is an important activity for any HPC service. There are many different groups that may require different levels of access to this data. Individual users need to be able to access information on their own use of the system. Project managers need to generate overview reports on the use by their project. System operators and funding agencies need reports on the overall use of the system. In addition reports need to be generated from a variety of different data sources. The Reporting sub-system of the SAFE[2] is capable of ingesting data from a variety of sources including most major batch systems. It uses a system of plug-in parsers to allow addition types of data source to be easily added to the system. The reporting system is not restricted to batch job information and can also handle such diverse information as project resource allocations, file-transfer activity or disk usage data. In addition policy plug-ins can be used to trigger side-effects such as job charging and to augment the raw information based on local site policies. For example additional accounting properties can be derived based on the queue where the job runs or additional log files can be parsed to add additional information such as executables used.

II. REQUIREMENTS

The requirements on the SAFE reporting sub-system can be grouped into three main types.

A. Performance

The key performance requirements are driven by the large numbers of accounting records generated by typical HPC services. In our experience it is relatively common to require reports that cover several years of operation of a service. A large HPC service can easily run several million individual compute jobs over such a period. The reporting problem can be several orders of magnitude harder for services that need to support high throughput rather than high performance. This kind of workload typically consists of very large numbers of much smaller jobs, a high throughput service can generate several orders of magnitude more accounting records than a similarly sized HPC service.

B. Flexibility

We chose to build a general purpose reporting system rather than focus solely on the generation of reports of jobs run on the HPC resource. This allows the same framework to be used to generate reports on disk utilization and helpdesk activity. The general nature of the reporting system has also allowed us to re-use the code in other unrelated software projects.

One clear requirement was to support many different sources of accounting data. The SAFE has been under continual development since 2002. Over this time we have needed to support many different types of data. As well as data from a variety of batch systems, we have also had to import data from Unix process-level accounting and support data imports from a variety of Grid-middleware. Unlike some alternative accounting systems we did *not* choose to normalize all data into a single common internal format. Instead data from each different source is stored in a separate table where the data format can be customized to meet the current operational requirements. This reduces the

likelihood of loss of information as the format of the stored data closely follows the original form of the data. A normalized view of the data is still available but this is generated dynamically. This approach requires the system to be very flexible about the database format of the data. In practice this flexibility has proved to be invaluable as it has ensured that historic data never needs to be reformatted to maintain compatibility with newer versions of the SAFE. It also makes it significantly easier to extend the remit of the system to new kinds of data such as file-transfer data or energy consumption data.

C. Extensability

General usage reports that are used to track overall changes in the use of the service are of perpetual importance and are run on a frequent basis. These kinds of report look at known metrics and change very infrequently. However in addition to these fixed reports it is also useful to be able to easily add new metrics and new types of report in order to look for new insights into the operation of the service and find ways of improving its operation. It is therefore desirable to be able to add new and complex reports without the need for source code access to the reporting system.

III. DATA MODEL

Because of our need for a general and flexible reporting framework we use a property based data model. Each data record is viewed as a collection of key-value pairs representing the various properties in the data record. This abstraction results in a great deal of flexibility as the same code can process records representing different kinds of data.

The simplest implementation of a property based model would result in records being retrieved from the data-base and converted into an in-memory Java Object representation before being processed. This would in turn severely limit performance. In order to meet our performance targets we need to translate as many operations as possible into efficient SQL queries. We therefore introduce *Factory* classes for data records of similar types. These Factory classes roughly correspond to the data-base tables and as well as retrieving sets of records they also implement high level query operation (expressed in terms of properties) that can be mapped onto efficient SQL queries. Each Factory can provide a set of supported properties. The Records managed by the Factory are constrained to only implement properties from this set.

While many of the properties will map directly onto fields in the underlying database it is also possible to define properties as expressions over other supported properties. In the SAFE properties are strongly typed. The keys used to specify a property contain type information as well as the name of the property. This allows type constraints to be checked when creating expressions.

This ability to implement expressions is vital to support the flexibility and extensibility we require. It allows us to define new properties for example:

- WallClock = EndTime – StartTime
- Residency = Numnodes * WallClock
- SlowDown = (EndTime–StartTime)/ (EndTime-SubmitTime)

As well as supporting Numeric, Date and String; types, properties can also be defined as references to records in other data-base tables, which allows expressions to include values from referenced tables. For example:

- Charge = Residency * Machine[ChargeRate]

When mapped to SQL this requires a JOIN to the referenced table.

By defining new derived properties in this fashion we can add a set of standard properties to each accounting table producing a normalized view of the data, independent of the underlying data formats. The SAFE supports composite Factories where multiple data-base tables can be queried through a single Factory object and a unified report can then be run combining data from all tables. The report has to be written in terms of the set of common standard properties but the implementation of these properties can be different for each table.

The above examples show the text representation of a property-expression. This is the form used in configuration files and report templates. Internally these expressions are parsed into an Abstract-Syntax-Tree Object representation. We use the Visitor Pattern[3] to implement operations on these objects. One important operation that needs to be implemented is to translate the expression into an SQL fragment. These fragments are represented by the *SQLValue* interface. A SQLValue provides methods for adding the fragment to a SQL statement and methods for extracting the resulting value from the Java ResultSet object returned by the query. Optionally a SQLValue can also provide a SQLFilter that modifies the FROM and WHERE parts of the SQL statement for example to add a required JOIN.

A special sub-interface of SQLValue is the *SQLExpression*. This is functionally equivalent to a SQLValue except that in a SQLExpression the value of the returned object is equivalent to the underlying SQL fragment so it is possible to combine SQLExpressions at the SQL level to make more complex expressions.

SQLExpressions are needed for SQL reduction operations like SUM, MIN or MAX. In other contexts a SQLValue is sufficient and the code may choose to simplify the SQL query by performing some of the processing in Java.

It is also possible to implement a property using a fragment of Java code that has no SQL equivalent however this forces the Factory class to implement queries involving these properties by iterating over individual records. This is only required in exceptional circumstances where the performance requirements are low and the property is easier to calculate in Java. For example the helpdesk component of the safe uses this feature when calculating properties that

depend on the number of elapsed working hours for a helpdesk request.

IV. DATA IMPORT

The SAFE uses a common mechanism to import external data of various kinds. Parsing and processing of data is handled by a series of plug-in modules. Each data table is configured with a parser and a series of policy plug-ins.

A. Parsers

The parser is responsible for breaking the input into a series of individual records and parsing each record into a collection of properties. Each parser usually generates a set of properties unique to that parser that faithfully reflects the data provided in the raw input. Parsers can also provide a set of default mappings (defined in terms of property-expressions) between these properties and the standard properties used in writing generic reports. In addition the parser specifies which of the properties it generates can be used to uniquely identify a record. These are used during the import process to identify if a data record already exists in the system.

B. Policies

Once the parser has performed the initial data-parse the collection of properties is passed through a series of policy objects. The purpose of a policy object is to generate additional properties based on local site policies rather than properties that are inherent in the data being parsed. Typical uses of a policy include:

- Generating a charged cost for a job.
- Generating a reference property pointing to a known user based on the username in the job-record.
- Cross-referencing related data from two different sources.

When the data being imported is usage data an extended form of policy can be specified that implements the Observer pattern[3] so that the policy object can be notified whenever a new record is added or removed. This allows additional side-effects such as decrementing/refunding budget allocations.

For systems with very large record counts, policies can be used to build aggregated records. Each individual record is still parsed into its own data-base record but an aggregation policy can be used to build an additional table where records with similar properties and similar time ranges are merged together into aggregated records. Reports can then be rapidly generated from these aggregated tables.

C. Database representation

The mapping between the properties generated by the parsers/policies and the values stored in the database is generated dynamically depending on the database fields available. A property is only stored if a corresponding database field is available. Even if not stored directly a property may still available in reports if defined as a property-expression over other properties that are available. Parsers and Policies provide a default set of database fields that are used to create the database table if it does not already exist, however these only represent a reasonable default. In normal operation the database schema can be modified to control the behavior of the system.

V. TIME MAPPING

The SAFE reporting model provides special support for handling data records that overlap the boundaries of a reporting period. When the reporting periods are very long compared to the duration of the record it is sufficient to use a single time property to determine if the record should be included in the report. However shorter reporting periods suffer from artificial variability as records are arbitrarily assigned to a single reporting period. This is a particular problem when generating charts that show the evolution of a property against time as each point on the graph represents a short period of time, quite possibly significantly shorter than the records being graphed (see Figure 1. for an example of this). Two distinct types of scaling are required to avoid this problem depending on the nature of the property being viewed.

If the property is a quantity that accumulates during the accounting record, for example *CPU-time* or *wall-clock-time*, then the value from a single record needs to be divided proportionally between each of the time-periods it overlaps. The contribution to a particular period therefore has to be weighted by the fraction of the record that overlaps with the period.

$$V_{period} = V_{record} \frac{l_{overlap}}{l_{record}}$$

We can calculate the value accumulated over the time period by summing the weighted values of records that overlap the period. This can be converted into an average rate by dividing by the length of the period.

If the property is a quantity that can be measured at a particular instance in time, for example *Number-of-processors* or *Memory-in-use*, then the value at any single point in time would be calculated by summing the values from all records that cross that point. A representative value for a time-period would be the time average across the period of this instantaneous value. This can be calculated by weighting the value from each record by the fraction of period that is overlapped by the record and summing over records.

$$V_{period} = V_{record} \frac{l_{overlap}}{l_{period}}$$

We can always generate an accumulated property from an instantaneous property by multiplying by the length of the record. The time average of the original property is the same as the average rate calculated from this property.

In order to perform these weightings it is necessary to specify two time-valued properties to denote the start and end of the record. The choice of which properties to use depends on the value being mapped, for example, if charting the number of queued jobs *submit-time* and *start-time* would be used.

VI. Report generation

The report generation system uses an XML based language to define the report to be generated. This XML template is processed using a series of steps. Each step is implemented using a XLST style-sheet with custom extension elements that implement queries on the database. XML queries that are expanded in one processing stage may be interpreted as input for later stages. Once all the query elements have been processed a final XLST transform converts the document to the desired document format. This has the advantage that the same report specification can be used to generate output in a variety of different formats including HTML, XML, PDF and spreadsheet formats. This abstraction also makes it easier to support new formats such as mobile applications.
The current version of the reporting language supports the following features:

A. Access control

Access control elements define which users are allowed to view a report. Access control can be applied to full reports or to individual report sections.

B. Parameters

Parameter elements define variables that control the report being generated, for example to select the reporting period. A parameter form is generated from these elements and presented to the user. The values the user selects are then inserted into the report document to control subsequent processing.

C. Filters

Filter elements are the syntax used to specify a set of records (equivalent to the FROM and WHERE clauses in SQL). Filters are defined in terms of properties. The reporting syntax allows records to be selected from multiple data sources (provided they all define the required properties) and will automatically merge the results from the different data sources.

D. Single value query

These elements expand to a single value that can be placed in-line in the body of text.

E. Tables and Charts

The reporting language supports high level syntax for the creation of tables and charts.

F. Formatting

The Format operation specifies a fragment of the report that is expanded multiple times, once for each record selected. This can be used to generate record listings in common interchange formats such as the OGF-UR XML format[4].

VII. Custom Analysis

The XML report generation engine is suitable for most general reports, though it is also possible to use the underlying code to perform more specialized analysis than is available in from the template interface.

A. Utilization vs. Job waiting

An example of such an analysis is shown in Figure 2. This was an exercise to look at the correlations between machine utilization and job waiting times. Normally such an analysis is conducted using the wait times of individual batch jobs. However as jobs of different sizes will tend to experience different wait times we wanted to use a metric which reflected the amount of work queued on the machine as a whole. In the figure each point on the graph represents a time average over a calendar week in order to remove any natural periodic variations over the course of the week. The X axis is the average utilization of the system (normalized by the overall size of the machine). The Y axis is the average amount of waiting work over the week (normalized by the maximum performance of the machine to give a time in hours). The data covers three different hardware generations of the Hector service. The graph shows a remarkable consistency over these three different generations of the service and appears to show a clear threshold of utilization above which job waiting increases significantly.

B. Project use profiles

One of the advantages of a unified management system is that job usage data can be combined with administrative data such as project allocation information. Figure 3. shows one such analysis. We calculated the fraction of overall allocation that each project had consumed at various points throughout the lifetime of the project. The graph shows percentiles from the data available at each point. Note that many of projects considered were still in progress so the number of points in each sample is higher at the lower end of the graph. This graph illustrates how a proportion of projects consume very little of their allocation until almost halfway through the project lifetime, and that a similar proportion (probably the same projects) fail to consume all of their allocated resources.

VIII. Conclusion

The report generation and accounting code in the SAFE is capable of efficiently handling the large data sets generated by HPC job-loads while still retaining sufficient flexibility to be re-usable as a more general report generation tool. In addition by integrating this code into a larger user

management application reports gain access to additional contextual information that improves the usefulness of the reports.

ACKNOWLEDGMENT

SAFE is maintained, developed, and by EPCC at the University of Edinburgh. Development of the accounting sub-system Grid-SAFE was funded by JISC.

REFERENCES

[1] JSR-000154 Java™ Servlet 2.5 Specification http://download.oracle.com/otndocs/jcp/servlet-2.5-mrel2-eval-oth-JSpec/

[2] Grid-SAFE http://gridsafe.sourceforge.net/

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley 1994, ISBN:0-201-63361-2

[4] GFD-R-P.098, OGF Usage Record Working Group, https://forge.gridforum.org/projects/ur-wg/

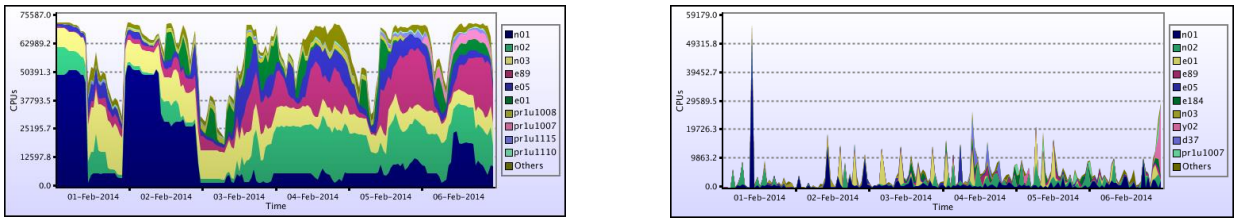

Figure 1. A comparison of CPU plots with and without overlap mapping. The graph on the left shows the full overlap calculation (time average of CPUs in use) where the graph on the right selects records that completed in the target period.

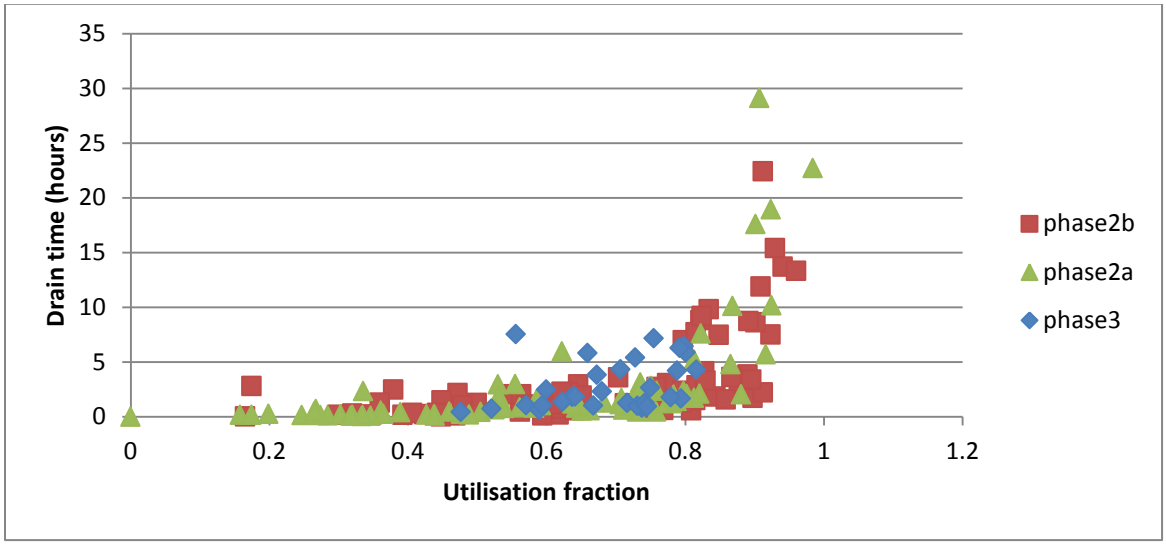

Figure 2. Analysis of the impact of machine utilisation on job waiting.

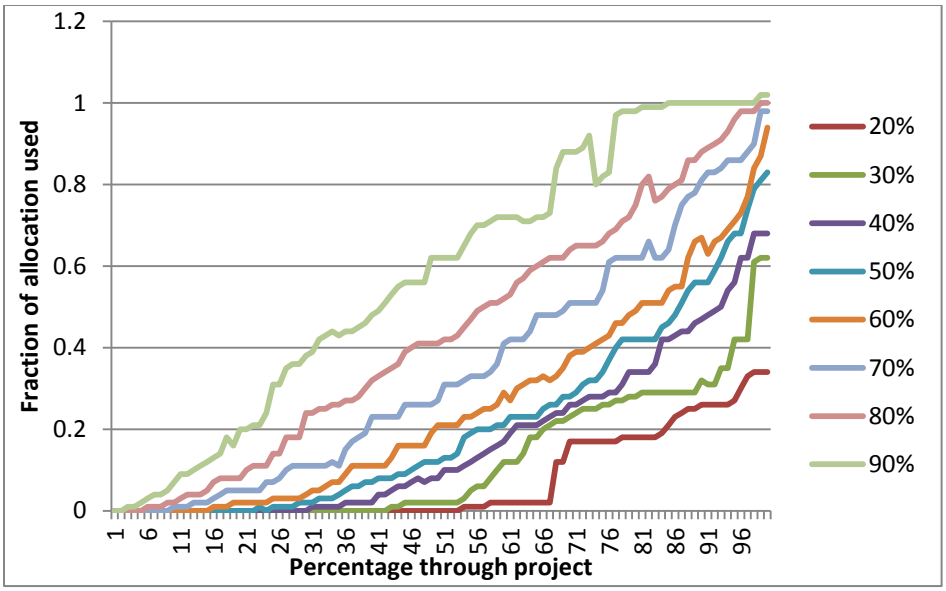

Figure 3. Analysis of job usage profiles