A MySQL-BASED DATA ARCHIVER: PRELIMINARY RESULTS*

M. H. Bickley#, C. J. Slominski, Jefferson Lab, Newport News, VA 23606 USA.

Abstract

Following an evaluation of the archival requirements of the Jefferson Laboratory accelerator's user community, a prototyping effort was executed to determine if an archiver based on MySQL had sufficient functionality to meet those requirements. This approach was chosen because an archiver based on a relational database enables the development effort to focus on data acquisition and management, letting the database take care of storage, indexing and data consistency. It was clear from the prototype effort that there were no performance impediments to successful implementation of a final system. With our performance concerns addressed, the lab undertook the design and development of an operational system. The system is in its operational testing phase now. This paper discusses the archiver system requirements, some of the design choices and their rationale, and presents the acquisition, storage and retrieval performance.

ARCHIVER REQUIREMENTS

Development of the MySQL-based archiver started with an evaluation of the user requirements for the system. In this context the users of the archiver include not just control system users, but the maintainers of the archiving system, computer scientists who write software to access historical data, and system administrators who must design and maintain the archiver's computer systems as well as back up the archived data. A cross-section of the lab's community of these archiver users met and documented a complete set of archiver requirements. [1]

Some of the requirements spelled out in the document merit special mention. In addition to the typical requirements that might be expected with any archiver (e.g. monitoring and recording all primitive data types and specification of data retrieval needs) there are some requirements that focus on the maintainability of the archiving system for a minimum projected lifetime of 10 years.

One requirement specifies that the archiving engine itself must be able to perform deadbanding on all scalar values. Deadbanding enables the archiver to ignore changes in value that are below some threshold specified by device experts. Because the archiver executes the deadbanding it is available for all control system scalars. EPICS only provides this facility for certain parameters, and provides no deadbanding ability for the remainder. Independent deadbanding is beneficial as well because the tolerances for archived data are not always the same as the tolerances required for real-time interaction with a control system.

Another requirement is that the archiving system must be able to maintain metadata information about each channel being monitored. A channel's metadata can be critical to interpreting the data's value. An example of this is the definition of the enumerations associated with an enumerated type. Without those (potentially changing) definitions, the meaning behind a particular value at a specific time may be lost.

Several of the archiver requirements focus on data handling issues. In order to manage the scope of data being collected, the archiver must provide facilities for controlling data retention relevancy and archive request lifetime. Data retention relevancy refers to a channel's storage time period after the data is collected. Rather than saving all data forever, this feature enables an archival system to eliminate data once it is no longer relevant. An example of this is the intermediate values in the computation associated with a control system device. The values might be only useful for post-mortem analysis of faulty device behavior. Therefore the data can be removed from storage after passage of the period during which analysis might be performed. In a similar vein, the archive request lifetime aids in management of the volume of data stored in the archive system. Because all archive requests are given a lifetime (which may be infinite for "core" operational needs), the rate of data collection will not grow monotonically.

Finally, there are a series of requirements that help ensure both scalability of the archiving system and graceful handling of some most-likely failure modes.

The archiving system, a "logical archiver," must be able to scale over time, as the demands on the system increase. This can be addressed by ensuring the logical archiver is comprised of one or more archiver instances. Each instance must operate independently, except for a master instance that has the additional responsibility of tracking which channels are associated with each instance. Scaling the system up will require only that an additional instance be added to the logical archiver. Figure 1 shows a schematic of a logical archiver.

The archiver has to gracefully handle the two failures most likely to afflict the system. First, it must deal with disk overflow. There may be operational periods during which data is accumulated at a rate higher than expected, resulting in the system's hard disks filling before the problem can be handled by administrators. With the assumption that old data is not as valuable as recent data, the archiver should discard the oldest online channel data in favor of the most recent data. The archiver's associated backup system may be used to recover the discarded data if needed.

Bickley@jlab.org

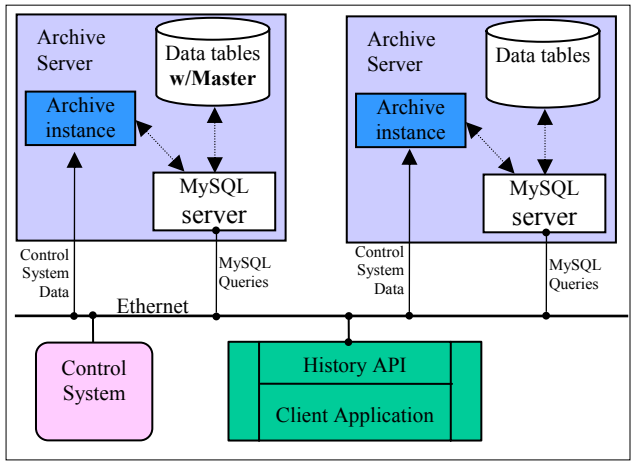

Figure 1: A logical archiver

The archiver is also required to handle failures where the CPU is overloaded. An overload for a sustained time will eventually result in data loss, but the system should be designed to prioritize functionality appropriately. In this context the lowest priority function for an archiver instance is support for data history requests, with the next-lowest being data storage. The highest priority of the archiver is obtaining and queuing control system data, so support for control system communication and buffering of data should receive all of the CPU time it needs.

PROTOTYPING STUDY

The Jefferson Lab prototyping effort started with the premise that we should take advantage of the expertise of other computer scientists and software engineers as much as possible. The developers of relational databases (RDBs) spend a great deal of time providing the functionality and performance required of their users. Taking advantage of their expertise enabled us to focus our efforts on the issues associated with control system interaction, process tuning and performance management.

We started the study planning to base the data store for the prototype on the widely-used open-source relational database MySQL. MySQL was attractive because the lab has several staff members with years of experience in using and tuning MySQL databases. With their advice we believed we would be able to determine the upper limit on MySQL's capabilities for this application. We also considered using the relational database system available from Oracle because it is used for many solutions at Jefferson Lab, and its ability to host archiving services has been demonstrated at other sites [2]. If MySQL proved incapable of meeting our performance needs, evaluating Oracle in its place was planned to be our next step.

Prototyping Design

The prototype archiver supported two different database table designs [3]. One was named a "few-table" design, with one table for each fundamental data type. The second design was called the "many-table" design, with a different database table for each channel. The data storage requirements of the few-table design are larger than the many-table design for two reasons. First, the few-table design has an extra field (specifying a channel identifier) that is not required in the many-table design. Second, the few-table design has an index that is much larger because it uses a two-column key (channel identifier and time) versus the single column key (time) for the many-table design.

The prototyping software was a simple multithreaded application. It had a main thread, used for overall control purposes, a communication thread which interacted with the control system, and a variable number of database threads, each of which supported its own connection to MySQL and was responsible for database insertions. This design lent itself to the primary purposes of the prototyping effort, evaluating the performance of MySQL and testing different database designs.

Issues Addressed During the Study

The prototyping effort was executed on a Pentium 4-based computer system with 1 GB of memory running RedHat Enterprise Linux version 4. One of the first tests was to see how the system performed when executed with real-time priority scheduling, rather than the operating system's default round-robin scheduling. For this test there was no interaction with the control system and no insertion of data into MySQL. The test provided a measure of the rate at which data could be pushed into the work queues of each of the database threads. With real-time threads, the system handled 2,000,000 events per second, versus 80,000 when using round-robin scheduling. It was clear that using real-time scheduling was vital to maximize the capability of the archiver on a general-purpose workstation.

A collection of tests were used to evaluate the performance improvement associated with a variety of database and programming choices. One test used MySQL *prepared statements*, precompiled database requests that execute faster than database requests which have to be parsed with each execution. Their use resulted in a 33% increase in throughput. Another test exercised the *insert delayed* feature of MySQL, which permits the database to buffer table insertions. This feature provided another 33% improvement. Attempting to perform two insertions for each database access provided a 59% increase in throughput, but resulted in unbounded buffering delays and in slower data fetches since associated time stamps would not necessarily increase monotonically. The study allowed us to examine the potential performance gains that could be derived from the use of a specialized free-list management library rather than the operating system's memory management functions. This provided an improvement of nearly 25%. The drawback to a scheme like this is that it requires the allocation of a large block of memory that is then unavailable for use in the caching of file system I/O.

Prototyping Results

The prototype study provided some clear results to guide continued development of a MySQL-based archiver. For example, the optimal number of database threads was in the range of 5 to 10. Fewer than 5 resulted in excessively long work queues for the threads, and the overhead associated with more than 10 threads became burdensome to the operating system. The few-table design enabled data insertion rates that were more than twice the rate associated with the many-table design. On the other hand, data retrieval rates were two orders of magnitude slower with the few-table design. Another drawback of the few-table design was that the associated file sizes were so large that their management might prove to be problematic in an operational system.

It was clear at the completion of the study that it would be possible to meet the Jefferson Lab archiving requirements using a MySQL database as the data store. A system based on a design like the many-table prototype would be able to meet the lab's immediate needs, and would require an estimated three Intel dual-core systems in order to meet the lab's needs.

DETAILED DESIGN

With the viability of a MySQL-based archiver established, we developed a detailed design [4]. Named MYA (for MySQL Archiver) the product of this design incorporated much of what was learned from the prototype study.

The process of developing the detailed design included additional testing and analysis as MYA's hardware and software needs were clarified. One component of this phase of analysis included evaluation of the efficacy of multiple processors on the archiving engine. This turned out to be a very fruitful area of performance improvement. The multi-threaded nature of the engine, with independent threads managing data insertion for their own database tables, led to significant performance boosts on systems with both multiple processors as well as multiple cores in each processor. Given sufficient I/O throughput that the performance bottleneck of the system was CPU loading, scaling the number of processors produced a near-linear improvement in performance. Multiprocessing also made it feasible to increase the number of database threads, enabling the system to handle more data without increasing the workload per thread. The benefits of hosting MYA on a multiprocessor system, on top of the database optimizations, made it realistic to meet the archiving needs of the Jefferson Lab accelerator using a single system, rather than requiring three as were estimated in the prototyping phase.

An area that was carefully examined during the detailed design was the latency of archived data. This is the time delay between the time of a channel value change and when it is available from the archiver. The length of the delay is driven by the queue size, the amount of data queued up for insertion into the database. As MYA gets more heavily loaded (by bursts of data from the control system, for example, or a large volume of requests for history data) the queue size, and therefore data latency, increases. Supporting the work load expected from the Jefferson Lab control system, the maximum latency measured was 23 seconds. On average, however, the latency should be well under 1 second.

One feature of MYA bears special mention. In order to maximize the volume of data stored by the system it uses the 32-bit UNIX time. These values are only valid until January 19, 2038. If the system continues to be in use as that date approaches, a significant maintenance effort will be required in order to ensure that the software remains usable and the data accessible.

CONCLUSION

Jefferson Lab's operational experience with this archiver has proven very positive. We purchased a robust machine to host the archiver. The system hardware consists of a Dell PowerEdge 2850 rack-mounted computer, an EonStore RAID disk enclosure and 16 300GB SCSI disk drives that provide a total of 2 TB of storage. The computer has dual quad-core CPUs, to take maximum advantage of the threaded design of the archiver. It has 16 GB of memory, ensuring that all data structures can reside in memory and still retain a large amount of memory for caching of disk I/O. The disk drives on which the data is stored use RAID 0+1, so that all of the data is striped (improving performance) and mirrored (to provide some protection against disk failures). Configured like this, a single system has been sufficient to support all of the operational archive needs of the Jefferson Laboratory accelerator for the last 8 months. It can process channel updates at a rate of more than 50,000 per second, deadband the data and perform more than 30,000 database insertions per second while simultaneously exporting data events to history data clients at rates up to 200,000 events per second.

MYA has been running in parallel with the lab's existing archiver for more than 8 months. This has enabled us to work on developing improved client tools while we verify MYA's performance and capabilities. Running the archiver in parallel also enables us to accumulate a large set of data that will be available to users once MYA becomes the operational archiver for Jefferson Laboratory.

REFERENCES

[1] C. Slominski "EPICS Channel Archive Facility Software Requirements Specification" JLAB-TN-07-063

[2] M. Clausen et al. "COSMIC – The SLAC COntrol System MIgration Challenge" Proc. ICALEPCS 2001, San Jose

[3] C. Slominski "Archiving Directly into a Database" JLAB-TN-07-065

[4] C. Slominski "EPICS Channel Archive Facility: Design Specification" JLAB-TN-07-064