1 Introduction

This document describes the design of an EPICS channel archiver. The design specifically addresses the requirements document created by the accelerator software group at Jefferson Lab\textsuperscript{i}. The document describes the general overall system structure and provides item by item accountability of how it meets each of the SRS requirements.

Prototyping of many design components has been performed to substantiate claims made about this design. Benchmark measurements will be provided where appropriate, along with a description of the environment used to obtain the measurements. In some cases an extrapolation of the benchmark data is required, as it was not always practical to emulate the full archiving scenario. Many of the design decisions were arrived at only after serious study and experimentation. A full report of these studies is available upon request\textsuperscript{ii}.
2 Overview

This archiver design relies heavily on existing software products. It does so to leverage off the expertise of other organizations, take advantage of proven tools, and to minimize the duration of the development cycle. At the heart of the archiver is a relational database. The database provides efficient file management, data structure, mutual exclusion among competing concurrent writers and readers, and standardization of data handling.

The MySQL database, version 4.1, was selected for this design. The rationale behind the choice was the excellent reputation of MySQL with regard to performance, its wide spread use throughout the Information Technology industry, and its cost effectiveness. As will be noted later in this document, the design requires a number of independent database servers running on separate host computers. Cost of a database like Oracle, which is licensed per CPU, is prohibitive.

From henceforth the archiver described in this document will be referred to as MYA, which stands for MySQL Archiver. It should also be noted however, that MYA is an acronym for Millions of Years Ago used by astronomers, geologists, and paleontologists, which seems somewhat appropriate for an archiver. As seen in section 9, the MYA logo is an elephant, which as we all know, never forgets anything.

Another major player in the MYA design is real-time threads. Threads are a very efficient means of achieving concurrency within a software system and never letting the CPU sit idle. This benefit is greatly enhanced by today’s modern multiple CPU computers. The ability to assign real-time scheduling priorities to threads means you can take full advantage of the processor’s capabilities, shielding MYA from the negative effects of thrashing, while pushing less important workstations tasks into the background.

2.1 General Design

The MYA software will be a C++ application, compiled with full optimization. It is designed to run on UNIX and Linux systems, ideally having two to four CPUs and a 64-bit architecture. The systems will need to have a large amount of local disk space, as the channel archiving is expected to consume up to 53 gigabytes per day, if the required 25,000 EPICS channel updates per second is realized. The requirements also state that a year’s worth of data must be maintained on-line, so the total disk capacity of all instance of MYA needs to be 19 terabytes.

The prototyping and benchmarking of MYA was performed on oparsrv1, which is a Red Hat Linux 4ws system with two CPUs and a 32-bit architecture. The system’s RAID array was configured as RAID 1+1 with 825 gigabytes of disk space dedicated to the channel history partition. Some portion of the testing was done with the RAID-5 configuration, with no significant effect on performance. Because of the efficient disk space performance of RAID-5, it is the recommended configuration. The following is a summary of the system specifications.

- 2 x 2.6GHZ INTEL XEON 533 FSB W/ 512K Cache
- 8GB PC266 DDR ECC Registered Memory
- SCSI Ultra320 10K RPM Hard Drives
- 2.5Tb (Disk Spare) U320 RAID Enclosure (5TB Non Raided)
MYA may be deployed onto a different hardware configuration than was used for the study. Since it was designed as a distributed system, a deployment may make use of a larger number of simple hosts or a smaller number of powerful hosts. Benchmarking needs to be performed on various types of hosts to determine the most cost effective strategy. Whatever hosts are selected, they must be able to hold 19 terabytes of files on local disk space and stuff new database rows at a rate of 25,000 per second.

2.2 Scalability
MYA achieves scalability through distributed processing. One archiver instance consists of one host computer, a MySQL server, local disk storage for database tables, and the MYA software. Each instance is independent, archiving its own designated set of EPICS channels and storing the data into local disk space. With this design, you need only to add additional instances to scale MYA to a larger set of control system data. The scalability is however limited by the capacity of the network that joins all of the instances to the Jefferson Lab control system. The network capacity is not an issue with the magnitude of channel access required by the archiver requirements. One logical deployment of MYA, consisting of one or more archiver instances, will have a channel directory residing on one of the instances, which is considered the master instance. All instances will obtain information about what they are to archive from the directory. Due to the magnitude of channel archiving scalability specified in the requirements, the master instance will not present a detrimental impact on the system, as the directory tables will remain relatively small.
2.3 **Software Architecture**

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

The Main thread simply sleeps on a UNIX named FIFO (pipe) waiting for user commands. The FIFO was chosen for its simplicity. 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 one of the worker queues, for each Channel Access monitor event. There are a variable number of Scribe threads, with the number defined in the MYA configuration file. Each of these threads has their own localhost (socket) connection to the MySQL database. They sleep on their personal work queue, waiting for channel events to process and store in the database, via SQL statements. Note that each EPICS channel is assigned to one of the work queues, so that its events will only be pushed into that queue. This is because of the MySQL prepared statement interface used by this design (see section 7).

Remote TCP/IP connections to the MySQL server are also created when client applications make history requests via the MYA history API. The MySQL client library is the heart of the MYA history API. Therefore the MYA history API is just a collection of SQL statements wrapped in a language specific abstraction made to facilitate usability for the general history client.

Two additional maintenance threads are defined within the MYA software. The Watchdog thread performs low rate database maintenance, such as culling old channel history and disabling channel archiving when a channel’s time period expires. The Copycat thread makes incremental backups of channel history tables.
The maintenance activities are also shared by the EPICS thread, because some maintenance activities must be more closely synchronized with EPICS monitoring. The EPICS thread waits for EPICS monitor events for a fixed time span, which is defined in the configuration file at 30 seconds, then pauses to see if there are some maintenance activities to be performed. A shutdown request is only acknowledged during one of these monitoring pauses. The maintenance activities performed during the monitoring pause are all very brief, so that EPICS channel event buffers do not overflow. They include temporarily disabling archiving to avoid queue buffer overflow and handling requests to continue/discontinue the archiving of a channel.

MYA threads use the FIFO style real-time scheduling. Higher priority threads will not yield to a thread of lesser or equal priority until they explicitly enter into a sleep state. A higher priority thread will force a lower priority thread to yield when it is ready to run. The order of MYA threads by decreasing priority is the EPICS thread, Main thread, Scribe threads, and Watchdog/Copycat threads.

### 2.3.1 Database Design

Each instance of MYA will have its own MySQL server associated with it. One instance in a logical group of instances will have a set of master tables that contain information about each channel being archived by any of the collaborating instances. The master tables include channel identification information, the instance that is responsible for archiving the channel, channel meta-data, channel controls, and channel grouping structures.

Every defined channel will have its own database table, within the responsible MYA instance’s MySQL server, which contains channel update events. Because a MYA instance may be responsible for archiving thousands of EPICS channels, each MySQL server can have thousands of channel event tables.

```
[mysqld]
open_files_limit=65536
table_cache=30000
```

Each instance’s MySQL server will be configured to optimize the case where there are many database tables. The default MySQL configuration file will be modified to change default settings as shown to the left.
2.3.1.1 The Channel Directory Table

The master MYA instance contains the directory table that catalogues all of the channels having channel history stored in one logical archiver. The directory table is a MySQL InnoDB table because transactional operations will be performed. The first four fields of the table uniquely identify the channel by database ID, EPICS name, EPICS data type, and array element count. The `adel` field is an optional archive-delta (see section 3.1.2). The `expire` and `clip` fields are optional fields, with `expire` declaring an absolute date when archiving of the channel will cease, and `clip` defines a relevancy span of collected data (days before now). The `active` flag denotes when a channel should be archived, and the `alert` flag notifies the running archiver that there is a change in state of the `active` flag. The next two fields identify the host computer and port number of the MYA instance that is responsible for archiving the channel. The last field is the date of the last backup for the channel.

The channel directory table is an extra burden to the master instance and could be expected to be a bottle neck when accessing channels from a complete logical archiver. The reason this table does not have significant negative impact on MYA is the number of channels that MYA is required to support is not larger, when considering sizes of database tables. MYA is required to support up to 100,000 channels, which is only a little more than one day’s worth of one-second update rate data. Searches will be fast because the search columns are indexed. The file sizes, including indexes, for a 100,000 channel directory is estimated to be about 7MB, which has insignificant impact on the disk capacity of a MYA instance. Even considering the required factor of ten in scalability, the channel directory table will not become a problem to the MYA design.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan_id</td>
<td>int unsigned</td>
<td></td>
<td>PRI</td>
<td>NULL</td>
<td>auto_increment</td>
</tr>
<tr>
<td>name</td>
<td>varchar</td>
<td></td>
<td>MUL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>type</td>
<td>tinyint unsigned</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>smallint unsigned</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>adel</td>
<td>varchar</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>expire</td>
<td>timestamp</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>clip</td>
<td>int unsigned</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>active</td>
<td>tinyint</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>alert</td>
<td>tinyint</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>host</td>
<td>varchar</td>
<td></td>
<td>MUL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>port</td>
<td>smallint unsigned</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>backup</td>
<td>bigint</td>
<td>YES</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
2.3.1.2 The User Configuration Tables

**users** (InnoDB)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_id</td>
<td>int unsigned</td>
<td></td>
<td>PRI</td>
<td></td>
<td>auto_increment</td>
</tr>
<tr>
<td>name</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>password</td>
<td>blob</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>privilege</td>
<td>blob</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>email</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

**groups** (InnoDB)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>group_id</td>
<td>int unsigned</td>
<td></td>
<td>PRI</td>
<td>NULL</td>
<td>auto_increment</td>
</tr>
<tr>
<td>active</td>
<td>tinyint</td>
<td>YES</td>
<td>1</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>comment</td>
<td>varchar(255)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>owner</td>
<td>int unsigned</td>
<td></td>
<td>MUL</td>
<td>0</td>
<td>users.user_id</td>
</tr>
<tr>
<td>expire</td>
<td>date</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

**members** (InnoDB)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>group_id</td>
<td>int unsigned</td>
<td></td>
<td>MUL</td>
<td>0</td>
<td>groups.group_id</td>
</tr>
<tr>
<td>chan_id</td>
<td>int unsigned</td>
<td></td>
<td>MUL</td>
<td>0</td>
<td>directory.chan_id</td>
</tr>
</tbody>
</table>

The configuration tables will be of a type where safe transactions will be performed. The MYA configuration tool will manage these tables. The tables keep track of the tool’s users and the channel archive grouping that they create. These tables are modeled after the ADM tables defined for use with CZAR. Refer to section 3.1.6 for information about the configuration tool.

2.3.1.3 The metadata table

**metadata** (InnoDB)

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan_id</td>
<td>int unsigned</td>
<td></td>
<td>MUL</td>
<td>0</td>
<td>directory.chan_id</td>
</tr>
<tr>
<td>keyword</td>
<td>varchar(255)</td>
<td></td>
<td>MUL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stamp</td>
<td>timestamp</td>
<td>YES</td>
<td></td>
<td></td>
<td>CURRENT_TIMESTAMP</td>
</tr>
<tr>
<td>text</td>
<td>text</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There will be two sources of metadata. The directory table described above already contains information about a channel’s type, adel, and element count. Other forms of metadata will be obtained from the meta-data table. It is a simple table of type InnoDB to allow for transactions and foreign key management (chan_id). Every piece of metadata has an associated channel, keyword, and timestamp.
2.3.1.4 The Channel Event Tables

These tables hold channel event information, and there is one table dedicated to each archived channel. They are defined as a MySQL MyISAM tables to gain maximum performance and because no multiple table operations are performed on the channel event tables. The table names are derived from the unique channel identifier numbers found in the directory table. For example, `table_8734` is the name of the table that holds update events for channel number 8,734. Channel event tables all have a very similar format. The basic structure can be seen in the MySQL table creation command shown above. The table definitions vary from this example by the value’s data type and the number of value fields defined. For example an array of three elements would have additional `val2` and `val3` fields. The associations of MySQL field types and EPICS channel types is shown in the following table. Note that a 32-bit floating point type is used to store EPICS double data in order to save storage space. The Jefferson Lab control system does not have quantities requiring the precision or range of 64-bit floating point format. This statement has been verified by Richard Dickson, Al Grippo, Scott Higgins, George Lahti, Clyde Mounts, and Jay Benesch.

<table>
<thead>
<tr>
<th>EPICS Type</th>
<th>MySQL Type</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBR_STRING</td>
<td><code>char(40)</code></td>
<td>identical</td>
</tr>
<tr>
<td>DBR_SHORT</td>
<td><code>smallint</code></td>
<td>identical</td>
</tr>
<tr>
<td>DBR_FLOAT</td>
<td><code>float</code></td>
<td>identical</td>
</tr>
<tr>
<td>DBR_ENUM</td>
<td><code>smallint unsigned</code></td>
<td>identical</td>
</tr>
<tr>
<td>DBR_CHAR</td>
<td><code>tinyint unsigned</code></td>
<td>identical</td>
</tr>
<tr>
<td>DBR_DOUBLE</td>
<td><code>float</code></td>
<td>64-bit -&gt; 32-bit</td>
</tr>
</tbody>
</table>

The time field is a 64-bit integer. A native MySQL time data type is not used because they do not include fractions of a second. The data stored in this integer is actually the composite of the standard UNIX time and the fractional part of the second. UNIX time is a 32-bit offset in seconds from the epoch (1970-01-01 00:00:00 GMT), and is stored in the upper 32 bits of the field. The fractional portion of the second is stored in the lower 32 bits of the field, with the weight of the MSB being 0.5 seconds. Therefore the units of the whole 64-bit timestamp can be thought of as seconds-times-2^32.

The data tables use the time field as the primary key. This enforces unique timestamps and makes queries for a channel over a desired time range very fast. Since the data for a channel is inserted into its table in increasing time order, a block of consecutive data updates is fetched with one disk seek.

The code field is used to identify the type of event. There can be data value update events and discontinuity events. Discontinuities can occur because of MYA shutdown, EPICS channel disconnection, or a request to discontinue archiving a channel. Note that EPICS connection events are not stored, as these are always followed immediately by a data update event. The code field from database also contains information about the first data
point in the table. It tells the reason why there is no prior data available, which can be no
data was collected, prior data was moved offline, or prior data was discarded.

MySQL tables do have size limitations, even when not limited by the basic host operating
system file size limit. The MYA design uses a 4-byte index pointer, therefore is limited to
$2^{32}$ rows per table, which translates to 136 years of data updating once per second.

The output above shows results from an analysis of a MYA channel event table that was
created to have one second updates for a year. It is a table holding an EPICS
DBR_DOUBLE data type, and takes up 791MB of disk space for the data and index files.
Note that arrays may pose a challenge to this design. Row lengths will grow linearly with
element count, so maximum file sizes could be much larger. I think EPICS has an 8KB
buffer limitation, I suppose that the number of possible array elements is also somewhat
bounded. Some thought may need to be given if we ever intend to archive many-element
arrays that change very often.

### 2.4 MYA Configuration File

Each instance of MYA will have its own configuration file, which will be read at
program start up. It will contain the following information.

- Host machine and port number used by the master instance’s MySQL server.
- Host machine of the local MySQL server, which is used to look up in the master
directory the names of channels this instance will archive.
- Parameters associated with maintenance activities, such as the detection of the
disk full situation, the automatic channel turn off, and relevancy culling.
- The number of database worker threads to create.
- Where to put backup files.
- The duration of monitoring before the maintenance pause.

### 2.5 Documentation

The MYA release will be bundled with a full suite of documents, including the
requirements document, design document, administrator’s guide, and developer’s guide.
The administrator’s guide will detail all features and techniques involved with installing and operating the MYA system. The developer’s guide will be a road map of how this design maps onto source code files, functions, and classes.
3 Addressing Specific Requirements

3.1 Functionality

3.1.1 Archive Channel Values
The MYA software uses the EPICS 3.13.10 client library to access channel events via channel monitor call-backs. All EPICS control system variables are accessible via this interface. The latest version of the 3.13 series was chosen over any of the 3.14 series because the latter versions are somewhat unfriendly to C++ exceptions. An application’s call-backs do not execute in the context of the thread that initiates the activity, but in some internal EPICS thread. An exception thrown causes a task abort instead of working its way back to your application.

3.1.2 Channel Value Resolution
The master directory table has a column for the archive-delta associated with each channel. It may be defined as null, which means MYA will archive all changes. The archive-delta for strings and all arrayed types of channels will be ignored.

3.1.3 Channel Metadata
Some channel meta-data will be obtained automatically by MYA at channel connection time. Included are the EPICS data type, element count, and the enumeration strings for the enumerated data type. The data type and the element count are included in the directory table and will not be modified by MYA when it detects a difference. An alert will be generated and the archiving of the channel will be discontinued.

The enumeration strings as well as manually entered meta-data will reside as informational records associated with the channel. These records will be identified by a set of keywords that may be used to request certain categories of informational metadata. All informational records will have a timestamp. See section 2.3.1.3 for more details.

3.1.4 Channel Disconnection
MYA will store a discontinuity entry in a channel’s table when a disconnection event is received. Discontinuities are also stored for MYA shutdown or the discontinuation of archiving by request. Refer to section 8 for information on connection management.

3.1.5 API Data Requests
See section 4 for a full specification of the MYA C++ API.

3.1.6 Designating Channels
The ADM tool\[iv\] that was developed to allow archive configuration requests will be renovated for MYA.

Major differences include the following.
- Concurrent usage will be allowed. Transactions and table locking allow multiple users to interact at the same time.
There will be no concept of streams, only channels.

There will also be a web site developed for making requests that affect the core history set. Normal user modifications to the history set will only apply to specially designated MYA instances, which will be thought of as a sandbox area for temporary archiving requests.

### 3.1.7 Event Logging

Each time MYA is started it will create a new log file, with the date and time encoded in the file name. The log is not intended as a repository of regular generated diagnostics, but will only contain messages indicating special events or exceptional conditions. The strategy is to keep the log short so that it may be easily browsed by an administrator. Each entry in the log will contain a timestamp. The brief log should not grow to significant size between executions of MYA and does not pose a disk capacity problem.

When MYA encounters serious exceptional conditions, it will generate alert messages. Alert messages will be emailed to several key staff members and various pagers. Alert messages will also be written to the MYA log file. Another form of alert message will be emailed to announce MYA configuration changes, such as those declaring that a user’s temporary channel group will soon expire, however these types of messages will not be included in the MYA log.

### 3.1.8 Data Culling

MYA performs certain functions on regularly scheduled maintenance intervals, as described in section 2.3. The culling of irrelevant old channel history falls within the maintenance functions. Note that a field in the channels table (section 2.3.1.1) has an age parameter for each channel. Channels with data older than the stored maximum age will be discarded.

Note that there is a MYA problem caused by the way MySQL deletes rows from tables refer to section 5 for details.

### 3.1.9 Automatic Channel shutoff

Similar to Data Culling, the maintenance functions will halt archiving a channel when the absolute date stored for the channel in the channels table is reached. The owners of any active group referencing that channel will be notified prior to the shutoff occurring, as is done for the current ADM tool (see section 3.1.6).

### 3.2 Usability

#### 3.2.1 Administration

The MYA prototype was created with an administrator’s tool, named mahout, which is used to start, stop, and communicate with MYA while it is running. The odd name can be understood when you remember that MYA is an elephant. A mahout is an elephant driver in India. Mahout must be executed on the host machine of the MYA instance you wish to control. That is because its communications mechanism is UNIX named pipes, which do
The command line syntax is `mahout command [argument]`. The following table lists valid commands.

<table>
<thead>
<tr>
<th>command</th>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>channel</td>
<td>&lt;name&gt;</td>
<td>Is the specified channel being archived, and if so, is it currently connected.</td>
</tr>
<tr>
<td>collect</td>
<td>on</td>
<td>Turn the collection of archiving statistics on or off. When on, statistics are computed every monitor pause cycle (30 seconds).</td>
</tr>
<tr>
<td>disk</td>
<td></td>
<td>Show results of the hourly disk partition check, including percent used and estimated disk-full date.</td>
</tr>
<tr>
<td>help</td>
<td></td>
<td>Shows this list of commands. You can also use &quot;-h&quot;.</td>
</tr>
<tr>
<td>Qsize</td>
<td></td>
<td>Show the current size of the MYA channel event queue.</td>
</tr>
<tr>
<td>start</td>
<td></td>
<td>Start MYA. It reads from its configuration file and runs as a daemon. It checks to make sure MYA is not already running.</td>
</tr>
<tr>
<td>stats</td>
<td></td>
<td>Print archiving statistics. Included are the channel event rate and the connected/disconnected channel counts.</td>
</tr>
<tr>
<td>stop</td>
<td></td>
<td>Stop MYA. This command will wait until the next monitor pause cycle (section 2.3).</td>
</tr>
<tr>
<td>version</td>
<td></td>
<td>Show the MYA logo and version number.</td>
</tr>
</tbody>
</table>

There will also be a program developed to send the same command, via ssh, to all instances of MYA. This will expedite mass startup and shutdown procedures.

Another administrative tool will be developed for managing the load on MYA instances. This tool will allow an administrator to change which instance a set of channels are assigned to. Designating channels will be done by direct name, lists in a file, or wildcard matching. The administrator will need to designate the host and port of the new instance. This feature is intended only for a shutdown MYA system, but an investigation of dynamic load balancing will be made at some time. If feasible, this capability would likely appear in some update to the initial MYA release.

MYA and all API client tools need to know the host name and port number of the master instance of the logical archiver with which they will interact. There will be a master configuration file in a certified location, so that all applications may get this information form a common source. Then a re-hosting of the master instance of a logical archiver can be put into effect by making one configuration change.

### 3.2.2 Archiver Instances

See section 2.3.

### 3.2.3 The Archiver API

See section 4.

### 3.2.4 The API Environment

The MYA C++ API (section 4) is built around a set of SQL statement, as all of the information provided by the API is in database tables. MySQL provides APIs to access a database server for ‘C’, PHP, Perl, C++, Python, Tcl, and Eiffel. The same queries used in the MYA C++ API can be utilized in APIs for these other languages. Language
specific mappings of data structures and interfaces will vary however. No MYA API other than the C++ style has been prototyped at this time.

### 3.2.5 The Configuration Tool
See section 3.1.6.

### 3.2.6 Channel Set Organization
See section 3.1.6.

### 3.3 Reliability

#### 3.3.1 High Availability Archiver
The MYA prototype, in various forms, has been running much of the time over the last three months and has had no unexplained faults. This includes long segments where automated retrieval requests have fetched channel data and automated executions of the mahout tool has accumulated statistics.

MYA will be developed under the Certified Software Initiative. This will help to ensure quality. The requirements analysis and the design review has already put the effort on the right track, but will also be supplemented by code reviews, third party testing, documentation, and configuration management.

An analysis of the virtual memory demands of MYA has been performed to check for memory leaks. The chart below indicates the memory footprint is not increasing over time, however the marked drop between two plateaus is odd.

![Memory Use Chart]

#### 3.3.2 Fault Tolerance
The MYA software explicitly looks for disk capacity limitations as part of its regular maintenance activities. It will delete history older than a computed date when it sees that
it is close to being out of disk space. It computes the date by adding an age from the
configuration file to the oldest sample in the history set. MYA’s being close to out of disk
space is also configurable, and is defined in terms of its disk check cycle. The proposed
disk check cycle is one hour and the too close limit is 24 check cycles.

MYA makes use of C++ exceptions to control what happens when fault situations are
detected. Exceptions handlers are defined at many levels and the only exceptions that can
escape to a thread’s main function are those detected at the program’s initial startup. The
default EPICS exception call-back is overridden so that serious EPICS problems will not
result in a program abort.

The MYA software examines its work queue at every monitor pause cycle (section 2.3).
If the size of the queue exceeds a configurable value it will stop queuing work generated
from EPICS monitors and log the action. This will cause the loss of current data, but is
preferable to running out of memory and crashing. Once the peek performed during the
monitoring pause sees a small work queue size, also configurable, it will re-enable the
queuing of EPICS monitor events and update the log accordingly.

EPICS also has fault tolerance built in. If the MYA does not respond fast enough to
monitor events, buffering will hold the events up to a certain point. Once its buffers are
full, the older channel events will be discarded gracefully.

Using an industry standard database to manage data archival and retrieval provides a
great deal of fault tolerance. MySQL servers have a huge user base and are a proven
component of MYA.

3.3.3 Channel History Backup
See section 6.

3.3.4 Core Channel Set Preservation
The use of dedicated MYA instances for the core channel set provides this assurance.
User archive requests are not placed on the reserved MYA instances.

3.3.5 Automatic Startup
A system startup script on a MYA host will contain the mahout start command and a
shutdown script will have the corresponding mahout stop command. I don’t really
understand this requirement, so some clarification is needed. Mahout will be responsible
for making the start decision. If it somehow ascertains that the hosting system is in an
untoward state, it will refuse to start MYA.

3.3.6 Watchdog
A standalone program will be developed, which will run on any chosen host and will
need to know the host/port of the master MYA instance. It will ask the master instance
for the names of all of the MYA instances, so that it can issue remote commands in order
to ascertain that the systems are up and the requisite processes are operational. The
MySQL server on the host will be contacted and asked its status. Do not confuse this
required external facility with the MYA watchdog thread.
3.4 Performance

3.4.1 EPICS Channel Capacity

The MYA prototype as benchmarked (see section 2.1) should not be loaded with more than a nominal 7,000 channel updates per second, unless a more suitable host is obtained. This rate will have an adverse effect on channel fetch time, so a rate of 6,000 is recommended. The *Queue Size* chart shows the sampling of the MYA work queues, summed once every minute over a five hour period. The chart shows queue sizes for update rates of both 7,000 and 6,000 per second. The queue size remains well below its 2,000,000 event limit for either rate, however the faster rate test averaged approximately 159,910 queued events. This would contribute about 23 seconds of latency for data retrieval, since data stuck in the queue can’t be fetched from the database tables. The queue size averaged 5,370 events for the slower update rate test run, which contributes less than a second to data latency.

![Queue Size Chart](image)

The tests above were performed with 7,000 and 6,000 channels updating at once per second. The EPICS monitor event rate is not the only factor in determining the maximum load MYA can handle. The number of channels also is a factor. MYA can handle 4,000 channels firing at once per second easier than 8,000 channels firing at once every other second. In fact MYA easily handled 1,000 channels updating at 10 times per second for an overall throughput of 10,000 channels per second. The reason has to do with each channel having its own data table, and therefore its own data file on disk. The MySQL server has to do more switching between files during updates of a larger channel set. The studies done throughout most of the MYA prototyping do not match the characteristics of the accelerator control system and the MYA loading limits will need to be studied once MYA is deployed in the field.

The evidence indicates that we will need four machines to meet the requirement for 100,000 channels at 25,000 updates per second, with the previously cited 19 terabytes of disk space (see section 2.1) divided up among them.
3.4.2 Prefer Recent History
Taking older data offline will provide for this requirement. MYA in general suffers very little due to the size of the history set. This is because of MYA instances, individual tables per channel, and table indexing.

3.4.3 API Throughput
The throughput of the API has been extensively analyzed using the MYA prototype software. Note that fetch times are greatly aided by caching on the MySQL server, but repeated fetches of the same data are not the normal scenario, so benchmarks cited in this section represent un-cached fetches. MYA loading also is a big factor. A set of 250,000 channel value fetches on an unloaded MYA instance were received at steady rates of around 140,000 and 210,000 samples per second for the ASCII and binary interfaces respectively. The charts below show that the loaded system times are variable and significantly slower.

Test software was written to repeatedly fetch channel data from MYA using the ASCII interface, logging the time taken to perform the fetch. A set of fetches were performed every five minutes. The history set started out empty and grew to about 60 days worth of data during the test. Note a day of data is defined as 86,400 points, which is produced by a channel firing once every second throughout a 24 hour period. The fetches were made of the oldest day, most recent day, and all days. The oldest day fetches were faster than the most recent day because the index is not traversed to find the start of the table. Also the oldest day time does not depend on the amount of accumulated data, but the most recent day search requires more traversal of the index the larger the size of the channel’s data set. The increase in fetch time should be logarithmic in nature however, quickly flattening out after a certain size of the history set.

As can be seen by the chart above, the full loaded fetch time of the oldest day of data makes the 3.5 second requirements except in a few isolated instances. There average amount of fetch time in this chart appears to be about .75 seconds from the cyan moving average plot, which corresponds to a rate of about 115,000 samples per second. This
should be compared to the unloaded samples shown in the chart below. There is not a huge difference in the average fetch time, but the variation is greatly reduced when loading is minimal.

The last day fetch shown in the chart below varies with the size of the history set. A logarithmic fit was added to estimate the fetch time of the latest day after a year of collection, which is calculated as 1.6 seconds using the formula shown on the chart. That fetch time represents a data rate of 54,000 samples per second, surpassing the 25,000 required rate.

Another interesting chart shows the amount of time taken to fetch the entire history set for a channel, as a function of that channel’s history set size. The time appears to be linearly increasing at about 0.64 seconds per day of data, corresponding to about 135,000 samples per second.
The requirement for fetching 10,000 different channels in one minute has not been met. Single value fetches are slow with the MYA prototype API, so a specialized API method called ArchivePortal::MultiPast was created for the case where you want to get a large number of channels at the same time. It took about 3 minutes to fetch single values for 10,000 channels on a normally loaded system (6,000 updates per second). This is triple the required 1 minute fetch time.

### 3.4.4 API Data Latency

The MYA prototype has easily met the latency requirement, and has often been measured to be less than one second. The one area that greatly affects latency is the work queue size, which can grow large on heavily loaded systems. The chart in section 3.4.1 shows an average queue size of about 10,000 samples, which contributes 2.5 seconds to latency at a channel update rate of 4,000 events per second. With low loading on MYA, the queue size is typically zero and contributes nothing to latency.

### 3.4.5 Offline Channel History

MYA will delete old channel history to make room for new data when there is not enough disk space to accommodate all of the information. MYA will regularly sample disk space availability and its rate of consumption to determine when it falls with a configuration file specified time span of running out of disk space. When it detects this situation, it will remove all data events falling within a time span starting from the oldest data in the database. The removal span is also defined in the configuration file. MYA will not save the data it deletes as it will already exist in the MYA backup set. See section 5 for more information about this feature and section 6 for a description of the backups.
3.5 Design Constraints

3.5.1 Budget

Needed to Buy with a 5 System MYA:

----------------------------------
4 RAID Array [8-1] $22000
88 SCSI Disks [(4x24) - 8] $66000
3 Computer systems $33000
Memory for one system $6000
----------------------------------------
$127,000

Note that this is the cost of the system that handles the full requirement running constantly year-round, which will not likely be the case. There are always scheduled down times and we currently have no need to archive 25,000 updates per second.

The scenario described above does not consider the cost of backing up the huge history set. The backup study shows we will need to do something with 6TB per year of backup data generated by MYA.

3.5.2 Schedule

I have not done a scheduling exercise for the MYA development effort. The time allotted by the requirements does not seem unreasonable for this effort, but closer examination should be made if this design shows promise and the deadline is serious. I did not want to spend a lot of time scheduling activities before at least a partial approval of this design is gained.

3.5.3 Environment

MYA makes no special demands on the lab computing environment.

3.5.4 Lifespan

MYA uses the 32-bit UNIX time, which is valid only until January 19, 2038. At this point significant MYA maintenance will be required. The C++ language, POSIX standards, and UNIX like operating systems should persist for many years. The MYA design does not rely heavily on any specific hardware system, so replacing platforms should not be an issue.

EPICS 3.13.10 is the last of the 3.13 series, dated April 2004. It is not known if there will be continued support of the 3.13 series since the 3.14 series is well vested (see section 3.1.1).
4 The MYA C++ API

4.1 Preliminaries

```cpp
namespace MYA {

    template <typename T, template <typename> class E=MYA::ArrayEvent>
    struct Era { typedef std::vector< E<T> > type; };

typedef std::vector<std::string> TextSet;
typedef std::multimap<float,float> Correlation;

class ArchivePortal

    public:
        ArchivePortal(const char *, unsigned=0);
        template <typename T>
        int Past(const char *, long, long, typename Era<T,ScalarEvent>::type &);
        template <typename T>
        int Past(const char *, long, long, typename Era<T,ArrayEvent>::type &);
        template <typename T>
        int Past(const char *, long, ScalarEvent<T> &);
        template <typename T>
        int Past(const char *, long, ArrayEvent<T> &);
        template <typename T>
        void MultiPast(const TextSet &, long, std::vector<T> &);
        int Count(const char *, long, long);
        int Correlate(const char *, const char *, long, long, Correlation &, CorrelateStyle=Interpolate);
        void Metadata(const char *, const char *, Era<std::string,ArrayEvent> &);
        void Keywords(const char *, TextSet &);
        void MatchChannel(const ChanSpec &, TextSet &);
        void MatchGroup(const ChanSpec &, TextSet &);
        void ExpandGroup(const char *, TextSet &);
        static bool m_alwaysData;
};
```

The MYA C++ API will be a certified library. Applications using the API will include the header file `archiveportal.h`. The MYA API facilities are encompassed in `namespace MYA`. Users of the API will start by creating an instance of an `ArchivePortal` object. The host name of the master MYA instance and the associated MySQL server’s port number must be passed to the constructor. A `cpplib::Exception` is thrown if the provided information is erroneous. Once construction is complete, all queries about channel history will be made through the `ArchivePortal` object. The API also contains functions and definitions to assist users in fetching and using the data associated with EPICS channels.
4.2 Requesting channel events

The API provides the user with channel events, which are encapsulated as either `ScalarEvent` or `ArrayEvent` objects and consist of a timestamp and some data. The data may be a channel value, discontinuity, or meta-data. Collections of events are kept in an `Era` object, which is a STL container. The events in an `Era` are stored in ascending order by timestamp. Note you can insert an `Event` into an output stream to show the encapsulated information in a concise text format.

The difference between the two types of events is that `ScalarEvent` objects can have only one data value, while the `ArrayEvent` objects can have many. The scalar type provides slightly better usability and slightly faster fetches. Note the array type can be used to fetch single value channel events.

These event objects are templates, so users can request channel event data in any desired basic data type. When the user request type is incompatible with the epics data type for the channel, an exception will be thrown. Note that `std::string` can be used as the request type for any epics data type.

The timestamp associated with an event may be obtained in various formats. All binary formats refer to an offset from the UNIX epoch. The `DateTime` methods provide the
timestamp as a 64-bit binary value in seconds-times-2^32, or as a POSIX `timespec` structure. You can also get just the seconds portion of the UNIX offset with the `Seconds` method. You can get the full date and time text string by using the `DateString` method. The format of the string returned is “YYYY-MM-DD HH:MM:SS” by default, but a decimal fraction of the second may be appended if the `m_precision` variable is set to a non-zero value. The variable specifies the number of fractional digits to be appended to the basic timestamp string. The variable is a static member, so setting it once affects all `Event` instances. It also governs the formatting of an event when using the stream insertion operator mentioned above. The time between two elements can be obtained, in fractional seconds, by simply subtracting the two event objects.

The value of a channel event is obtained via the `Value` method. Scalar events return a reference to the channel’s value, while array events return a reference to a vector of the desired data type. Note that the value returned is undefined for non-data events.

The `Code` method returns a bit field of information about the type of event. You can use inline functions defined in the static `EventType` class to interpret the bit fields in the code variable. Note that the code is only meaningful for EPICS monitor events. The code will distinguish between data and discontinuity events via the `Event::IsData` method. You can use the function `EventType::IsNormal` on a data event code to determine if the data event follows a discontinuity (false).

The `Event::Reason` method returns a phrase describing the nature of a discontinuity. The `EventType::Prior` method returns a descriptive phrase about `Events` occurring before this one. It will always return “There are previous data values available” for normal and
discontinuity *Events*, however it will return information about data being discarded, moved offline, or origin of the history for not normal data *Events*.

The *Elements* method returns the number of array elements contained in the data, which will always be ‘1’ for non arrayed data. The *Value* method returns a string of single space separated elements for arrays.

### 4.2.1 Specifying a Date and Time

API users request channel values by time, which is specified as the number of seconds past the UNIX epoch. This is not a human friendly format, so the `cpplib::StrToSec` function is provided to convert between formats. The function takes a user time specification, absolute or relative, and converts it from an alpha-numeric string to the binary number of seconds past the UNIX epoch. The absolute time format is rigidly formatted as "YYYY-MM-DD HH:MM:SS", however today’s date is assumed if the date is not supplied and 00:00:00 is assumed if the time is not supplied. Relative time is specified as a signed offset from NOW, with an optional units specification character appended to the numeric offset {s:second m:minute h:hour d:day w:week}. If no units are specified on the relative offset, seconds are assumed. Note a positive non-zero offset means some time in the future. Also note that this function does not account for leap seconds when using with relative times. Note that this section is included for completeness, however this utility function was added to `cpplib` version 3.2 and is not actually part of the MYA API.

#### long StrToSec(const char *);

**Examples:**
- `StrToSec("2000-01-01 00:00:00");` // Millennium
- `StrToSec("2000-01-01");` // "
- `StrToSec("23:59:59");` // End of today
- `StrToSec("0");` // Now
- `StrToSec("-1d");` // Now minus 1 day

### 4.2.2 Getting EPICS Data

These `ArchivePortal` methods fetch EPICS monitor events for a channel. The *Past* methods are used to fetch the channel events. Each of them returns the epics type code of the requested channel. Note that this may differ from the requested data specified to the function template. These methods come as two pairs, one pair for fetching events between a range of time and the other pair for fetching an event at a time. The reason for method pairs is that both types of fetches can get results as either `ScalarEvent` or `ArrayEvent` objects.

When fetching a range of events, you pass both a begin and end time as a long integer number of seconds past the epoch, as well as the name of the desired channel and a
container for the returned events. An exception will be thrown when the request cannot be granted. The container can be filled with both data events and discontinuities.

The other pair of methods is passed a single time value, and the closest prior event will be fetched. This may be either a data event or a discontinuity. The behavior of this pair of methods may be modified to always return the closest prior data event by setting the `ArchiverPortal::m_alwaysData` flag to true. This flag defaults to false.

A specialized method was created to optimize one special use-case. The situation when a large set of EPICS channels are being restored to their archived values from one one specific point in time is not well handled by the standard single value fetch signature of `Past`. It takes too long. The `MultiPast` method is passed a container of channel names and one timestamp. It returns a container of values for the passed channel names, in the same order that the channel names are found. Values for all of the requested channels are returned as the same type, which is defined by the type of vector passed to the method. Conversion will occur when the actual archived type differs, although an exception will be thrown when the conversion would contribute to loss of information.

In all methods, a `cpplib::Exception` is thrown when the requested channel does not exist.

### 4.2.2.1 Correlation Data

The situation when a large set of EPICS channels are being restored to their archived values from one specific point in time is not well handled by the standard single value fetch signature of `Past`. It takes too long. The `MultiPast` method is passed a container of channel names and one timestamp. It returns a container of values for the passed channel names, in the same order that the channel names are found. Values for all of the requested channels are returned as the same type, which is defined by the type of vector passed to the method. Conversion will occur when the actual archived type differs, although an exception will be thrown when the conversion would contribute to loss of information.

In all methods, a `cpplib::Exception` is thrown when the requested channel does not exist.

#### 4.2.2.1 Correlation Data

<table>
<thead>
<tr>
<th>ArchivePortal Methods</th>
<th>To create a correlation between two channels, you must supply the names of both channels and the time span of interest. The result is returned in a <code>Correlation</code> object, which is a collection of data pairs. Refer to using an STL <code>multimap</code> for handling the result. The result is always pairs of floating point values, no matter what the original EPICS types of the channels. Conversions are made when necessary, except string data types will cause an exception to be thrown. The non discontinuity <code>Events</code> of the first channel are used as a basis. Interpolations, by default, into the time/data points of the second channel are made, however the first channel’s sample is discarded when the value of the second channel is unknown. You will also be able to choose other techniques besides linear interpolation for obtaining the value of the dependent variable at a sample of the independent variable. These are MYA::Prior, MYA::Next, and MYA::Closest.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Correlate(const char *, const char *, long, long, Correlation &amp;, CorrelateStyle=Interpolate);</code></td>
<td></td>
</tr>
</tbody>
</table>

#### 4.2.3 Getting Channel Meta-data

You request meta-data for a channel by using a keyword that identifies the type of meta-data desired. There are a set of keywords used for all channels (adel, element_count, epics_type, first_timestamp) and other keywords that are defined for only certain channels (enumerations, cull_dates, offline_dates, units) and administrator defined keywords. You can get the list of keywords for a channel using the `ArchivePortal::Keywords` method. Use the `Metadata` method, passing the channel name and keyword to get one or more time stamped text string related to the keyword.

<table>
<thead>
<tr>
<th>ArchivePortal Methods</th>
<th>You request meta-data for a channel by using a keyword that identifies the type of meta-data desired. There are a set of keywords used for all channels (adel, element_count, epics_type, first_timestamp) and other keywords that are defined for only certain channels (enumerations, cull_dates, offline_dates, units) and administrator defined keywords. You can get the list of keywords for a channel using the <code>ArchivePortal::Keywords</code> method. Use the <code>Metadata</code> method, passing the channel name and keyword to get one or more time stamped text string related to the keyword.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void Metadata(const char *, const char *, Era&lt;string,ArrayEvent&gt; &amp;);</code></td>
<td></td>
</tr>
<tr>
<td><code>void Keywords(const char *, TextSet &amp;);</code></td>
<td></td>
</tr>
</tbody>
</table>
4.2.4 Getting Channel Configuration Data

Users of the MYA API request channel information by channel name. However, users of the applications that use the API may not know the exact name of channels they want to inspect, or they may wish to request information for all channels in a user defined channel-group. The API provides methods to examine the MYA channel archiving configuration.

The \texttt{MatchChannel} and \texttt{MatchGroup} methods are used to get the names of channels known to MYA, and the names of user defined groups of channels. You use \texttt{MYA::Glob} or \texttt{MYA::RegExp} as the \texttt{ChanSpec} parameter to specify which type of pattern matching you want to use. For example, passing \texttt{RegExp("^m")} and \texttt{Glob("m%")} both request all of the names that start with `m` or `M`. The glob wildcard characters are the standard UNIX shell characters `*` and `?`. Note that `const char *` is also a valid \texttt{ChanSpec}, and it is interpreted as an exact match, although it is case insensitive. Passing `m%` will only match the two literal names `m%` and `M%`. The \texttt{ExpandGroup} method provides a list of all channel names in a specified group.

4.3 Examples

4.3.1 Printing Channel History

```cpp
#include <archiveportal.h>
#include <util.h>
#include <string>
#include <iostream>
using namespace MYA;
using namespace cpplib;
using namespace std;

void PrintHistory(
    const char * host,    // Master instance host name.
    unsigned port,        // MySQL server port number.
    const char *channel,  // EPICS channel name.
    const char *begin,    // Sting encoded time stamp.
    const char *end)      // String encoded time stamp.
{
    ArchivePortal ap(host, port);   // Open a portal.
    Event::m_precision = 1;         // Show timestamps w/ tenth second.
    // Fetch a set of EPICS monitor events.
    Era<string,ArrayEvent>::type era;
    ap.Past<string>(channel, StrToSec(begin), StrToSec(end), era);

    // Print the results to standard output.
    Era::iterator it = era.begin();
    while (it != era.end()) cout << *it++;
}
```

27
### 4.3.2 Computations with Channel History

```cpp
#include <archiveportal.h>
#include <exception.h>
#include <util.h>
using namespace cpplib;
using namespace MYA;

// Note that this function will throw if the requested channel is not
// being archived, is of string data type, or has no data events
// within the specified time range.

double LargestValue(
    const char * host,    // Master instance host name.
    unsigned port,        // MySQL server port number.
    const char *channel,  // EPICS channel name.
    const char *begin,    // Sting encoded time stamp.
    const char *end)      // String encoded time stamp.
{
    ArchivePortal ap(host, port);   // Open a portal.

    // Fetch a set of EPICS monitor events.
    Era<double, ScalarEvent::type> era;
    ap.Past<double>(channel, StrToSec(begin), StrToSec(end), era);
    if (era.empty()) throw Exception("What the hell!");

    // find the largest value. First element only if arrays.
    Era::iterator it = era.begin();
    double largest;
    bool found = false;
    while (it != era.end())
    {
        Event &event = *it++;
        if (EventType::IsData(ev.Code()))
        {
            if (!found || event.Value() > largest)
            {
                found = true;
                largest = event.Value();
            }
        }
    }

    if (!found)
        throw Exception("As a failure, you are a great success.");
    return largest;
}
```
5 The Data Culling Problem

MySQL behaved not as I expected when I was testing the deletion of channel data from the database tables. When table rows are removed via the SQL DELETE command, no disk space is actually returned to the operating system. The space is held by MySQL for use only by the table from which they were deleted. This behavior is not ideal for MYA because it makes it very difficult to know when disk space limitations are approaching, and it reserves space for channels that may no longer be getting new channel updates.

The way you deal with this is to perform an OPTIMIZE table command on each table after deleting rows. Unfortunately both the delete and optimize commands take significant time, which grows with the size of the table being trimmed. One thousand tables were created with a busy year of fake data, having 31,536,000 events each. This filled up the oparsrv1 storage partition. The culling algorithm was configured to kick in when the partition was within 12 hours of filling, and cull one week of data from the tables. MYA was lightly loaded in the test run. The culling operation averaged 4.9 minutes per table, which indicates a fully configured MYA instance could take two weeks to complete a data culling session. This being the case, the time span for data culling should be set at one month, so the operation can complete before starting again. A detailed study should be made to aid in the setting of the MYA culling parameters, which are in the MYA configuration file.

The lengthy test run mentioned above pointed out a difficulty with my initial plans. I assumed that the insert-delayed queue for the table locked during culling would just absorb the updates to that table until the culling operation completed on the table. Therefore, the only tweaking necessary would be to make the insert-delayed queue size large enough, and the 1,000 update MySQL default limit seemed sufficient. This did not turn out to be the case. As soon as a table is locked for its long delete and optimize, and the MYA Scribe thread attempts to insert into that table, the thread sleeps until the table culling is complete. Even though the INSERT DELAYED is used, MySQL must still insist on verifying the insertion is legal, which it can’t do until table modifications are complete. This situation would not be such a big deal if it only affected the channel for the table being culled. Updates would just gather in the work queue until the culling was complete. Putting the thread to sleep however, means that all channels handled by that thread will not go to their respective table, and the thread’s work queue will grow rapidly. The MYA data culling design will include a holdout queue. When MYA instigates culling on a table, the updates to the associated channel will be stuffed into the holdout queue instead of one of the work queues. Once the culling is complete, the holdout queue will be emptied into the appropriate work queue. Note that this adds latency for the one channel being held, which is beyond the latency limit stated in the requirements. It is not a persistent latency however.

Another adverse situation found during the trimming of tables was what appeared to be a MySQL bug. Occasionally a worker thread would receive status from the MySQL server that the server was shutting down and the insertion could not be done. This was rare but did occur even at times when the MYA load was such that the table optimizing was not causing queue size overflows. I have entered it as a MySQL bug report, which has been entered into their tracking system. I don’t believe it will be a problem with the scheme I
describe above. I think it was an artifact of the original scheme that continued to try to update a table being culled.

One solution to this problem, as suggested by Theo Larrieu, would be to implement the channel data tables as a collection of MySQL Merge tables. With this scheme, readers of the data address the merge table, while writers store in one individual table. Since data for a channel is then stored as a collection of finite size individual tables, culling of old data could be performed by simply dropping a table from the collection, which is a fast procedure. The individual table sizes would be made to match the desired cull chunk size.

During my initial study of design, I did not give the MySQL merge table much consideration due to overhead, and perceiving little benefit. I may need to rethink this due to the benefit with regard to the problem cited in this section.
6 Table Backup Strategy

The MYA database tables used to store channel events will be backed up incrementally on a daily basis. At the turn of the day, MYA will backup all of the data accumulated since the previous backup for each channel, which will typically be one day’s worth of data. This is sufficient for these types of tables because channel history does not change. The accumulated set of daily backups will always form the complete history set. This is not true for the MYA configuration tables, which may gave modifications and deletions as well as insertions. The backup of the configuration tables will be described later. MYA will create one file per channel in a directory named in the MYA configuration file. This directory must be in a different disk partition than the one used for the database tables. The file names will be <channel name>.<previous date>.<current date>. For example, the backup file R121GSET.2006-01-03.2006-01-04 will include all the channel events for R121GSET that occurred on January 3, 2006. The backup files are expected to be regularly moved to some other form of storage by an external system yet to be created.

The channel event backup strategy was tested on oparsrv1 running with 2,000 updates per second. One day’s worth of data (172,800,000 data points) took 37:12 to dump to files in the local file system directory /tmp/backup. The size of the created data files was 4.5GB, which reduced down to 1.4GB when zipped. The zipping process took about twelve minutes and is not part of the MYA backup actions. I expect a heavily loaded MYA instance to actually be averaging two and a half times this load, so the backup directory will need about 14GB to hold the unzipped files for one instance.

The external system that fetches the MYA backup files will need to be able to handle a much larger set of data for all instances of MYA, assuming that it is designed to meet our 25,000 updates per second system wide requirement. Using the previously described test results indicated that MYA will generate up to 17.5GB of zipped backup files per day, which translates to an annual accumulation rate of over 6TB.

The MYA channel configuration tables differ from the history data tables in that they may be modified over time with more the just appending new data. Therefore regular complete backups of these tables must be made. This poses little burden on MYA or our computing environment as these tables are small. The mysqldump utility will be run at some regularly scheduled interval to take a complete snapshot of the MYA configuration tables. The output will be zipped and stored as directed by system administration staff.
7 Post Study Findings

There has been additional knowledge gained since the original feasibility study for archiving directly into a MySQL database\(^1\). This section serves as a supplement to the cited document, providing additional insight into the MYA design described in this document.

7.1 Timestamp Storage

The study described the timestamp of a channel event as two fields in a database table, which was a four byte value for the UNIX time and a one byte fraction of the second. The two separate fields proved to be too costly in data retrieval times. A better design was to have one field for the entire timestamp, serving as the table’s primary key. Sorting and performing arithmetic on a one piece timestamp is advantageous, unfortunately at the cost of disk space requirements. MySQL does not have a five byte integer data type. I tried using character data types, but the handling of the timestamp was cumbersome and inefficient. I settled on the 64-bit timestamp described in section 2.3.1.4, which does increase the disk footprint of the channel history set.

7.2 Prepared Statements

The design of choice was the one called the “Many Table” design, where each channel is stored into its own database table. One disadvantage of this choice, as described in the study, was not being able to make use of the MySQL prepared statement interface for channel event insertions. The reasoning was that there would have to be one prepared statement created in the server for each channel, multiplied by the number of database worker threads. The multiplicative factor comes from the fact that prepared statements are managed by the server per connection, and each thread has its own database connection. Although this is true, it turns out thousands of prepared statements are not so bad after all, and the multiplicative factor can be avoided all together.

The design proposed for MYA does create a prepared statement for each EPICS channel being archived. The number is of the order of 10,000 for a MYA instance. The multiplicative factor is removed by having one work queue per thread instead of one work queue for the whole instance. Therefore a channel’s updates can always be queued to the same thread, allowing for just one prepared statement for the channel. The chart below shows the advantage of using prepared statements and multiple queues, versus standard SQL and a single queue. The numbers following the run types listed in the legend are the number of worker threads. The software enforces a queue limit of 2 million, therefore archiving is temporarily suspended when the limit is reached, until the queue size is reduced back down to zero. Three of the test runs exhibited the queue overflow behavior for runs receiving 8,000 channel events per second. As can be seen from the chart, the multi-thread/queue prepared statement technique out performed the others significantly, when appropriately ignoring the queue ramp down in the other runs. It appears that the 10 thread/10 queue run almost can handle 8,000 channel updates per second, compared to the approximately 4,500 updates per second achieved with the standard query and single work queue.
7.3 Key Cache and Memory

The study included trying different amounts of memory allocated to the MySQL key cache. This is the amount of memory set aside by the server for the sole purpose of caching table index blocks. Having plenty of memory reserved for this purpose is normally very beneficial, and is in fact what rescued CZAR from very poor fetch times. MySQL documentation states that 25% of memory is a reasonable amount to allocate to a host which is mostly dedicated to the database. The MySQL server keeps statistics that allow you to determine how much of the cache is utilized and the percentage of times an access needs to go to the index file instead of memory. Note the index file can be still cached by the operating system. The tests cited throughout this report worked with the server’s default 8MB key cache, since I did not think caching would be a big factor in the many table design. On closer inspection I saw that 100% of the cache was utilized and index files were accessed about 50% of the time, which is far worse than the recommended 1%. I performed some test runs with a cache size of 512MB and found that MySQL utilized over 90% of the cache and the index file read rate was well less than the recommended 1%. That sounds really good, but unfortunately the MYA performance was significantly worse. I don’t believe that the key cache can ever have a bad effect in of itself. The only problem it may pose is taking memory away from other things, like data file caching. My conclusion from this result is that the test platform could have used more memory. Taking memory away from the operating system caused poor performance, but the server’s statistics showed it was making good use of the key cache. The oparsrv1 host was configured with 8GB during my tests. I believe maxing out at 16GB will be beneficial for this design.
7.4 Delayed Insertions

The original study indicated great performance advantages to using the MySQL `INSERT DELAYED` method for adding data to the channel event tables. The MYA design in its current state does not benefit as much from this feature. During the study, there was one work queue, meaning waiting for an insertion to complete stalled all database insertions. A many queue design keeps insertions going. The study also focused on two very different database configurations. The design that stuffed the channel events into one table, per data type, benefited greatly from the delayed insertion, as the insertions from many channels tended to log jam at the data table. Still, the many table design benefited significantly during the study. The many queue design utilized in the MYA design described in this document keeps insertions moving and may not need the overhead of the delayed insertion MySQL feature. The MySQL documentation thoroughly describes how the feature works and when it is likely to be advantageous. It also warns that there is significant cost to using the feature, so benefits may be offset by cost in some scenarios.

A set of tests were performed to measure the effectiveness of the two insertion techniques, as a function of the number of Scribe threads employed by MYA. Since this analysis came just prior to the release of this document, the test runs were not as long as necessary to make absolute conclusions. Further analysis should be done if the MYA design is selected for development. One other thing the tests showed was that the optimal number of Scribe threads is now larger than what was determined before the prepared statement interface. During the study, more than ten threads typically had a detrimental impact.

The chart below show seven different queue growth runs. Each run was performed with 10,000 channels firing at once per second, with queue size sampled over a fifteen minute period. The runs are named like 5D and 15N. The number represents how many Scribe threads were utilized and the character stands for either Delayed or Not Delayed. The lines are drawn in solid for delayed insertions and dashed for those that were not delayed. The colors are matched so that the 5D and 5N runs have the same color. Don’t bother trying to read this chart if you don’t have it in color.

The result shown in this chart is that 15 threads is the best choice for the delayed insertion technique and the 25 and 50 thread runs are about equivalent for the standard table insertions. There does not appear to be significant difference between the 15D and 25N runs. Much longer runs need to be made to determine which insertion technique and what number of threads is optimal.
7.5 Disk Block Matching

The MySQL server may need to be tuned to match the host system’s native disk block size. The myisamchk command output shows some block size equal to 1024, but it is unclear what this refers to. Some investigation into block size matching may yield better performance.
8 MYA Connection Management

This section describes the actions taken by MYA with regard to connection and disconnection events for EPICS channels. The impact on the MYA database is the focus of the descriptions. For this discussion there is no distinction between discontinuities caused by disconnection events, MYA shutdown, or channel stop requests.

EPICS does not provide a timestamp with a disconnection event, therefore the time from the MYA host system will be used for all types of discontinuities. A time allowance will be added to the local timestamp to account for any time synchronization error between the channel’s source computer and the MYA host, as it is better to fudge the discontinuity a few seconds into the future than have it marked at a time appearing before some already collected data. A study of the network synchronization of clocks will be performed to characterize the time delta.

8.1 MYA Startup

The execution of the MYA program is a special case of a connection event. The MYA software will read from the MYA database the last real data point’s timestamp for each channel that is to be monitored, skipping the discontinuity that must be the last thing stored for disconnected channels (if its not there MYA will add one of type unknown).

The MYA needs the timestamp to detect the situation that the first connection point has already been stored in the database.

8.2 Connection Event from EPICS

MYA will set a flag to indicate the next point received from EPICS is the first point of a connection span. The EPICS library is nice enough to always give you the current value of a channel upon connection. There are four scenarios to consider when the first point arrives, but note that no explicit connection indicator is ever stored into a channel’s database table.

8.2.1 New data point has a timestamp of zero

This occurs when the EPICS server on an IOC does not have an update to a channel since the EPICS database was loaded on the IOC. This data point is meaningless and will be discarded. Note that zero seconds past the EPICS epoch is not zero in the UNIX domain.

8.2.2 New data point is a repeat of previous point

This occurs when the EPICS server has run continuously since the MYA disconnection was observed, and the value of the channel has not changed. The MYA software will match the stored “previous point” with the new point to detect this scenario. Both the disconnection event already stored in the MYA database and the new data point will be discarded because this scenario means there really is no discontinuity of known states of the channel over the time indicated by the stored discontinuity.

8.2.3 New data point occurs before stored disconnection

This unlikely scenario occurs when the EPICS server on an IOC gets an update to a channel, but could not get the update to the MYA due to network difficulties. This
situation invalidates the timestamp on the disconnection event, but does not invalidate the event all together. There still is a discontinuity between the “previous point” and the new data point. The action here is to modify the disconnection timestamp to be in between the two data points, and the new data point is stored in the MYA database.

8.2.4 New data point occurs after the disconnection
This is the normal situation. The new data point is simply stored in the MYA database.

8.3 Disconnect Event from EPICS
When a disconnect event is signaled from the EPICS client library, create a disconnect entry in the MYA database with the present UNIX time as the timestamp. Store it in the database only if the last thing stored was not also a disconnect event. A fudge factor of 1.5 seconds is added to the timestamp to make sure there won’t be a timestamp incompatibility due to the unsynchronized clocks on the MYA workstation and the IOC.

8.4 MYA Shutdown
Act as though EPICS disconnect events had been received, although the reason code will clearly show that the discontinuity introduced is due to operator requested shutdown. The same thing is also done when the archiving of individual channels is ceased by external request.

9 MYA Logo

---

i EPICS Channel Archiver – Software Requirements Specification – Version 2.3; by Christopher Slominski.
ii Archiving Directly into a Database – by Christopher Slominski – January 5, 2006. (cjs@jlab.org)
iv refer to the ADM User's Guide and ADM Developer's Guide

37