Relational model for information and monitoring

Steve Fisher, RAL

February 26, 2001

1 Introduction

The GGF has so far been treating monitoring and other information separately. However a lot of monitoring data will end up in an archive, and we need to ensure that the data can be efficiently queried when it gets there. It is also seems desirable to have a common interface to access data, whether it is *fresh* monitoring data or data from an archive. Consider the information:

> *The short batch queue on the CSF system at RAL has 34 jobs on it.*

To make this useful monitoring information it needs to have the time of the measurement recorded. This gives a tuple such as `(RAL, CSF, SHORT, 34, 2001/5/02:16:07)`. A set of such tuples could be stored in a table:

ComputingElementQueue

Site	Facility	Queue	Count	Time/Date

Any structured data can be represented in tables in this manner. The RDBMS community has been doing it for years.

The GGF performance architecture[5] does not specify the protocol between the consumer and producer. I would like to restrict this to be any protocol consistent with some chosen data model – that is one data model and any protocol consistent with that model. The chosen data model must have the power to represent all the queries we need to make. LDAP, in common with other hierarchical structures is fine *if you know all the queries in advance* as you can build your database to answer that question very rapidly. Unfortunately if you fail to anticipate the question getting an answer could be very expensive. The LDAP query language cannot give results based on computation on two different objects in the structure – or, expressed in relational language, there is no *join* operation. We can take as examples Usage Scenarios 14 and 15 collected by Ruth Aydt[1], for which it is very hard to see how LDAP could be adequate.

As an aside, it is interesting to note that the relational database was offered by Codd[2], 30 years ago, as the solution to the inflexibility of hierarchical and network data bases.

It is pleasing to see another paper, by Peter Dindas and Beth Plale[3] which is also being presented at GGF1, which has come to many of the same conclusions and should be studied to see more of the advantages of the relational (or tabular) model. They advocate an RDBMS to hold all the information and therefore

probably have no need to register producers. I prefer to use the relational model but to keep the performance architecture[5] to ensure good scaling, reliability and performance.

To be fair to LDAP, it does have one advantage in that there is a defined wire protocol. This does not exist for SQL. However we could adopt the solution used by MySQL[4] which allows remote data bases to be accessed.

2 Registration of producers

Following the GGF performance architecture, producers normally register themselves so that they can be found by consumers.

Now if the RAL CSF has a producer of ComputingElementQueue information it can register itself as `ComputingElementQueue(Site=RAL, Facility=CSF)`. This is saying that it has rows from the *ComputingElementQueue* for which two columns have fixed value: *Site* is `RAL` and *Facility* is `CSF`.

This information could be stored at the level of RAL by an RDBMS with both a consumer interface and a producer interface. The producer would register itself as `ComputingElementQueue(Site=RAL)`. If another RDBMS held information on all ComputingElementQueues this would register itself simply as `ComputingElementQueue`.

So the rule is simply to *register the name of the table and the names of any attributes which are fixed and the values of those attributes.* Details of the registration procedure are explained in section 6

3 Protocols and APIs

The model I suggest here makes some restrictions as it implies that the protocol must be suitable for transmitting relational data. What is the desired functionality of the protocol?

For a producer offering rows from a single table it is easy to make it process an SQL SELECT statement by either streaming rows which match the query or returning the latest row if it matches the query according to whether a single event or a stream of events is requested, easy. The SQL statement may of course only request certain columns of the table (or fields within a row if you prefer to think that way).

To bring the benefits of the relational model we want to be able to send queries which include joins and thereby select information from two or more tables. The result of a an SQL SELECT statement is normally that of a dynamically created table.

For example, we may be interested in finding a queue which is not too long and which has been up for some time. Now the UpTime could be added as an attribute to the ComputingElementTable however this would be inefficient as the queue length varies frequently but the time when the queue started changes perhaps every few months. So we need a second table:

ComputingElementStartTime

Site	Facility	Queue	Time/Date

It is now easy to formulate an SQL SELECT to return the (Site, Facility, Queue) which has been up for more than 24 hours and which currently (say within the last 10 minutes) has a queue length less than 10. This is done by joining the tables and projecting out the desired columns. Instead we could return the tuples from each table which contribute to the query. In this way we keep the communication simple as the only data which move around are simple known rows of a table.

To allow aggregate functions to be used (for example to compute an average) and to be able to reduce the traffic to just what is required, the full power of SQL would be needed. You transmit an SQL query and a dynamically constructed table comes back. If this data were to be made available via a producer interface a new table would have to be registered describing this dynamically constructed table.

It is clearly desirable to offer a consistent and simple API to all producers and consumers of Grid information. As has already been demonstrated, the LDAP model is not capable of addressing the more complex of the GGF monitoring use cases. Allowing the LDAP and the relational models to co-exist appears to bring no benefit in terms of expressive power and brings needless complexity. So what might the API look like in a pure relational solution? For the producer, for each table it produces it should register the table name and the identity and value of any fixed attributes. Then a producer simply has to announce a table name and the row(s) of a table.

For the consumer API you send a SQL query and get back rows of a table or request that rows of a table are streamed to you. The client can analyze the query and based on the tables involved send the query to the right producer or producers. Queries which can be processed by a single producer can be handled efficiently, but others will result in the some operations being carried out by the client side. This suggests that there will be *advantages in having Producer/Consumer/RDBMS units able to hold data which will often be joined.* In fact such a unit might be created automatically and then destroyed when it is no longer frequently used.

4 Time to live

If data are archived as suggested above how do we decide when to get rid of it. The information may no longer be up to date, but if we are interested in historical data this is of no consequence. This means that the source of data is no judge of its continued worth and so TTLs are useless. Only the collector of data, who knows why he is collecting the data can devise a suitable strategy.

5 Surrogate keys

Normally small integers are used as *surrogate keys* when designing data base schemas. A small integer is used as the primary key rather than some more natural string so that it can be referenced more compactly by other tables holding this integer. The allocation of these small integers would be difficult and it is suggested that this practice not be used.

6 Registration of producers and schemas

The schema information i.e. the table descriptions must be universally known. This is a problem for application monitoring data where the schema could be very short lived. One solution is to ensure that the registration of new tables is easy to do.

If there is more than one producer offering the same data what should happen? It could happen that two archives are set up to archive and offer the same data. Many events will be identical though not all because of different clean up strategies and because of losses where the producer is potentially faster than its consumers. To make this less likely the distinction between archives and producers only of *fresh* data should be noted in the producer directory. An archive would only connect to a source of fresh data and would not collect from another archive. Services would exist which perform computations upon data and produce new *fresh* derived data. Finally there will be pass-through consumer/producers these are needed by computer centres where their nodes are not visible from the outside. They must be able to pass on data they know nothing about: i.e. user monitoring data from the applications.

A possible strategy would be to only allow one source of information of a given type to register and so preventing any confusion. The registration system within a computer centre would not register the pass-throughs but these would be registered on the outside.

There will not be a very large number of producers in a Grid at any one time and each one requires only a very small amount of information to be stored when it registers. So a possible solution would be to have an RDBMS holding both the schema and the available producers. This should duplicate itself over a number of RDBMS around the world – all of which are trying to become identical. When you register, you use any one and the information spreads to all of them. When you want information you just use any one.

In the usual way we can have a table to describe the tables and one to describe the columns of the tables as indicated below, taking as an example the two tables shown earlier. TID (TableID) is acting as a surrogate key and so appears in the column table to show which columns appear in which tables[1]:

Table

TID	Name
1	ComputingElementQueue
2	ComputingElementStartTime

Column

CID	Name	TID
1	Site	1
2	Facility	1
3	Queue	1
4	Count	1
5	Time/Date	1
6	Site	2
7	Facility	2
8	Queue	2
9	Time/Date	2

The other two tables show the registration of producers, corresponding to the three examples shown in section 2. The *Archive?* field is `Y` for an archive and `N` for *fresh* data. The field *ValueAsString* has been given this name because it has to hold the value of a field of unknown type.

[1] This example also shows to those not used to the relational model how a simple hierarchy (a table with columns) is represented as a pair of tables

ProducerTable

PTID	TID	Archive?	URL
1	1	N	...
2	1	Y	...
3	1	Y	...

ProducerColumn

PCID	CID	ValueAsString	PTID
1	1	RAL	1
2	2	CSF	1
3	1	RAL	2

Notice that here we do use surrogate keys and so care must be taken to renumber when information is moved from one RDBMS to another. Relating the small integers is tedious for a human reading the tables, but integers take less space and are handy for indexing.

It would be useful to add columns to these tables to indicate when each record was added (at least for the *ProducerTable* table). The producers will periodically re-announce themselves and their record will be dropped from the tables when they are old if not refreshed. When a producer registers itself as a producer of a certain table, if the table is not known about it can be added to the schema. If a *Table* is not used by any *ProducerTable* its definition can be removed. This is convenient for schema evolution.

One problem which this will not solve is the case of two producers registering a table with the same name. Eventually the names will move around the system and will clash. In the same way if we wish to prevent a producer registering itself with the same information as an already registered producer this will not always work reliably. A solution to this problem would be that each copy of the schema/registry RDBMS knew about every other active one and so could synchronize important changes such as a new table definition.

7 Conclusions

It appears beneficial to support a data model which can support arbitrary queries. It seems practical to introduce the relational model without any major impact upon the GGF performance architecture. The mechanism for partitioning and managing a distributed RDBMS outlined in this paper seems practical. Plans are to start building prototypes soon to verify this.

Acknowledgements

It is a pleasure to thank Brian Tierney and Dan Gunter for providing valuable feedback. Manfred Oevers and James Magowan from IBM, two partners in the EU DataGrid project, have taken a special interest in the practicality of the proposal. Their input has been greatly appreciated.

References

[1] Ruth Aydt and Darcy Quesnel. Performance data usage scenarios. Technical report, GGF, 2000.

[2] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6), Jun 1970.

[3] Peter Dinda and Beth Plale. A unified relational approach to grid information services. Technical Report GWD-GIS-012-1, GGF, 2001.

[4] Mysql. http://www.mysql.com.

[5] Brian Tierney, Rich Wolski, Ruth Aydt, and Valerie Taylor. A grid monitoring service architecture. Technical report, GGF, 2000.