Requirements for Scalable Conditions Data Delivery

Fermilab Computing Division/Running Experiments (CD/REX)

November 15, 2010

DRAFT

Introduction

Conditions data is an umbrella term that refers to information that describes detector and beam conditions. It is generally valid for detector data taken during specific periods of time, sometimes referred to as Intervals Of Validity (IOV). It is a type of meta-data necessary to make sense of the detector data, and it includes calibration, alignment, attenuation, pedestal, etc. for detector channels, as well as information about the intensity and characteristics of the beam.

Some of this information is required for processing and analysis of detector data and thus access is required by many clients running simultaneously on interactive and GRID resources. Much of this data is stored in central databases or files, and approaches to scale the delivery to thousands of clients are needed.

Parameters

Following are parameters that define the problem. Typical values need to be obtained from experiments and/or estimated.

1. Expected request rate: Peak and Average
2. Data unit size
3. Latency requirements
4. Accepted failure rate
5. Some estimate of time correlation between requests
6. Number of clients
7. Location of clients

Also important are boundary conditions such as hardware to be used, network bandwidth available and which technologies to use and not to use.

MINERvA Example

Based on MINERvA jobs run over the last few months on the General Purpose GRID farm, some basic understanding of the requirements for the conditions database service can be examined. Table 1 includes these parameters.

Description	**Estimate**	**Comment**
Job Duration	1 hour	
Number of simultaneous Running Jobs	100	
Events per job	1000	
Internal job cache hit ratio	100%	Most events processed by a job use the same conditions data set
Size of conditions data set	3 MB (uncompressed, binary)	

Table 1. Conditions database access for MINERvA jobs.

With some additional assumptions, a clearer understanding of the scale of the delivery service can be obtained.

- Number of requests processed simultaneously without significant loss of scalability: 5
- Peak job start rate = 10 times average job start rate
- Allowed latency = job duration/10 = 6 min

Table 2 shows possible values for average and peak database server activity.

Description	**Average**	**Peak**	**Comments**
Job start rate	2/minute	20/minute	N running jobs/job duration
Conditions data requests rate	2/minute	20/minute	assuming 100% internal job cache hit ratio = job start rate
Network bandwidth	100KB/sec (3MB * 2/minute)	1MB/sec (3MB * 20/minute)	
Disk throughput	100KB/sec	1MB/sec	
Time to retrieve and deliver 1 data set to sustain the request rate	30 sec	3 sec	Single threaded DB server
	150 sec	15 sec	Five threaded DB server

Table 2. Average and peak database activities based on the assumptions in the text.

ID scheme and version control

There needs to be a mechanism by which the various kinds of conditions data can be managed as "sets" valid for a given time interval (A.K.A. IOV). Changes to the *conditions set* need a mechanism for "tagging" them so reproducibility in processing can be insured. Requests from a client must refer to the *set*, *IOV* and *tag*, or some similarly unique identification scheme.

In the simplest cases, experiments will request conditions data based on an event time, say the time for the first event in a file. For systems with intermediate caches, this will not work since physics data files with different starting events may request the same cached date be stored again.

Achieving Scalability and Reliability

Achieving the scalability and reliability to satisfy the requirements can be done by properly sizing a central database service, or providing additional caching tiers between the database server and the client. Having a single central service is the easiest as it preserves the original interfaces used by each experiment. In some cases scrutinizing the way each experiment stores and retrieves information to and from the database can also provide significant improvements in performance. However, caching layers can enable scaling to meet the performance and reliability needs of the experiment. Some approaches to adding these additional tiers include database replicas, distributing the data as files, or using proxy/caching servers located on the network.

Although reliance on a single central database may be simple, it is not always feasible or practical. In the environment of the Intensity Frontier there are many database technologies being used. Each experiment has chosen the solution that works best within their development environment and is supported by their framework software. Among the solutions being employed are Oracle, MySQL, PostgresSQL, SQLite and file-based options such as ROOT or ASCII formats. Simply scaling up the central service is limited by server hardware and network constraints.

By including additional caching layers, or tiers, lightweight components can be deployed to unload the demand on the central database service and provide additional reliability. This can be done close to where the clients are running and significantly improve throughput while maintaining low central server and network loads. These caching layers can be in the form of 1) database replicas, 2) files delivered to the processing site, or 3) some form of proxy/caching server deployed at or near processing centers.

Replicating some or all of the information in the central DB service can be an effective approach to scaling the system. Some database technologies provide replication software that can make this fairly straightforward. A read-only replica is practical to set up and support, and can in some cases satisfies the requirements.

For conditions data that is more-or-less static, distributing files with the detector data being processed is a satisfactory solution. If the conditions information is changing then delivering files can be problematic and requires a method of indexing them to ensure that the proper info is provided and used for specific processing. Exporting conditions data from the central database to SQLite files can provide a convenient method to capture the relational nature of the data in a transportable static file. Some of the frameworks used by experiments already support SQLite and it is therefore an attractive solution.

For conditions data that is changing http proxy/caching servers represent a good solution. This is typically done with SQUID and this has been shown to be reliable and versatile. If the client requests are done properly, and the SQUID cache can be used effectively, the performance achieved with modest hardware can be extremely high. Redundancy is also easy to design into the system so reliability can also be achieved. Most OSG GRID facilities have SQUID services already in place providing a standard infrastructure near the processing client.

In a cached system, the cache can be stale and the refresh policy must be understood. This issue is referred to as “cache coherency”. Several techniques have been developed to mitigate potential issues in this area. These need to be clearly understood and appropriately implemented for any type of caching system.

Monitoring

A desirable is a mechanism that allows monitoring and recording the requests that are being sent by the clients. This adds to the understanding of who, where, and what kinds of requests are being made. Having a middle tier reduces the need for some of this monitoring since the chance for bottlenecks is greatly reduced and any serious contention at the central service can be traced by looking at the access logs.

Short-term strategy

For the most part, to date each experiment has chosen a different approach to managing their conditions data information. Solutions like replicated databases may make it possible to scale to increasing numbers of clients under such an environment. Some frameworks, specifically GAUDI/COOL//CORAL, support multiple technologies including Oracle, SQLite, MySQL (deprecated in recent versions) and FroNTier. This provides some flexibility as to the choice of solution, but nevertheless requires effort to set up infrastructure and understand feature sets that are, sometimes, not well documented. Other frameworks, like FMWK, are more specific to particular solutions, PostgresSQL in this case. Other frameworks, like the CMS lite (A.K.A. ART), will need to be understood as they appear.

Long-term strategy

In the long term, providing common solutions would simplify deployment and support. Common API's would make documentation and user buy-in straightforward. In some cases shared repositories may be possible and features such as monitoring could also be uniform and shared. As we move ahead working closely with the development teams for the experiments and frameworks will enable us to include the up-front design needed to implement the best solutions.

Conclusion

Understanding the requirements for conditions data storage and access are important to providing proper solutions to the problem. From a preliminary analysis of typical access patterns, a basic understand of the conditions data delivery performance needs can be examined. Having a consistent scheme for identifying conditions data sets and IOVs, with proper version management, is essential. Reliability is also an important requirement for such a service. Several approaches for deploying a system to meet the performance and reliability requirements are possible.

In the short-term, we will work closely with the experiments to understand their existing systems and implement straightforward solutions to improve their DB access when possible. In the long-term, we would like to move toward common solutions across experiments that will be more easily monitored and maintained.