ALGORITHMS AND DATA STRUCTURES FOR THE EPICS CHANNEL ARCHIVER

J. Rowland, M.T. Heron, S.J. Singleton, K. Vijayan, M. Leech,
Diamond Light Source, Oxfordshire, UK

Abstract

Diamond Light Source records 3GB of process data per day and has a 15TB archive on line using the EPICS Channel Archiver. This paper describes recent modifications to the software to improve performance and usability. The file-size limit on the R-Tree index has been removed, allowing all archived data to be searchable from one index. A decimation system works directly on compressed archives from a backup server and produces multi-rate reduced data with minimum and maximum values to support time efficient summary reporting and range queries. The XMLRPC interface has been extended to provide binary data transfer to clients needing large amounts of raw data.

INTRODUCTION

This paper investigates the architecture of the EPICS Channel Archiver and describes the changes made to improve performance and usability at Diamond. Other database designs were also considered as part of the upgrade progress and for intermediate processing of decimated data, and this information is summarized as a reference for future archiver developments. Information about the data structures used in external storage is not always available in user documentation, but it is essential to consider these together with hardware capabilities and query patterns when good performance is required.

Problem Statement

The EPICS Channel Archiver (Archiver) records data from N channels, each producing samples at a different rate. We assume that timestamps are monotonic, and that samples arrive sorted in time but unsorted by channel. The goal of the Archiver is to manage this data and fulfil queries in a timely manner to support operations. The most common query returns samples from M << N channels between times T0 and T1, so query results exhibit the opposite locality of reference to sample insertion. If the most common query returned samples from all channels in a small time range then there would be no mismatch between insertion and retrieval and the problem would be trivially solved; for example a sequential logfile of samples with a sparse index of timestamps would suffice.

Locality of reference is important because it determines how to find the next sample in a query; sequential samples can be iterated without traversing an intermediate data structure. The problem is compounded when the storage medium penalises random access. This is the case for all common systems, as RAM, solid-state disks and hard disks transfer data in blocks. Hard disks have the extra problem of mechanical latency when positioning the head.

Channel Archiver at Diamond

The important figures for Diamond are presented in Table 1. The Archiver is split across multiple engines for functional isolation and to utilize multiple processor cores. Diamond also operates a redundant system with two complete Archiver servers and disks to ensure high availability, so every component in the table is duplicated.

Table 1: Archiver Parameters

Data Size	12.5 TB
Data Rate	3 GB/day
Index Size	23 GB
Archive Engines	39
EPICS Channels	148819
Server RAM	24 GB
Server CPU	2x Intel X5670
Server Cores	24 (HT)
RAID Storage	36 TB
RAID Cache	1GB (write-back)
RAID groups	5

DATA STRUCTURES

Chunked Arrays

An array contains a single type of element, and supports appends and random reads. An array is contiguous in storage so that element addresses can be calculated directly, but resizing involves allocating new storage and copying the whole array. Maintaining contiguity whilst allowing for growth is possible using the 'dynamic array' where the array is resized by a constant multiplicative factor rather than one element at a time. This gives amortized O(1) append complexity, and is commonly used for in-memory arrays such as the C++ std::Vector. Drawbacks are memory fragmentation, unpredictable delays due to copying, and the need for increasing amounts of space at the end of the array.

An alternative is the chunked array, where the array is not copied on resize and is no longer fully contiguous. Storage is allocated in chunks; now element addresses must be mapped to chunk locations using an index. This

is the storage mechanism used by the Archiver and other scientific data stores. The Archiver also chains chunks together in a linked list so that the index is only touched to find the initial chunk of a query. One common file format supporting chunked arrays is HDF5, used for beamline experimental data at Diamond [1].

The Archiver has a minimum chunk size of 64 samples. As new chunks are added to an array the chunk size is doubled up to a maximum of 8192 samples. This reduces wasted space for small arrays and at the end of chunks whilst ensuring that large arrays have reasonable runs of contiguous storage.

The Archiver also partitions array storage by date, into weekly data directories. Once written, partitions are immutable, making backups simple as old directories do not have to be scanned for updates.

R-Tree

The Archiver uses an R-Tree as the chunk index. An R-Tree is a sorted search tree optimized for efficient range intersection queries. Each tree node has M children; having large nodes reduces the depth of the tree and the number of seeks required to complete a search compared with a binary tree. At Diamond the default M value of 50 has been increased to 200 to reduce the depth of the Master Index.

The keys in the tree are timestamps and the values are chunk offsets. As the archiver does not allow overlapping chunks to be stored in an array, the index structure could be replaced with a B-Tree without loss of functionality. This is relevant as there are many robust B-Tree implementations available offering useful upgrades such as compression and file locking [2].

Hash Table

The Archiver contains many channels, so the first level of index is a chained hash table with linked lists mapping from channel names to chunk indices. The default table size of 1009 entries resulted in 150x overfilling for Diamond channels, and the linked list nodes are spread throughout the index file, making lookups expensive. An option was added to the ArchiveIndexTool to adjust the hash table size and the Master Index was rebuilt using a prime near 300000, improving channel retrieval performance.

Master Index

The Hash Table and R-Trees for an archiver partition are stored in an index file in the partition directory. The partitions are joined by merging the individual index files into the Master Index. The data type used to store file offsets was increased from 32 to 64 bits to allow the size of the Master Index to grow above 2GB. This increases usability by presenting the user with a single index for all data; previously the user was responsible for splitting queries across multiple indices. Also, the interpolation algorithms in the server only produce consistent bin boundaries when retrieval is from the same index.

HARDWARE

Diamond uses a magnetic disk system in a RAID5 configuration to store Archiver files. The Archiver services all queries from the on-disk arrays, so samples must be written soon after they are received to support timely control room diagnostics. Samples are stored in circular buffers in RAM, and written every 30s. This write operation appends samples to every chunk that has been updated in the last period, and chunks are scatted across the disks. This is illustrated in Fig. 1; numbers in the matrix are disk offsets; shading indicates write order. The write pattern is not contiguous.

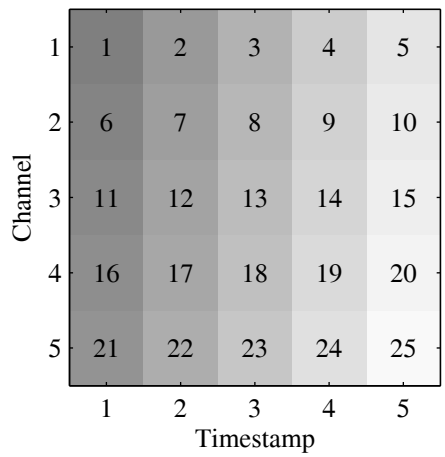

Figure 1: Archiver Write Order.

A typical magnetic disk can perform 100 I/O operations per second (IOPS) and has a sustained sequential transfer speed of 100MB/s. A disk seek costs as much as transferring 1MB of data. Each channel update costs one IOP. Therefore we can service approximately 3000 channel updates every 30s, assuming that none of the array chunks are sufficiently close together on the disk for their writes to be merged into a single IOP. This demonstrates that the key to Archiver performance is servicing write requests. RAID5 configurations typically have reduced random write performance due to the need to update parity bits, but spreading the archiver partitions across multiple RAID groups increases random write performance linearly with the number of RAID groups.

Write-Back Cache Controller

The RAID controller in the Diamond Archiver server has a battery-backed write-back cache with 1GB of RAM, enough to store 8 hours of data. Increasing the write period allows small writes to be merged, greatly increasing throughput. Figure 2 shows the write pattern after re-ordering by the cache controller.

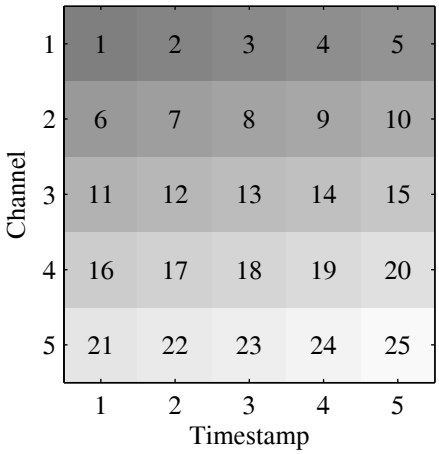

Figure 2: RAID Cache Write Order.

Table 2 shows the I/O utilization reported by the Linux tool IOSTAT when running the full Diamond archiving workload on three systems with different RAID controllers. 100% write utilization indicates that samples are discarded. Old is the previous production archiver, New is the current production archiver with upgraded cache controller, Desktop is a machine without a write-back cache controller.

Table 2: I/O Utilization

Hardware	Write%	Read%
New	3	75
Old	40	100
Desktop	100	100

RETRIEVAL

The Archiver query interface is exposed through the XMLRPC language-independent remote procedure call. This has proved pleasant to use from a variety of platforms including VB.NET, Matlab, Python and Java. However some users retrieve a large number of raw samples from the Archiver to produce multi-year reports and found the performance disappointing even after the server and disk upgrade. The XML encoding of arrays of EPICS datatypes was the limiting factor, these were packed as arrays of structures with the field names in ASCII for each sample, leading to a large space and time overhead. A new XMLRPC retrieval type was developed returning the EPICS sample array as a structure of arrays, one for seconds, nanoseconds, value, status and severity, each array packed into a byte buffer in native little-endian format and base64 encoded. Table 3 shows the performance improvement, the NFS method shows the performance of direct file access. The number of samples per request was still limited to 10000 as the RPC result is assembled in memory before sending to the client, this has since been increased to 1M samples to reduce the overhead of the RPC roundtrip.

Table 3: Retrieval Method Peformance

Method	Overhead	Time
NFS	1x	10m20s
XMLRPC	20x	112m7s
XMLRPC Base64	1.3x	26m

COMPRESSION

At Diamond backups are compressed with tar and gzip, and typically achieve a 4:1 compression ratio. Some of this is accounted for by partially empty chunks at the end of each array partition, but the low entropy of sorted EPICS time series data makes it an ideal candidate for compression. Figure 3 shows the sparsity pattern of the difference between adjacent samples for real double-precision data. High-order bytes of the timestamp corresponding to days and months, status and severity words, and padding bytes put in to enforce aligned memory access all rarely change.

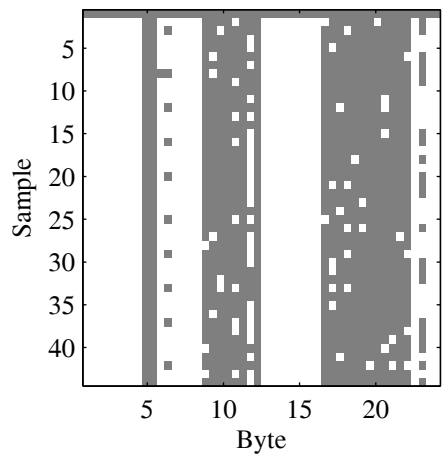

Figure 3: Differences of Adjacent Samples.

Compressed tar files are not seekable and cannot be read by the Archiver tools directly but libarchive [3] can stream decompressed data from a tar file without extracting to a temporary directory, this is used at Diamond to perform complete scans of the archive. Every sample can be read in a few days directly from compressed backups. This was used to feed the decimation tools discussed below.

Another option for direct access to compressed data is the squashfs [4] file system. This uses gzip in blocks to create a seekable compressed file system image that can be mounted by Linux. Using this to compress older data files will effectively quadruple the capacity of the Archiver storage without any software development effort. Squashfs is part of the Linux kernel from version 2.6.29.

DECIMATION

Many queries request decimated data from the archiver, and this decimation is performed on-line, traversing every raw sample on disk in the time range. As part of the upgrade program a method of pre-calculating these decimated queries was investigated. The required performance was achieved without making use of pre-decimated data, but the method is described because the data structures are more generally useful and offer an alternative to the chunked array store.

The data structures for decimated data were based on Hypertable [5], a write-optimized sorted key value storage engine developed to run on commodity hardware. Hypertable has previously been considered as a storage engine for the Archiver by INFN [6], and is itself inspired by the Google Bigtable [7] database.

Hypertable

Hypertable eliminates random writes through the use of a RAM cache. The key components of Hypertable are as follows:

- Sequential logfile for durability,
- RAM cache for data re-ordering (MEMTABLE),
- Periodic writes to immutable sorted disk tables (SSTABLE),
- Periodic external mergesort,
- Index of tables.

All operations are appended to a logfile which is periodically flushed to disk. In case of system failure, pending operations can be replayed from the logfile. Key and value pairs are maintained in a sorted data structure in RAM known as the MEMTABLE, and written to immutable sorted lists known as SSTABLEs (sorted string tables) once the RAM table is full. This provides similar functionality to the write-back RAID controller, but in software only. Keys are located in SSTABLEs by a B-Tree index.

In summary Hypertable provides a write-optimized structure to maintain partially sorted data on disk with the degree of sorting limited by available RAM. To improve sorting, SSTABLEs are periodically merge-sorted together. Typically the maximum size of the SSTABLE must be limited to prevent full-disk sorts. For EPICS time series data a suitable sort key is the (channel, timestamp) pair. This will ensure that channel samples are stored with good locality on disk for retrieval.

The decimation system uses a simple implementation of the Hypertable scheme. The sort key is (decimation rate, channel name, binned timestamp). The MEMTABLE uses the Berkeley B-Tree in-memory database. Raw Archiver samples are read from the compressed backups. If the MEMTABLE contains a binned sample containing the timestamp, the sample is added to the bin, otherwise a new bin is created. Once the MEMTABLE is full, it is saved to disk as a list of key value pairs. Once all samples have been read, the SSTABLEs are merge sorted into a single large table, and a sparse index is created containing the file offsets of each channel. The final table and index are compressed with squashfs.

Decimating all channels at 10 minute, 1 hour and 1 day intervals results in a compressed data size of 62 GB. An XMLRPC server provides the same interface as the original Archiver. Retrieval performance for long decimated queries is improved in proportion to the decimation interval.

SUMMARY

The required performance and usability improvements to the EPICS Archiver at Diamond were achieved without major changes to the software, but it was essential to consider hardware and software together to achieve this. Table 4 shows some benchmarks of the old and new systems. The DI benchmark returns decimated data for a week from a few hundred channels, the VA benchmark returns raw data for years from a few 10s of channels.

Table 4: Benchmarks

Hardware	Test	Time
New	DI	30s
Old	DI	3m30s
New	VA	8m30s
Old	VA	110m30s

For future Archiver developments the Hypertable combination of logfile and in-memory queryable cache offer a way of implementing an efficient and durable storage engine without requiring a hardware RAID controller.

REFERENCES

[1] HDF5, http://www.hdfgroup.org/HDF5

[2] Berkeley Database, http://www.oracle.com/us/products/database/berkeley-db

[3] http://code.google.com/p/libarchive

[4] http://squashfs.sourceforge.net

[5] http://hypertable.org

[6] Mauro Giacchini, Hypertable Archiver, http://www.lnl.infn.it/~epics/joomla/archiver.html

[7] http://labs.google.com/papers/bigtable.html