Archiving Directly into a Database
By Christopher Slominski
January 5, 2006

BACKGROUND ........................................................................................................................................... 2
INTRODUCTION ........................................................................................................................................ 2
DATABASE DESIGN ................................................................................................................................. 3
THE TEST SOFTWARE ........................................................................................................................... 3
PRELIMINARY RESULTS ......................................................................................................................... 4
RESULTS ...................................................................................................................................................... 6
  FEW-TABLE DESIGN ............................................................................................................................ 7
  MANY-TABLE DESIGN ........................................................................................................................ 7
THINGS TO TRY ........................................................................................................................................ 8
CONCLUSION ............................................................................................................................................. 8
APPENDIX-A; A FEW-TABLE TEST RUN ............................................................................................ 9
APPENDIX-B; CZAR 2-0 DATA COLLECTION PROFILE ..................................................................... 10
Background

The current Channel Access Zippy Archiver (CZAR 1-2) was developed at Jefferson Lab for the purpose of archiving the state of the EPICS control system used at the laboratory. A significant amount of development resources were expended in this effort, however the product has proven to be brittle and buggy, requiring significant attention from the software group staff. The software group has created an update, CZAR 2-0, which fixes a number of the bugs and provides a much needed backup strategy. Unfortunately, the CZAR 2-0 software suffers from a significant design compromise that may prove unacceptable in the operations environment. The delay between the occurrence of a control system event and the corresponding data becoming available to CZAR users is as much as fifteen minutes\(^1\). This shortcoming along, with a long outstanding list of CZAR 2-0 bug reports\(^2\) prompted a study of alternate archiving options.

Introduction

One option to resolve the EPICS data archiving issues at the Lab is to design an entirely new archiver. Although this sounds audacious, given the time and expense already invested in the CZAR software, it still may be a legitimate option considering the amount of effort that may be required to fix CZAR 2-0.

The study described in this report focuses on a quick fix to the archiving dilemma. The strategy is to minimize the development effort and thereby minimize the faulty nature associated with a large new software product. The idea is to utilize existing reliable software to do much of the work, and spend money on commercial hardware and software as opposed to staff hours for development and maintenance.

I studied the feasibility of using a MySQL database as the repository of all control system archives. MySQL was chosen because of its availability and reputation for speed, however the concept could be applied to any reliable database product. Using a relational database off-loads much of the sophistication of archiving to well-tested production software. The database system is already optimized for high-speed file I/O, handles concurrency issues, and provides the structure for fast archive data retrieval. When leveraging off of a database system, the archiver development effort is fairly straightforward and does not require a large code development effort. The crux of the study was to demonstrate that the data archival and subsequent retrieval could be performed fast enough to satisfy the Jefferson Lab needs.

The CZAR 1-2 system connects to approximately 37,000 EPICS channels and receives Channel Access event notifications at a rate of about 9,000 per second, although this

\(^1\) CZAR 2-0 study by Anthony Bavuso (atb@jlab.org).
\(^2\) [http://devlnxsrv:8080/secure/BrowseProject.jspa](http://devlnxsrv:8080/secure/BrowseProject.jspa) (see project czar).
number has not been properly verified\textsuperscript{3}. The CZAR 1-2 storage scheme is efficient, having a cost of about 8 bytes per data point\textsuperscript{4}.

**Database Design**

There were two different database designs considered in the study. The first design has one unique database table for each EPICS channel being archived, and is referred to throughout this document as the many-table design. The second design has one table defined for each EPICS data type (eight) and is called the few-table design. Both designs utilized fixed length rows for efficiency. Each design also defines a directory table that associates EPICS channel names with support information.

The EPICS timestamp is 64 bits of offset from the EPICS epoch. The database timestamp information differs from what is received from EPICS to require less storage space. The EPICS format reserves 32 bits of fractional-second, which is both unavailable and unnecessary in the Lab computing environment. Just eight bits is reserved in the database for the fractional-second, giving a resolution of about 0.004 seconds, which is plenty for the archiver timestamp needs. This reduces the footprint of the timestamp from 64 to 40 bits. Space saving are also realized in the database by storing EPICS double data as float. I expect that quantities in the control system are adequately described with 24 bits of resolution and a $10^{39}$ range. The following are examples of table formats for the two database designs. Note that the data type of the value field will differ for tables storing differing data types. The difference between the two definitions is the extra field (chan\_id) needed when more than one EPICS channel can be stored in a table. Also the multiple channel tables have a two-column index key, which leads to a much larger index file.

```sql
CREATE TABLE table_4329 ( 
    sec int NOT NULL default '0',
    frac tinyint unsigned NOT NULL default '0',
    value float default NULL,
    KEY (sec)
) ENGINE=MyISAM

CREATE TABLE DOUBLE_DATA ( 
    sec int NOT NULL default '0',
    frac tinyint unsigned NOT NULL default '0',
    chan_id smallint unsigned NOT NULL default '0',
    value float default NULL,
    KEY (chan_id,sec)
) ENGINE=MyISAM
```

**The Test Software**

All of the software tests performed in this study were made on Red Hat Enterprise Linux (4ws) systems. In particular the two platforms oparsrv1 and devlnx05 were used, with the

\textsuperscript{3} CZAR appears to store data at about 5,000 points per second, however monitor fire rates have been observed between 8,000 and 11,000 per second. Also see Appendix-B.

\textsuperscript{4} Appendix-B details this finding.
former being considered the primary test machine. Any comments will refer to oparsrv1 unless specifically stated otherwise. The MySQL server used was version 4.1.12. The programs were compiled with the gcc 3.4.3 with optimization level three (-O3). Special versions of my certified libraries were created with the high optimization level, and to add special features utilized by the test software.

The software I developed for this study is stored in my home directory at /usr/user1/cjs/wrk/prg/test/caa/store. There are two folders within this directory, named many and few, containing the software associated with the two different database designs being evaluated. Each package has an archiver program (store), a program used to control the archiver (cmdr), an API for data retrieval in a sub-directory named fetch, and various helper utilities pertinent to the specific database strategy stored in the sub-directory tables.

The basic software design uses POSIX threads to maximize utilization of hardware resources. The following diagram shows the various threads created with the archiver program.

The main thread simply sleeps on a UNIX named FIFO waiting for user commands. These commands are for shutting down and for obtaining status information from the EPICS thread. The EPICS thread monitors Channel Access and inserts a work request into a queue for each Channel Access monitor event. There are a variable number of Database threads, defined on the program command line. Each of these threads has their own connection to the MySQL database. They sleep on the work queue, waiting for channel events to process and store in the database.

Preliminary Results

A number of tests were performed to provide insight into various techniques that could be utilized by one or both archiver database designs. These had to do with programming techniques, database tuning, and operating system configuration.

Running at real-time priorities has shown to be very effective. The skeleton of the design, with the worker threads not updating a database and no real Channel Access events, was used to measure the number of items that could be pushed into the queue and handled by
the workers (10). These tests were run on devlnx05. Standard round-robin scheduling yielded a limit of 80,000 events per second, while the use of real-time threads operated at 2,000,000 events per second. Clearly the use of real-time priority scheduling is of great benefit to the archiver when running on a general purpose workstation.

Using MySQL *Prepared Statements* showed a 33% increase in throughput when writing one table with three threads on devlnx05. The limitation on prepared statements is the table and column names can't be one of the variables, and the prepared statements are only valid for a single connection. Therefore each Database thread needs its own set of prepared statements. This technique only benefits the few-table database design, as it seems impractical to create 10's of thousands of prepared statements in each worker thread. This effect appears to be a major reason for the high write speed of the few-table design.

Using the MySQL *insert delayed* feature provided another 33% increase when also using more threads (7) on devlnx05. Allowing the MySQL server to buffer table insertions appears to be a definite benefit with no ill effects. My Database threads don’t really care that the data has actually been written into a table before returning to the work queue. This technique benefits both database designs, and was used in the final oparsrv1 tests.

MySQL allows multiple insertions into a single table with one SQL statement. Making the Database threads buffer work requests and send two updates at a time gave a 59% increase in throughput. This technique could not really be used with either database design being studied because worker threads don’t necessarily get consecutive work requests targeted to the same table. The buffering also causes unbounded delays in database updates in the rare situation where there are almost no Channel Access events taking place. Also, fetches from the table will likely be slower since the rows won't naturally take on an increasing time stamp ordering.

Running preliminary single table test scenarios on oparsrv1 performed about three times faster than on devlnx05, however enabling disk stripping on oparsrv1 provided no measurable effect. I expected the stripping to be of great benefit and do not know why it was not. This should probably be pursued further. Enabling of hyper-threading did show some performance improvement, however odd behavior was observed in this scenario and hyper-threading was quickly disabled. The operating system updates of database file access times were disabled by mounting the RAID array partition with *--o noatime*. No measurements were made on the effect of this strategy.

Setting the MySQL parameters *table_cache* and *open_files_limit* to 30,000 and 65,535 respectively benefited the many-table design significantly because of file caching and reduction in file opening and closing. I do not however have a good feel for how to tune MySQL in the many-table design. Although I read the book *MySQL Database Design and Tuning* (Robert D. Schneider) during this study, I did not glean any high-impact tuning information as I had hoped. Instead, I mostly obtained confirmation on the things I had been tweaking.

The software designs rely on the C++ new/delete features for allocating and de-allocating memory for the work requests. The general purpose heap management is slower than
specialized management alternatives, so I wrote a free-list memory management library
that pre-allocates fixed size blocks and provides them through a simple API. This proved
to be beneficial in the single thread scenario where no mutual exclusion locking was
required, however it provided little benefit in a multi-threaded usage pattern. I suspect the
POSIX locking interface provided to programmers is probably slower than the low level
services used by the built-in memory manager. Since the proposed archiver is a multi-
threaded design, the free-list software would have little impact. The benefit did not seem
worth the cost of reserving a large chunk of memory ahead of time, that could not then be
used for file system caching. The results of my free-list comparison, run on devlnx05, are
shown in the following table. Ten-thousand allocations of struct Item { double x, y, z;
int i, j, k; } were timed. All times are in micro-seconds and represent all 10,000
allocations and de-allocations. Note that the Free-List allocation is performed before the
timer starts, and is of sufficient size as to not need to reallocate. These tests were
performed with real-time thread priority to minimize interference from other OS
activities.

<table>
<thead>
<tr>
<th></th>
<th>New/delete</th>
<th>Free-list</th>
<th>Free-list w/locking</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No compiler optimizations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,150</td>
<td>4,166</td>
<td>464</td>
<td>3,198</td>
</tr>
<tr>
<td></td>
<td>Optimization -O3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,166</td>
<td>97</td>
<td></td>
<td>3,074</td>
</tr>
</tbody>
</table>

The C++ stringstream classes provide a sound capability for formatting text strings from
a set of variables, and is a preferable technique in most circumstances to using the ‘C’
style sprintf mechanism. There is a cost to using the type-safe and memory-safe C++
mechanism, even when you initialize the stringstream object to a large enough size so
that memory reallocations do not occur. I suspect that the calls generated by the insertion
operator form some overhead, and the fact the result is returned as a std::string object
must have some penalty. I timed string formatting of a typical SQL insertion statement
used in my archiver program, in the many-table design scenario where MySQL Prepared
Statements can’t be used. The statement being generated was insert delayed into <table>
values (<sec>,<frac>,<value>). The four variables in the statement are indicated by
angle brackets. This test was performed on devlnx05 and the times were measured in
micro-seconds. The ‘C’ style formatting took 2.8 usec per string format operation, while
the C++ style took 4.9 usec. I did not take advantage of this technique in the final test
results described below, but it may be worthwhile if squeezing performance is needed for
a new archiver.

Results

The benchmarking of the two archiver database designs was performed on oparsrv1, with
devlnx05 serving as the remote client accessing the archived data. There were two
sources of EPICS channels. My ioctl9 was used to generate channel updates in a
controlled pattern, as was done previously for the CZAR 1-2 data loss testing. The

5 Does CZAR Loose History Data?; September 21, 2005.
accelerator control system was also used as a source of EPICS data, specifically the same channels that CZAR 1-2 connects. Note that the ioct9 generated data was used more because of the repeatability and tighter control of the data source. The number of Database worker threads had an impact on results, although more than a few quickly became over burdensome to the system. Differing scenarios created throughout the testing period responded best to a differing number of threads. A good choice is five threads for either design, with ten being an upper limit. More research into the number of threads should be performed if an archiver is to be developed.

**Few-Table Design**

The few-tables design had very good write performance. I could store about 19,000 channel updates per second into the database tables. Most of my testing was actually done with just two EPICS data types being generated by ioct9; double and long, so there were just two tables being updated. The most significant problem with the design was history fetches, made once the data set covered a day or so, took a large amount of time. A test run with 10,000 longs and 10,000 doubles per second was made for one day. A single channel fetch of a day’s worth of data took about two minutes. The reason for this is that a channel’s data is spread out throughout one very large file, which contains data for many thousands of channels. The MySQL server has to read very many blocks of the file to extract the scattered pieces of one channel. I performed a very time consuming SQL command to re-order the rows on one of these supper large tables so that rows would be ordered by channel, not by the way they were initially received. This had huge impact and fetching a day’s worth of data for a channel was instantaneous. Unfortunately this is an after the fact technique, and is very time consuming. A table with 693,930,000 rows (10,000 channels/second times 69,393 seconds) took 3:24:56 to re-order, then 47:44 when repeated on the already sorted table). For some reason, reordering the rows made the index shrink by 17% which is a nice unexpected benefit. Interestingly, using the MySQL index packing operation on the table actually made the index grow by 1%.

Another problem with the few-tables design is the sheer size of the files being created as well as the amount of disk space required to store the data. Refer to Appendix-A for a description of one test run with the few-table design.

**Many-Table Design**

This design provided blazing fast history fetches, but with marginal write performance. As stated previously, the MySQL tuning parameters open_files_limit, and table_cache were set at 65,535 and 30,000 respectively. This gave much better results than the previous many-table attempts, but results seemed to vary much more than other scenarios. I don’t believe proper database and operating system tuning was realized for the many-table design.

I could achieve a sustained update rate of about 4,500 channel events per second in this design scenario, and this was done with both ioct9 generated data and actual accelerator control system data.

This design also suffers from storage size requirements, but not as much as the few-tables design. In a run similar to that described in Appendix-A, the disk size requirement was 18.33 bytes per data point, giving a history set span estimate of 138.8 days. This design
does not suffer from maximum file size limitations, as the default 4-byte row pointer
would allow for 13.6 years of data collection with channels updating at ten times per
second.

**Things To Try**

There are still a number of things to try to attempt to improve on performance. The
following summarizes various activities that might be worthwhile to study if a new
archiver development effort is undertaken.

- Use a 64-bit architecture machine. The MySQL database uses a lot of 64-bit
  variables internally.
- Further investigate disk stripping.
- Use symbolic links to split databases files to different physical disks, such as
  separating a table’s data from its index. Disk stripping would lessen the impact of
  this technique.
- Compile your own mysqld, optimized for performance on a specific host. There is
  a special x-86 version of the gcc compiler (pgcc).
- Play with various INSERT DELAYED parameters (%DELAYED%).
- Disable unnecessary Linux services.
- Use memory locking.
- Use more CPUs. Both the MySQL server and the archiver application are heavily
  threaded.
- Try a Sun workstation.
- Try an embedded archiver, running with a real-time operating system.
- Use the Free-list to allocate memory, as cited above.
- Don’t use `ostringstream`, as cited above.

**Conclusion**

Archiving EPICS data from the Jefferson Lab control system directly to a MySQL
database is feasible, but the study failed to prove that it can be done with a single Linux
X-86 system (having 2 CPUs). Neither of the designs studied could make the grade, but
some compromise between the many-table and few-table designs may achieve the goal.
The tens of thousands of database files involved in the many-table design clearly causes
some form of thrashing within the file system, and the huge files generated in the few-
table design introduces a different set of problems. Perhaps a hundreds of tables design
would be ideal. A compromise between the extremes would however complicate the
design somewhat.

I believe the many-table design has numerous advantages for an archiver. Not only do
you get extremely fast history fetches, but history set maintenance benefits as well. Table
locking because of maintenance activities will have less impact with the fine granularity
gained from the many-table design. The speed of culling old history from the history set will be maximized. The write performance issues with this design can be mitigated by deploying a distributed archiver. The design, though deployed for one machine in this study, was made with this in mind. The directory table that catalogues each channel would also have a column for the machine that archives the data. Each machine would have its own database server and disk storage space. One machine’s database would also have the directory table. This design scales well, though incurs the expense of buying additional hardware. I recommend a three machine archiver to support the many-table archiver described in this study. The amount of disk space required for each would depend on the span of channel history we wish to keep online.

Appendix-A; A few-table test run

The program store was run on oparsrv1 with 20,000 channels, and an update rate of once per second per channel. Ten worker threads were created. Only long and double data types were being collected, 10,000 per second of each, so the database had two data collection tables. The test began at 12:46:17 on December 14 and concluded at 13:35:58. The size of the work queue within the program was sampled every thirty seconds to determine if the worker threads could keep up with the load. It would spike occasionally to a few hundred due to mystery OS activity, and would spike to 200,000 or more when length history fetches were made. After the fetches the queue would quickly reduce in size to near zero.

The total run duration was 49:41 and at a rate of 20,000 updates per second, I expected to have approximately 59,620,000 data points collected. The tables each had 29,819,273 rows for a total of 59,638,546 rows. The extra 18,546 data points fall within the one second resolution of my experiment control.

Each row of collected data represents either an EPICS long or double data type. There are 9 bytes of raw data for each data point (binary data and 5-byte timestamp). Each row of data also contains a 2-byte foreign key into an EPICS channel directory table, as well as one byte created by MySQL. Therefore table rows actually consume 12 bytes. The table indices defined to speed up access are a large overhead in storing archive data. The table below depicts the disk space used for the long data type. Note the average disk space cost was 24.26 bytes per data point.

<table>
<thead>
<tr>
<th>Type</th>
<th>Expected size</th>
<th>MYD file size</th>
<th>MYI file size</th>
<th>Total file size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long</td>
<td>357,831,276</td>
<td>357,831,276</td>
<td>365,521,920</td>
<td>723,353,196</td>
</tr>
</tbody>
</table>

The size of the oparsrv1 partition is 1.8 TB with the RAID-5 configuration. The test used 1.347 GB of disk space, which corresponds to a usage rate of 0.4628 MB/sec. This means I can only archive for 47.2 days at the rate of 10,000 longs and 10,000 doubles per second. Currently CZAR receives about 9,000 updates per second, with the distribution of data types unknown. Assuming the same average memory consumption as with the test data, this provides a history span estimate of 104.9 days. This does not take into consideration periods of low control system activity, but is shorter than desirable.
The data collection tables as created have 4-byte row pointers, meaning that the table is limited to $2^{32}$ rows. At 10,000 rows per second these tables would be at their limit in 4.97 days. This is of course unacceptable and a 5-byte data pointer size must be selected to give a 3.48 year limit. The impact of the larger data pointer on write and read speed of rows has not been measured.

**Appendix-B; CZAR 2-0 Data Collection Profile**

This information was obtained after CZAR 2-0 was run for about 1.5 hours. The history set was cleared before the test began. Queries to the database and file system were made to analyze the disk requirements of CZAR. The standard set of EPICS channels that are archived were used in this test. The CZAR 1-2 storage requirements are identical to the CZAR 2-0 requirements. The data set size computation does not take into account the connection information stored in the CZAR database, which has insignificant impact in the overall result.

```sql
mysql> select sum(num_points) from data_directory;
+-----------------+
| sum(num_points) |
+-----------------+
| 22285894 |  
+-----------------+
1 row in set (0.17 sec)

mysql> select min(begin_ts) from data_directory;
+---------------------+
| min(begin_ts)       |
+---------------------+
| 1121327981132476160 |  
+---------------------+
1 row in set (0.01 sec)

mysql> select max(end_ts) from data_directory;
+---------------------+
| max(end_ts)         |
+---------------------+
| 1134680731127585792 |  
+---------------------+
1 row in set (2.05 sec)

mysql> select count(*) from data_directory group by file_id;
+----------+
| count(*) |
+----------+
| 27028 |  
| 16341 |  
| 16285 |  
| 15512 |  
| 14305 |  
| 13902 |  
| 13913 |  
| 16135 |  
+----------+
8 rows in set (0.11 sec)

mysql> select count(distinct stream_id) from data_directory
    -> group by file_id;
+---------------------------+
| count(distinct stream_id) |
+---------------------------+
| 27028 |  
+---------------------------+  
```
mysql> select sum(num_bytes) from data_directory;

<table>
<thead>
<tr>
<th>sum(num_bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>168425443</td>
</tr>
</tbody>
</table>

1 row in set (0.03 sec)

devsys05 cjs/wrk % czartime 1121327981132476160 1134680731127585792

Thu Jul 14 03:59:41 2005

Thu Dec 15 16:05:31 2005

Duration = 1:21:09 = 4,896 seconds
Points = 22,285,894
Rate = Points / Duration = 4,577 points/second
Size = 168425443(store) + 4669735384(MYD) + 58286083072(MYI) = 178,927,242
Cost = size / points = 8.03 bytes/point