Lessons Learned from Managing a Petabyte

Jacek Becla*
Stanford Linear Accelerator Center
2575 Sand Hill Road, M/S 97
Menlo Park, CA 94025, USA
becla@slac.stanford.edu

Daniel L. Wang*
Stanford Linear Accelerator Center
2575 Sand Hill Road, M/S 97
Menlo Park, CA 94025, USA
danielw@slac.stanford.edu

Abstract

The amount of data collected and stored by the average business doubles each year. Many commercial databases are already approaching hundreds of terabytes, and at this rate, will soon be managing petabytes. More data enables new functionality and capability, but the larger scale reveals new problems and issues hidden in "smaller" terascale environments. This paper presents some of these new problems along with implemented solutions in the framework of a petabyte dataset for a large High Energy Physics experiment. Through experience with two persistence technologies, a commercial database and a file-based approach, we expose format-independent concepts and issues prevalent at this new scale of computing.

1. Introduction

Today, most large databases reside in government and university laboratories. The largest resides at the Stanford Linear Accelerator Center (SLAC), a national laboratory operated by Stanford University. Currently, the main focus of the laboratory is on BaBar, one of the largest operating High Energy Physics (HEP) experiments. In its fifth year of data taking and with over a petabyte of production data, BaBar continues to actively refine approaches to managing its vast amount of information. The experiment's data set is expected to continue to grow rapidly in the next several years.

This paper presents problems and solutions in petascale computing, drawing on experience with the BaBar data store from the perspective of building, deploying, tuning, scaling and administering. The evolution of the system through two major design iterations is discussed, presenting a unique perspective on large data set challenges. The first system utilized a commercial Object Oriented Database Management System (ODBMS), successfully serving a set of complex data to an international collaboration. The second system replaced the database with an open-source, file-based object persistence, improving many aspects of the system, but bringing its own challenges and issues.

Chapter 2 explains why HEP is so data intensive, discussing database related needs and challenges. Chapter 3 describes how these needs and challenges were met using an ODBMS-based approach. Chapter 4 describes a simpler, home-grown approach. Chapter 5 highlights the essence of persistent technology independent experiences, problems and commonalities. Finally, chapter 6 summarizes the paper.

2. Requirements for HEP computing

HEP experiments often focus their research on one or several types of *events* that are very rare. Such studies are highly statistical: these events are usually generated by colliding other particles together, however due to their rareness, thousands or even millions of collisions are needed to generate one "golden" event—the more golden events, the more precise the measurements. In addition to the "real" data produced by colliding particles, an equally large data set must be simulated to understand the physics, background processes, and the detector. In studying the asymmetry of matter and anti-matter in our universe, BaBar has registered over 10 billion events and simulated nearly the same amount since its inception in 1999.

Finding these events usually requires hundreds of iterations of study, and thus requiring long-term data persistence. Such studies tend to produce extremely large data sets by today's measures. Today's HEP experiments are expected to generate a few petabytes in their lifetimes, while the next generation experiments starting in 2007 at

* For BaBar Computing Group

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Proceedings of the 2005 CIDR Conference

CERN (Large Hadron Collider, or LHC) are likely to reach 20 petabytes.

The problem of *storing* so much data is only the tip of the iceberg. Analyzing data involves reading small chunks of data, each sized in the hundreds of bytes. Searching is done iteratively, narrowing the set of candidates but increasing the level of detail each time. Analyses are performed simultaneously by a large collaboration of physicists, reaching hundreds or even thousands, usually working from their local institutes world-wide.

To perform analysis in a timely fashion, data needs to be efficiently managed. Current technology would require three years for a single node to finish a single sequential scan of one petabyte. Therefore, good data structures and careful organization are most crucial, followed by raw performance from powerful servers. A huge data set is large in every dimension: millions of files, thousands of processors, hundreds of servers. This leads to many challenging scalability issues, discussed in more detail throughout this paper.

2.1 Data processing

Data processing in BaBar is broken into several independent production activities, as shown in Figure 1.

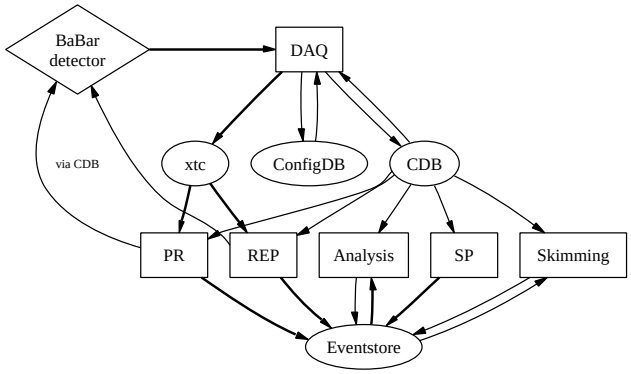

Figure 1 Data flow

The *Data Acquisition System* (DAQ) is an example of an activity that requires immediate response time and round-the-clock reliability, but only limited concurrency and throughput. Since storing all data produced by the detector would not be feasible, fast filters reject most collisions, accepting only about three hundred out of millions of events per second. Each event is stored in a home-grown bit-packed form called *xtc* and is about 30 KB in size. Each xtc file stores a single *run* which is a block of events taken with similar detector conditions. A typical run contains about a million events.

Prompt Reconstruction (PR) reads these xtc files, reconstructing each collision in quasi-real time and persisting the results. It is a high volume activity, requiring hundreds of parallel processes running continuously to keep up with the data acquisition. As an important source of feedback for detector tuning, it is also a low latency activity. Turnaround time has a direct effect on data quality and must be kept under a few hours.

A similar activity called *Reprocessing* (REP) reprocesses (reconstructs) all of the data taken since the start of the experiment using the latest software with the latest algorithms. It is done once a year on average.

Simulation Production (SP) is tasked with generating the simulated data mentioned above, producing about one event per real event. Unlike PR and REP, each process generates a different run, which can be stored separately from others. The main challenge is distribution: production of SP data is done at many sites around the world. Comparing simulated and real data is the basis of all HEP analyses, including BaBar.

To simplify the process of selecting data, and to spare users from repeatedly filtering the full data sample, all production data is "skimmed" (filtered) into streams. *Skimming* is CPU intensive by nature, requiring heavy analysis to perform selection. It reads data sequentially, applying analysis to select events and write them out into over a hundred separate streams, each defined by different sets of selection criteria. Each event may pass filters for multiple streams, thus forming overlapping sets. Skimming utilizes around 2000 processing nodes to satisfy its heavy throughput and low-latency requirements. The massive aggregate I/O makes a highly parallel system crucial. Other challenges include its unfriendly I/O load of small-grain reads and writes, its event duplication problem, and contention with user jobs– skim shares resources with the user computing pool due to costs. Production activities other than skimming run in their own isolated computing environments.

2.2 Data mining

Physicists need a single, integrated data source to do analyses, their searches for golden events in a haystack of collisions. The main challenge for the data store is to keep up in a cost-effective way with needs of thousands of data-hungry jobs sparsely reading small objects. It is followed by unpredictability and varying load – jobs are run in a completely unpredictable way with no easily discernible data access pattern. A high level of availability is also required. Any outage is disruptive for hundreds of users, so downtime has to be kept to a bare minimum. In addition to the centralized data store, users need their own persistent space for testing and storing intermediate results.

Finally, BaBar physicists write their analysis code in C++, so the computing environment must support accessing and querying data in that language.

2.3 Data distribution

BaBar is a large collaboration of over 600 physicists in 10 different countries. Many countries contribute by providing dedicated computing resources for the project. For this reason, a sizable fraction of data population and

mining happens outside SLAC. Although it would be simpler to concentrate all activities at one location, activities are distributed, partly for political reasons: BaBar hardware sponsored by local governments must stay at their local sites. Initially, most activities were done entirely at SLAC, which also served as the only analysis center. However, as the involvement of external sites increased, many production and user activities were moved to offload SLAC:

- PR and REP were moved entirely to Italy.
- A large fraction of skimming was moved to Germany and Italy.

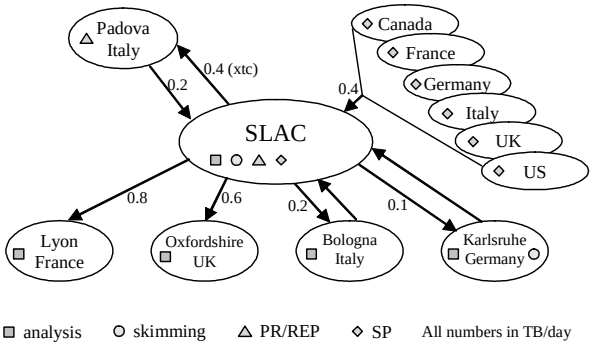

Figure 2 BaBar's data distribution

Currently, 90% of SP data is produced at 23 sites outside of SLAC. The number of large analysis centers increases almost every year, and is currently five, four of which are located in Europe.

Having major data production activities scattered around the world complicates the task of managing both local and world-wide data flow. All data exchange happens over a WAN, not via tape, and terabytes of data are moved daily through dedicated cross-Atlantic links. SLAC itself is connected with an outside world via two links: one 622Mb/s (OC-12) link to the Energy Sciences Network (ESNET) and one 1Gb/s link to Internet2. At any given time, between 30-50% of their capacity is used.

3. First generation: meeting the challenges

The persistency system for the BaBar experiment is required to store and provide access to multi-petabyte data set in a cost-effective way. The main challenges are data organization, scalability sustaining performance, solving concurrency issues, reliability, administration and distribution.

3.1 Organizing bulky data

BaBar's data is logically organized in several *domains*, with no direct cross-references. One of the key domains is the *eventstore*, the domain discussed most in this paper. This domain stores the heavily publicized petabyte of data bulk (events).

Events are organized into *collections*, which may contain either events, or pointers to events. With collections numbering in the hundreds of millions, various production activities maintained dedicated *bookkeeping* systems, which were consolidated and centralized in the second generation computing model.

Reconstructed events carry much more information than their corresponding bit-packed versions, and are therefore split into several *components*, each of which carries different level of details. Components vary significantly in size leading to important implications on read performance.

Component	***Size [KB]***
raw	100
rec	150
mini	10
micro	4
tag	1
(overhead)	3

Storing the two largest components, raw and rec, is expensive due to storage and access costs, especially when analysis rarely needs them. The multi-level, multi-component access model also proved too cumbersome and after a few years BaBar shifted to a simpler model involving only micro, mini and tag.

3.2 Choosing a storage technology

Storing petabytes of data on disk remains prohibitively expensive compared to tape in terms of operating cost (power and cooling requirements), durability, and purchasing cost. Tapes do not consume power or generate heat when not in use, two increasingly important factors at data centers. To manage tapes we use IBM's Mass Storage System (MSS): High Performance Storage System (HPSS), storing over 1.3 petabytes of data on about 13,000 tapes managed by 6 StorageTek tape silos.

With such a large MSS system, a large disk cache is still essential to deliver data to jobs in reasonable time. SLAC's 1.3PB is currently backed by 160TB of disk cache. The disk cache is implemented on thousands of physical disks bound into large arrays for simplified maintenance. In the past, they were managed by VERITAS File System (VFS), but more recently by Sun's Solaris 9 UFS. Both systems support journaling, an absolute necessity when dealing with extremely large file systems.

An object oriented approach seems to be the most natural way to model HEP data, where complex many-to-many relationships are commonplace. As explained above, the persistency system has to also provide multi-petabyte scalability. None of the RDBMSes available in mid-nineties offered any of these features (and they do not appear to meet all these requirements even today). The decision to use an Object Oriented Database Management System – Objectivity/DB followed recommendations from a study at CERN [14]. ODBMS technology seemed

to best fit BaBar's needs of scalability, complexity and C++ bindings. Also, the thick-client thin-server architecture and distributed features seemed well suited to scaling, in contrast with the traditional, server-heavy RDBMS approach.

3.3 Integrating the system

The commercial ODBMS provided a powerful database engine including catalog, schema management, data consistency and recovery, but it was not deployable into a system of BaBar's scale without extra effort. Half a million lines of complex C++ code were required to customize it and to implement needed features that did not come with the product.

One of the major features added was an extended address space. An important requirement is that each analysis job should be able to access all data. The Objectivity/DB system allowed only 64K database files per federation (described later), and each job could access only one federation during its lifetime. Theoretically, this allowed storing many petabytes since the database files could be huge, but database files of that size (~10-20GB) were very impractical, especially for staging and distributing. After discussions with the vendor, the latter limitation of accessing only one federation was removed, allowing BaBar to implement *bridging* technology to glue data spread across multiple federations [11]. This allowed collections to be transparently accessed across many federations through a central *bridge federation*. A bridge federation contains bridge collections each containing a list of other collections and the federations in which they exist. Since Objectivity/DB transactions and local client caches could not span federations, implementing bridging also required implementing a complex transaction management system to transparently and automatically switch transactions as jobs change federations.

Due to BaBar's customized storage system, the data server provided by Objectivity/DB–Advanced Multithreaded Server (AMS) –was insufficient. Access to database files needed to be transparent whether they resided on disk or tape, so we worked closely with the company to re-architect the AMS. This involved splitting the product into three separate components: the AMS protocol layer, the logical file system layer (Objectivity Open File System, or OOFS), and the physical file system layer (Objectivity Open Storage Layer, or OOSS), of which we controlled the latter two [4].

To better utilize a precious disk cache, we worked with Objectivity on an API that would allow us to plug compression into their system. Data compression reduced the disk footprint by half, and client-side decompression kept server load under control.

Other features added on top of the ODBMS included clustering and placement strategies, specialized indices, and administrative tools.

Significant effort was spent tuning and optimizing the database system from the first day of data taking on [3]. With the detector's rapidly improving performance, it was clear that the system had to scale to unforeseen levels. A dedicated test-bed, occasionally requiring 500 nodes was established to tackle the performance and scalability issues. Tuning the system, scaling and understanding unforeseen bottlenecks required a lot of time, effort and thorough understanding of the whole system: operating system, client code, network, and Objectivity/DB including the AMS.

Finally, a transient-persistent abstraction layer was implemented early in the development phase to contain persistency-specific code. This layer isolated the bulk of the BaBar code from the details of persistency implementation with a small performance hit (~2%). With this fundamental decoupling, hundreds of users were shielded from persistency details, facilitating a reduced impact as the features described above were implemented.

3.4 Coping with server load

BaBar computing makes extreme demands on its servers. For instance, a single PR federation often needs to handle data injection from hundreds of nodes and sustain over 15MB/sec of write throughput. Data analysis routinely needs to deliver 200MB/sec to over 1,500 jobs. In one case, a single server (host) was observed to sustain a thousand open files and over 15,000 open TCP connections. Coping with such loads requires powerful and robust servers.

Clients access data remotely through the AMS provided by Objectivity/DB. The AMS was designed specifically to serve Objectivity/DB data, serving database pages directly with no caching. NFS and AFS are somewhat problematic alternatives. NFS often times out under heavy load, and AFS has several issues related to client cache synchronization and token renewal. Many small institutes prefer the simpler NFS access because of its low overhead under a light load.

AMS load depends on the number of clients, as well as their access patterns. One of the important "features" of Objectivity/DB is that it opens a TCP connection for each opened database file per client. In our environment that translates to tens, often hundreds of connections to the server per client. This large number of connections saturates a CPU which must linearly search a file descriptor table containing many thousands of entries to handle each I/O request. Tuning the AMS was limited by the Objectivity/DB protocol, but we had the freedom to do modifications in the OOFS and OOSS layers.

Improvements in the OOFS and OOSS layers included separating TCP file descriptors from database file descriptors, closing inactive files, and sharing file descriptors of the files opened by multiple clients. Due to thread synchronization around the single file descriptor table, we ran four AMS servers per host to efficiently

utilize the 4-CPU servers. Each quarter of a compute farm was then redirected to a different AMS server. Another important change was to increase the default limit of open file descriptors on UNIX from 1 to 8 K.

An automatic load balancer was implemented to efficiently balance the load and reduce "hot spots." It measures the load on data servers and redirects clients accordingly. While it is also capable of duplicating data, this feature is turned off to avoid excessive staging. It is designed to work with immutable data, which is sufficient for analysis. We have not found a reliable way to apply it to mutable data yet.

The lock server was another bottleneck, especially since each federation had only one lock server, and Objectivity/DB's implementation was single-threaded. Unfortunately, in this case, we had no access to modify it, so we resorted to less invasive techniques such as reducing the offered lock traffic (explained in detail later) and using faster hardware. Bridge technology (also explained later) further alleviated lock server contention.

3.5 Sustaining performance

HEP jobs require heavy computation; and are therefore expected to be CPU, rather than I/O bound—the average CPU efficiency should be above 90% for both data population and data analysis.

Even though the jobs did heavy computation, both data population and data mining dealt with a lot of I/O. Data population often produced over 2 terabytes per day, and data analysis often read 10 times that amount. In many cases, thousands of jobs competed for server time or disk bandwidth. To keep the farms CPU-bound, numerous measures were taken.

Not only were disks and servers strained by thousands of simultaneous jobs, but the high load also resulted in randomized access patterns at the disk level. This was often seen in data analysis, even though data placement and clustering algorithms provided spatial locality for data that is frequently accessed together. Turning off read-ahead in VFS alleviated the problem somewhat; however, the poor performance of small random reads on disk continues to be a major problem without an efficient solution.

Data population is an organized activity. Its heavy write bandwidth requirements were achieved by client-side cache size tuning and transaction length randomization such that clients flush data to disk only during commits and only a few clients commit at any given time.

Parallelism is another way to improve performance. During data population we parallelized event and run processing. Different events within a run were processed by different nodes. The optimal number of nodes participating in processing a single run was about 150; with larger numbers, startup and shutdown times dominated. Different runs were processed by independent farms, each farm had dedicated sets of client and server hardware. To enable this in PR, feedback calculations for the detector were extracted and run independently from the rest of PR.

In data analysis, parallelism is achieved with a batch system. Since all BaBar applications are single-threaded, each job can utilize only one CPU, so users may implement parallelism by initiating multiple jobs simultaneously. Each job process a subset of total data sample to be analysed and the results (selected events) can then be combined into a single collection. This is "quick and dirty" parallelism, but very convenient for analyzing large data sets.

3.6 Dealing with concurrency

Concurrent access to data is possible using transactions and locking mechanisms provided by the database engine. In Objectivity/DB, locking occurs at the granularity of a *container*. A container is a logical section of a *database*. Each database maps to a single file. A consistent set of databases forms a *federated database*. Each federation has a catalog (database metadata) and schema (definitions of the various object classes) and delegates lock and transaction handling to one lock server (may be shared).

Generally in PR, more than a hundred nodes process the same run and write to the same set of output streams. Each job writes to the same set of databases, but to its dedicated containers. The only shared resources are collection navigation metadata, and some internal structures maintained by Objectivity/DB, such as a database catalog or a page table for each database. Jobs contend for both these resources when creating collections, databases, and containers. Contention on collection metadata was removed by precreating collections in a single job. Contention on Objectivity/DB metadata was removed by centralizing the pre-creation of databases and containers and assigning them to jobs through a CORBA server [2].

Contention on collection metadata is also a problem in data analysis. Thousands of analysis processes access the data, often in competition with skimming, which itself heavily reads and writes the data. Internal Objectivity/DB resources are not an issue due to the relatively small volume of written data, but collection metadata is a bottleneck. It is organized in an efficient directory-like tree-structure, but the contention comes because many tree-nodes share containers. (Container opening has significant overhead, and databases can have a limited number of containers.) Rearranging the tree nodes into different containers and providing access to central nodes inside separate mini-transactions reduced but did not eliminate the contention. Eliminating it requires removing all dependencies on a central index when updating.

Introducing bridge technology and spreading data across many federations, lock servers, and database catalogs helped reduce lock contention, but the collection

metadata problem resurfaced in the bridge federation. For that reason, some users found it more convenient to bypass the bridge federation, and maintain their own bookkeeping of collection-to-federation mappings.

Other tunings were needed to improve concurrency, especially in PR farms. One such tuning involved minimizing container naming. Container naming involves an extra lock on another shared resource, a page table. Another tuning was reducing database name lookup (which requires an expensive catalog access) by caching id-to-name mappings and using ids whenever possible. The last important tuning was presizing containers to their expected final size. Growing a container requires locking the database's page table, and therefore is too expensive to be done simultaneously by hundreds of writing clients

Scaling to BaBar's heavy concurrency needs can be summed up in one directive: Minimize contention around shared resources. This means making updates as fast as possible, or planning ahead and pooling the updates into a single large update.

3.7 Availability

Data needs to be highly available. Since BaBar is a large international collaboration with users scattered across many time zones, there is no good time of the day (or night) to take the database offline. Unfortunately, software and hardware faults still happen, so BaBar's computing must have measures in place to minimize their impact.

3.7.1 Durability

All event data is potentially valuable–new algorithms can reveal insights in data previously thought uninteresting—so it must be backed-up. Production data is thus written to tapes as soon as it is generated. Disk faults are backed by tape, and the tapes containing the most demanded data are duplicated in multiple regional centers. Faulty eventstore tapes can be rebuilt from tapes containing raw detector data (xtc format), which are stored in exactly two places, SLAC and Padova, Italy. Frequently changing metadata is backed up to HPSS during scheduled weekly outages. This metadata includes database catalogs, collection metadata, database identifier allocation, and user access control information. It is relatively small, measured in tens of megabytes. On the other hand, user data is not backed up because it is considered scratch data for debugging and testing; it comprises less than five percent of the total data sample. The remaining data is immutable.

Large fractions of the event data are multiply duplicated in offsite analysis centers. This improves safety and pushes data closer to remote users.

At SLAC, major services like lock, journal, and catalog servers are backed up by uninterruptible power systems, but data servers remain prone to power outages. Unexpected power outages are the primary reason for seriously exercising our database backup system, which happens about once a year. Hardware problems often follow each power outage resulting in prolonged database outages. During one major power outage, damaged hardware kept the database system offline for over 24 hours.

3.7.2 Planned and unplanned outages

Some administration tasks require exclusive access to a federation, thus necessitating an outage. Killing thousands of jobs, many of which have been running for hours or even days is not a viable option and is used only before planned major outages. An *inhibit* system was thus developed to facilitate these administrative tasks. The inhibit system stalls all clients between transactions instead of killing them, using the fact that each BaBar application is required to commit its transaction periodically. Jobs stalled in this fashion do not keep any locks, which is enough for most administration tasks.

Short regular outages are scheduled weekly, to perform necessary administration operations like backing up metadata. The overall achieved uptime is about 98%, with most of the remaining 2% downtime caused by scheduled and unscheduled power outages. Bridge technology has helped to achieve the high uptime: administrators can take down one federation at a time while the rest (over 99% of the total data sample) are still available.

3.7.3 Platform unreliability

The problem of unreliable platforms appeared when we first bought a large batch of commodity machines to be used in PR. The problem turned out to be a vendor-specific hardware problem which caused random reboots (~4% of machines every day) and frequent hardware failures. This visibly reduced overall efficiency and forced us to look for new ways to manage recovery from crashes and the resulting orphan locks associated with unusable machines. Today, the number of crashes of commodity hardware used as clients is well under control, although still far from zero.

3.8 Using the data

Another interesting problem for large data sets is how to make all of it available and usable. The BaBar eventstore is not a closed archive—it is live data that is collected and served to physicists to support their cutting-edge research. This section offers overviews on how event data is published, accessed, and indexed.

3.8.1 Publishing

The system makes data available primarily via Objectivity/DB AMS data servers. Data comes from DAQ in the form of xtc files, which PR reconstructs to be inserted into the eventstore. From there, individuals and other BaBar groups access the data, often deriving additional event data to be inserted. Individuals read and write data in real-time, but the publishing of production data is often delayed by a few days. Data produced over a

week's time may take two days to be published due to contention with shared eventstore constructs, although the turnaround time remains impressive compared to previous HEP experiments. Since the BaBar eventstore experiences constant usage, lock collisions happen frequently during the two steps for publishing: attaching the db file and adding the collections to the index.

Partly because of this resource contention, BaBar data is distributed geographically. As a member of the BaBar collaboration, a physicist may access data via a variety of data centers, which are categorized into two groups, according to their capabilities and responsibilities. Tier A sites such as SLAC form the core of BaBar computing. Large scale analysis and processing is split up among these sites, and they are open to any BaBar collaborator. The entirety of BaBar data is available through Tier A sites. Tier C sites are much smaller, have much less data, and are almost exclusively managed and used by local users. Many of them still participate in producing simulated data.

3.8.2 Accessing

All production data is truly read-only, protected by setting appropriate permission bits on database files. Groups' and individuals' data is stored in private databases, providing writable scratch space. Custom C++ code implements access control as well, preventing inadvertent modifications of other users' data. Each user can freely read other users' data, but cannot update/delete it, unless explicit permission is granted by an administrator.

The Objectivity/DB product provides C++ bindings for making queries, along with a data definition language for describing persistent objects. Because of portability issues in writing persistence-specific code, the BaBar computing policy specifies that analysis code be kept on the transient side of the persistent-transient interface. To access persistent data, this analysis code is modularized and plugged into a framework system for controlling and providing access to event data. This Event Analysis Framework is modeled as a pipeline of modules which receive and process event data in turn. With the ability to reorder, reconfigure, enable and disable modules via Tcl scripts, users generally use published framework binaries, adding and removing event filters and customizing their jobs without recompiling.

Unfortunately, in the ODBMS implementation, the BaBar eventstore could not be accessed outside of the framework system. All the database customizations were done within the framework, and so no tools had been provided to manipulate the data outside of the pipeline model. Because of this limitation, physicists often extracted the data out of the eventstore to the ntuple format for simpler data manipulation.

3.8.3 Indexing

Eventstore data is not indexed. The most natural place to use indices would be for event selection (tag data). Usually, tens of attributes (out of over 500 possible) are examined for each event, which would require complex multi-dimensional indices. With these complexities, the strategy is to cluster these attributes into a separate component ("tag") and placing tag components for events in sequence.

It is well worth noting that most jobs that run on eventstore data spend most of their time analyzing sets of events for selection or statistics. Considering that jobs need to be crafted to avoid the 48-hour limit of CPU time, a single job would overwhelm the server if the "query" was processed there.

To combat the high computational cost of selecting events, the skimming activity filters event collections into many smaller subsets according to common characteristics. Every six months, all data is read and skimmed according to the latest requests by various analysis working groups. With this advanced pre-filtering technology, selection rates on the skimmed data exceed 30% where they were previously less than 10% or even under 1%.

While the eventstore database was free of indices, non-eventstore databases, particularly the Conditions Database (CDB, which stores detector conditions) [6], relies heavily on indices. The most common queries in the CDB are partial-range queries over two dimensions. So common were these queries that generic two-dimensional indexing implementations were too inefficient. Using certain characteristics of CDB data allowed us to build an efficient, customized B-tree algorithm.

3.9 Administration

Some say that 1 full-time person is needed to manage each terabyte of data [7]. For BaBar, the level of effort needed to maintain the petabyte system was less then three full time database administrators. Initially, administration required a lot of human interventions—too many by our measures. These interventions were slowly automated, reducing tedium and opportunities for user error, but the system continued to grow more complex with new features like bridge federations. Most of the biggest challenges of administration were touched on in the previous sections. They include (in order of importance):

- hardware and software failures
- metadata management
- data distribution
- manual load balancing
- lock collisions

3.9.1 Hardware and software failures

Hardware and software failures continue to be problematic. In a system with over 3000 physical disks, hardware failures are not uncommon. Even with enterprise server hardware, a disruptive hardware crash (e.g. disk array controller) still happens more than once a

month, on average. Disk failures are more common, averaging about 3.5 disks per month, but generally have a low impact because of RAID 5 redundancy. On the software side, AMS failures occur about once a month in one of over a hundred servers, generally as fallout from a new AMS feature.

3.9.2 Metadata management

Though bridge technology solved the address space problems, it also led to the proliferation of federations. With 120 federations in analysis, moving (“*sweeping*”) data from production farms to analysis federations was a daunting task. Another complication was a collection count that exceeded the original design by several orders of magnitude. This was primarily because of three factors which themselves were beyond the specification: (a) a large number of data streams, (b) a large number of skims, and (c) constantly improving detector and collider performance. With so many federations and collections, keeping their metadata consistent while data moved from production farm to analysis was a real challenge in terms of volume and concurrency.

Occasionally database files were temporarily or permanently unavailable. Because no simple mapping existed between database files and collections, users had no easy way of avoiding afflicted data sets. Sometimes a user’s job would crash after many hours, and the user would find that an unavailable database file was the cause only after extended help from a database administrator.

3.9.3 Data Distribution

Distributing data across multiple servers, federations, and sites requires dedicated tools. Within SLAC, the local data distribution strategy takes advantage of the HPSS catalog to minimize the actual copying of databases. Databases from PR, for example, can be published for analysis by simply updating the target federation’s catalog. By centrally managing database ids, database files could be quickly moved between federations without destroying external references in dependent databases.

World-wide data distribution of BaBar’s challenging data volume utilized grid technology. Shipping production data between SLAC and the site in Lyon, France, for example, was done through the Storage Resource Broker (SRB) [12]. BaBar has also been investigating the Globus Toolkit [5] for Simulation Production.

Wide data distribution also complicated data quality. External sites were not always up-to-date with the database system: uninformed operators sometimes removed journal files with “rm” and copied dirty database files. Without reliable and fast QA tools, these mishaps caused administrative headache.

3.9.4 Others

Due to frequently changing hot spots, data had to be almost constantly rebalanced to reduce bottlenecks and improve system performance. The automatic load balancing discussed earlier was only available in late 2003, so before that, data was balanced manually. Manual balancing was effective, but its heavy cost in database administrator effort was too large to sustain.

Finally, lock collisions were a constant problem. As mentioned before, collection publishing frequently conflicted with user access. Users also had locking conflicts with their own jobs. Stubborn locks left by dead jobs stymied many users until they referred to a FAQ or an administrator. With high turnover in students working on BaBar, the user pool needed frequent assistance.

3.10 Summary of experiences

Mammoth size and complexity requirements forced BaBar to look for innovative approaches to managing its data. Using a commercial database system provided a basic persistence model, but brought new challenges as well. BaBar was the first in the HEP community to use an object database, and globally alone in scaling to a petabyte.

With these unknowns and a changing set of requirements, BaBar built significant flexibility into its data management software. This foresight allowed the system to quickly adapt to many issues in performance, volume, organization, administration, and functionality. Though it was clearly overdesigned in certain aspects, the extra effort was an overall benefit. A few requirements were not anticipated, such as the high collection count. Thus the flexibility to accommodate those issues did not exist, and coping with them required reengineering during user operation. Designing the right amount of adaptability is difficult in any situation, and BaBar’s especially volatile analysis needs rewarded the extra effort.

Initial deployment was rocky, impeded by many scalability and performance problems, requiring many tunings and optimizations to stabilize the system. The original requirements were met quickly, but rapid expansion and constantly changing requirements kept the data store operating near the edge of its capabilities throughout the past 5 years of production.

BaBar’s vibrant research effort continues to demand more data, more quickly from the detector, as well as higher levels of data service. The first generation eventstore was undoubtedly a great success, providing storage and service throughput well beyond its original design goals. Data rates, for example, were several times higher than originally designed. Hundreds of users analyzed data in BaBar. Its complexity and size has put it beyond today's scalability frontier: in 2003, it was larger than the largest 200 relational databases combined, earning the grand prize in Winter Corporation’s TopTen Program (a survey of world’s largest databases).

Further details on this ODBMS-implementation can be found in [1], [2], and [3].

4. Second generation: refactoring

Experience is often considered the best teacher. When a project is finished, its developers invariably ponder how much better it could have been, had they known then what they know now. Thus, as BaBar's first persistence system matured and stabilized, thoughts began shifting from "how can we fix X?" to "if the system was like Y, then would X exist?" Research is necessarily forward thinking, and with an increasingly stable system, developers were able to design and implement BaBar's second computing model.

4.1 Motivation

Migrating BaBar's large computing system was not a decision made lightly, requiring many months of discussion by BaBar management and staff. The idea itself began circulating in 2002, soon after the LHC experiments at CERN decided not to use Objectivity/DB for persistence. Their choice was not based on the perceived technical feasibility, but rather on the uncertainty of Objectivity Inc.'s long term viability. Because the ODBMS market had grown far slower than expected, the company's sustainability in a difficult economic climate and over the LHC's operating lifetime (15-20 years) was deemed too uncertain. Following the LHC's decision had other benefits including: (a) greater reuse of skills for physicists switching experiments, and (b) reducing the number of systems to be supported.

As the only commercial software in a sea of 5 million lines of home-grown C++, Objectivity/DB was the only non-source-accessible component. Though support was excellent, their priorities and release cycle did not always align with BaBar's. Dependence on their libraries, for example, locked an otherwise open system in certain compilers and operating systems. Monetary cost was also an important factor.

Object databases are problematic for BaBar in other ways. The overhead of database semantics on usability and manageability was too high, considering that nearly all of the workload was on read-only data. Data seemed "locked away" from physicists, and was non-trivial for administrators to migrate between machines or setup on a physicist's laptop.

Finally, the existing transient-persistent abstraction layer within BaBar software made migration a practical possibility. Such a layer was crucial in mitigating any lock-in to a single persistence technology, and allowed enough freedom to consider alternatives.

4.2 Design goals

The second computing model aimed to select the best features from the original implementation and leave behind the most troublesome. Users still wanted a high performance, flexible, scalable, and unified computing environment, but disliked the overhead of network connectivity, database semantics, and the central point of failure. Yet the new design had to minimize user impact. BaBar was not going to interrupt researching the origins of the universe or give up its mammoth amount of existing analysis code.

BaBar wanted to eliminate dependence on commercial software, and so chose ROOT I/O [13], an open-source, almost BSD-licensed persistence technology that had wide acceptance in the HEP community. ROOT I/O lacked many standard ODBMS features, never claiming to be a "database," but early tests indicated that it could be adapted to meet BaBar's demands.

Improving data administration was another goal. Users found the array of database files representing their data unnecessarily difficult to manage. They eschewed the organization of database files as clustering constructs for their data, whose logical structure depended on collections storing references to events with references to data objects. The new computing model simplified the mapping, defining the one-collection-one-set-of-files model. This model also simplified exporting data and facilitated unconnected laptop analysis.

The last major goal was to reduce the complexity of the system. The same physicists confused by the need for the many files were also annoyed that the many data servers, lock servers, transactions, and database semantics seemed not to help but rather hinder their analyses.

4.3 ROOT I/O

ROOT I/O provides data persistence for the ROOT data analysis framework, and can be characterized by its light-weight, "raw" interface to persistent objects. Persistent classes are defined very similarly to Objectivity/DB, with the added benefit that member fields may be marked as non-persistent. This is a marked benefit when using the persistent classes directly, although users of the BaBar framework are shielded from persistence details. A key disadvantage, however, is ROOT I/O's lack of true object referencing (ROOT has since developed a limited form of object references, but BaBar does not use them).

A major benefit to using ROOT I/O for persistence is the availability of the data under the CINT C++ interactive interpreter used by ROOT. Interactive use was among the motivations to migrate, and it is hoped that its convenience will allow BaBar users to work more closely with their data. Interactive manipulation of ROOT I/O has been immensely useful to developers, but has also proven to require extra diligence in developing interactive-accessible code. As of this writing, interactive access is still very new to users, but it is hoped to become a disruptive technology that will energize and spur different ways of analysis. Still, there remains the danger of writing code too close to persistence—interactive analysis is necessarily aware of the details of ROOT. Finally, ROOT I/O persistence provides built-in data compression, encoding a batch of objects of the same class in a single

compressed unit. Compared to compression at the level of database files in the first generation system, ROOT provides compression at a finer granularity made possible by its unique model of clustering persistent objects.

4.4 Kanga: an abstraction layer above ROOT

To successfully migrate BaBar's existing analysis and processing software to a new persistence technology, significant effort was required. The Kanga system was implemented to fill-in the essential functionality that the previous ODBMS provided, and to replace the existing ODBMS-dependent layer of code.

The Kanga system provides its own persistence layer that implements object-references/pointers, basic schema evolution, and transient-persistent object binding on top of ROOT I/O. Because object-referencing operates on a low level in object persistence, client code had to be shielded from direct interaction with ROOT, so all accesses had to be made through the Kanga persistence system. Transient-persistent object binding was also integrated into Kanga where it was previously a separate abstraction layer from the ODBMS. Kanga is accessible inside the ROOT CINT environment, giving the interactive user access to its enhanced functionality.

It is interesting to note that although Kanga's object referencing system is limited, BaBar data can be modeled adequately. It is a characteristic of BaBar event data that objects of a particular event do not reference data in different events, so more flexible, more generic object references are not necessary.

Another key characteristic of Kanga is that it does not attempt to provide any database features. Data must be persisted in a very specific fashion, and ROOT files themselves cannot be updated, only appended. However, BaBar's model of data collection, processing, and access (read-only after data is persisted) was such that it did not suffer too greatly. In fact, the feature-light characteristics have yielded lower overhead and overall performance benefits in current practice.

One example of the tradeoffs of Kanga is in its management of collections. In the ODBMS, data from many collections were stored in a single database file. Kanga defines the model: one file contains data for only one collection. One can then manage collections easily as files in a file system. Data production can run in parallel, creating "sub-collections" which are later merged into much larger collections. Since the collections are merged, there are less collections overall, which makes management easier. Unfortunately, large collections, some exceeding 2GB, are difficult to manage by the end user. Jobs running over these collections exceeded the batch system's CPU limits, and more aggressive data quality efforts needed to mark some sub collections as "bad" due to discovered errors in their processing and production. To handle these cases and provide greater overall convenience, a new syntax for selecting collections was needed to select subsets of collections, shifting the problem of too many collections to a new problem of implementing and debugging a syntax for selecting parts of them.

4.5 A new data server: Xrootd

In considering the new computing model's persistence technology, ROOT I/O, client/server data access was an important issue. Data servers in the new system needed to reliably serve tape-resident data backed by local disk caches. The bundled data server, rootd, was insufficient for BaBar's needs, so it was thus re-architected into a more performant solution, using past experience with Objectivity/DB's AMS and the rootd source as references. The resulting effort is called Xrootd [8], [16].

Xrootd was built specifically to address some of the larger problems encountered with the AMS daemon. The AMS protocol specified one TCP connection per file per client. With clients typically accessing many files, and sometimes failing to close their connections, this caused not only server overhead, but administrative headache. AMS daemon restarts were highly disruptive, as the protocol had no inherent failover or fault-recovery provisions. The Xroot protocol uses only one connection per client, and has specific features for fault-tolerance and load balancing.

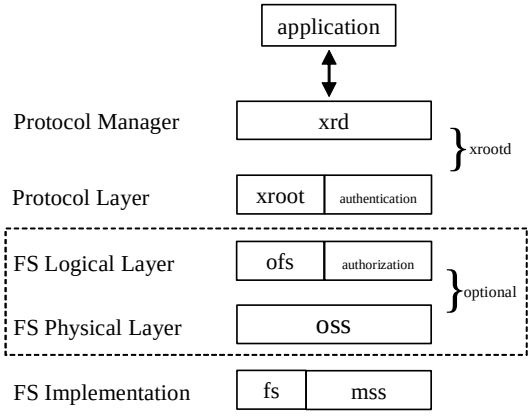

Figure 3 Xrootd architecture

The Xroot protocol utilizes a connection as an asynchronous pathway supporting multiple independent and overlapping operations. In this way, it uses each connection as multiple logical connections, removing the wasteful use of network connections. I/O for each independent operation can be segmented, preventing a large transfer from necessarily blocking other operations. Fault tolerance is facilitated through explicit *redirection* and *deferral*. Redirection allows the server to perform dynamic load-balancing as well as fault-handling. Deferral is a server-originated message that effectively pauses the client for a particular time period. It is used to allow clients to gracefully continue through server restarts or other maintenance, or to delay clients while data is

staged. Deferral allows server maintenance without disrupting thousands of active jobs, and this ability itself makes the in-house data server development effort worthwhile.

An Xrootd server instance has two components: the Xrootd data server daemon and the olb load balancer (reused from AMS plug-in). Each server instance can be configured as redirector or data host. Instances configured as redirectors serve no data, but are well-known to the clients. Data hosting instances serve data but do not redirect. Using a combination of the two, clients can be load balanced among the data hosting instances.

Xrootd's dynamic nature significantly eases administration. New Xrootd data host instances automatically register themselves with the redirectors and are immediately available to stage in data and serve it to clients. This behavior happens automatically upon the daemon's startup without additional administrative attention. Many other high-performance features are available in Xrootd, but are beyond the scope of this paper.

Overall, the Xrootd server has shown itself to satisfy its design expectations. On modest dual-CPU hardware, a single data hosting instance has easily handled over a thousand clients and over two thousand open files simultaneously. Redirector instances handle over five-thousand redirects per second. With its robust architecture, Xrootd easily saturates its disk and memory bandwidth at below 50% CPU usage.

4.6 Bookkeeping

The ROOT-based eventstore lacks a central catalog, so a separate bookkeeping system is needed to track all collections and files. Bookkeeping shields users from many unimportant details, presenting groups of collections and files to users as named *data sets*. Data from activities like PR, REP, and SP is tracked in this system, while a QA group uses the same system to mark the data which may become part of an official data set. Another responsibility of the bookkeeping system is to manage tasks. This part of the system shepherds a task from definition to completion. This includes generating batch job configuration files, tracking the splitting of larger jobs, the submittal to the batch queue, monitoring job status, and the resubmittal of failed jobs.

The technology that best fits bookkeeping is an RDBMS. As with the eventstore, the specifics are abstracted, allowing more than one RDBMS implementation to be used. Some large centers, like SLAC (which has an Oracle site-license) use Oracle, while most small institutes opt for an open-source alternative, MySQL. The core database resides at SLAC, but is mirrored at around ten other sites not including the laptops of more adventurous users. All updates to the core are done on SLAC's instance, and replicas trigger their own synchronization (usually daily), flagging missing collections afterwards and downloading them automatically. External sites keep additional, non-synchronized local information specific to their local installations.

To reduce the dependency on central servers, jobs never talk directly to the bookkeeping system. Instead, all bookkeeping operations are done in advance: a query is sent, and the response is fed to the job in form of a site-independent configuration file.

The main goal is to provide a compact, simple catalog for users to find the most common data quickly, while allowing access to the full archive when needed. The bookkeeping system, now around 3GB in size, is currently being used for central production, data distribution, and analysis, but the task management component is only used for central production and has yet to be deployed for analysis use.

4.7 Other improvements

Apart from the major features already discussed, several features were implemented independently. Where skimming produced pointer collections in the first generation, a more flexible specification scheme was implemented, allowing the production of deep-copy, partial-copy, or pointer collections on a per-skim basis. With this feature, events are now duplicated by a factor of 3.2, trading off space for increased data locality and I/O performance.

Load-on demand is a feature intended to reduce data pressure when reading back event data. It operates as a thin abstraction layer between objects and their referents, faulting in the latter only when referenced. This sort of architecture is sensible when the object access is expensive and dynamic, adding only a minor overhead in the worst case. Load-on-demand has shown promise, but it is unclear whether the current pipelined analysis model will benefit, since most analysis jobs are based on preset processing sets which seem to read most of the data anyway.

Another independent feature is a facility for monitoring data servers. This project offers centralized views of measurements on BaBar's data servers, allowing administrators to more easily and quickly understand the system's basic health. Using the Ganglia package [10], the system is able to gather, store and present through web interface measurements such as CPU load, disk I/O and networking Alerting capability has been added on. Such information is expected to reduce downtime by speeding troubleshooting and diagnosis, and perhaps even prevent downtime and improve efficiency by making it simple to see destabilizing factors before they disrupt the larger system.

4.8 Have we met the challenge?

Initial experiences with the system have included teething problems, but things have been positive. The system was

built with an aggressive schedule, allocating relatively little time and resources to implementing the core features. Some well intentioned efforts to ease things in one area have complicated things in another.

To ease collection management, the new system adhered to a one-collection-one-set-of-files model. Instead of using database files as raw storage blocks, each file now belonged to only one collection. One could then move collections as easily as files, without being forced to utilize special tools. Yet this improvement carried with it some problems. Skimming and other production activities needed to merge their output collections to prevent the pre-merge number of collections (~millions) from straining the file system, the mass storage system, and other metadata bookkeeping systems. This improved analysis efficiency and reduced collection count by an order of magnitude, but increased each collection's size. At the same time, improved data quality began marking parts of collections as "bad." These two factors motivated the implementation of a more complex scheme to select events at a finer granularity within collections. The increased convenience and flexibility of this scheme has been well worth the added complexity.

The new system has been impressive in many ways. Administrators have found data and server management to be simpler. Users are excited about the analysis possibilities with interactive access, and relieved to not worry about database semantics. Though the more-complex, non-eventstore database was not migrated as part of this larger effort, work is already underway in evaluating its migration to a non-commercial alternative. Further development and tuning of the larger system are still taking place, but the system as a whole is now in wide use.

5. Lessons for future implementation

Managing large data sets requires unconventional techniques and poses major challenges which do not appear in smaller-scale systems. In the past five years we have learned how to build and manage large system for managing a petabyte data set, encountering such aspects as integrating client and server software, running large-scale computing farms, and administering databases. Experience in two different persistency mechanisms—a commercial ODBMS and a hybrid RDBMS/file-based solution—puts us in a unique position to recognize common data management themes. We have found that the largest of these are: (a) providing access, and (b) scaling the system.

5.1 Providing access

Efficient and convenient access to data is one of the primary goals of a data management system. Having a hierarchy backed by tertiary storage allowed the system to handle the bulk, but the heavy access load continues to be a challenge. The small, fine-grained random disk access that is typical in BaBar is especially challenging to the disk-based cache. Poor disk performance is a major bottleneck "solvable" only by buying more and more hardware.

Nowadays, disk capacity is growing at an impressive rate. The MB/$ ratio is rapidly going up, while the number of disk heads per MB is going down. Disks of a given capacity are cheaper, but slower. Their characteristics are therefore more like tape, offering good performance for sequential reads of large chunks (~1MB), but glacial performance for sparse, random reads of small chunks (<1KB). Access at the latter operating point is far from optimal for a disk, but it is the most common access in HEP (and many other applications). The disk-memory performance gap continues to widen. Memory is still too expensive to be used in bulk to replace disks.

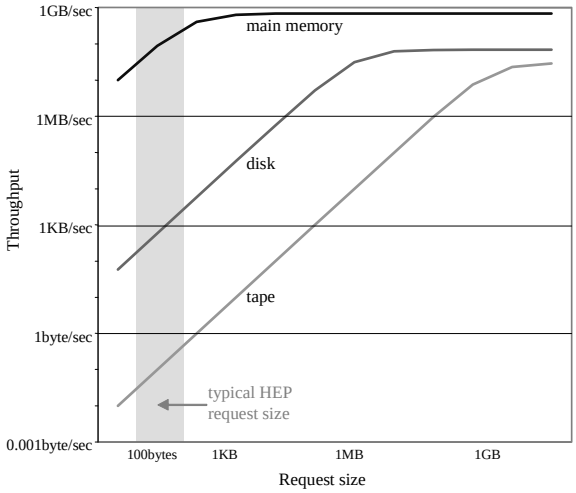

Figure 4 Current performance gap in memory systems

One potential solution to this problem is to de-randomize disk access. Techniques like pre-fetching or data train would be effective ways to schedule the disk, but they require an understanding of the access patterns. An effort is underway to trace and understand Xrootd access patterns for this purpose. Another way of achieving de-randomization is to use hints from applications to schedule less random access. Applications often know in advance what they will access, but scheduling efficient reads across multiple applications may not be trivial.

5.2 Scaling the system

When dealing with petascale data sets, scalability issues come up everywhere: number of servers, files, persistent objects, connections to servers, crashes, lines of code, and so on. Managing such data sets is a challenge easily underestimated. Very large systems are complicated by nature, therefore the simpler the solution, the better. At this scale, managing the metadata of the data set itself becomes a major problem that requires special consideration in the early stages of design. Being a

pioneer at this scale, BaBar did not appreciate this at the beginning.

With so much hardware in use, availability is another big challenge. Having a large number of nodes, servers, farms, disks, and tapes means that the system experiences frequent failures. To combat this, systems should minimize dependencies on central servers, use reliable server hardware and software, reduce failure impact (e.g. by replicating), and invest heavy effort in fault recovery mechanisms. Using commodity hardware reduces cost, but care must be taken to meet minimum reliability and to recover from the increased number of failures.

Distribution of data and computing load is crucial for performance at this scale. Files need to be distributed across many servers, or even sites. To reduce the server load, a thin-server/thick-client architecture can be used, as the load can be pushed off servers and distributed across many clients, reducing overall server cost. Load balancing is required to take full advantage of available server capacity, but it must be automated.

Mammoth data stores have necessarily specific and non-generic needs. Since such systems push the limits of technology, it is highly important to design carefully and control complexity. Needed features should be implemented robustly, and unnecessary features should be dropped for simplicity. These systems are often the first of their kind, so built-in flexibility is also important to allow adaptation to the continually evolving requirements and use cases.

Finally, systems of this size must account for physical considerations. Running large computing farms requires dealing with lots of heat, power demand, and floor weight. These three are among most serious challenges for people maintaining hardware at SLAC and collaborating sites.

6. Conclusions

Currently there is very little, if any, literature that covers management of a petascale database simply because there are no other databases today of that scale. With the amount of data collected and stored by the average business doubling each year [7], petabyte systems will be popular in a few years. Building such systems is very expensive, and wrong decisions might be very costly; therefore any practical experience with this new scale of computing is invaluable.

This paper presented our experience with managing BaBar's petabyte data store—the world's largest database. The experience has already proved useful for many, such as intelligence and law enforcement agencies [9]. The paper highlights design choices, describes the toughest challenges, points out unexpected surprises, and provides advice on the building, deploying and administering of a very large data set. Experience with two different persistence technologies (one commercial and one open-source) has allowed us to expose format-independent aspects and themes. The immense scale of the data set magnified nearly every aspect of data management in both technologies. Keeping data structures, workflow, and overall design simple is crucial. The inclusion of non-essential features is costly and will remain unappreciated by the real users. Planning for change makes inevitable migrations practical. BaBar's data sample will continue to grow rapidly in the next few years, and we expect to continue building our understanding of beyond-petabyte data sets.

7. References

[1] J. Becla at al, "On the Verge of One Petabyte – the Story Behind the BaBar Database System", in CHEP proceedings, La Jolla, USA, March 2003

[2] J. Becla, I. Gaponenko, "Optimizing Parallel Access to the BaBar Database Using CORBA Servers", in CHEP proceedings, Beijing, China, September 2001

[3] J. Becla, "Improving Performance of Object Oriented Databases, BaBar Case Studies", in CHEP proceedings, Padova, Italy, January 2000

[4] J. Becla, A. Hanushevsky, "Creating Large Scale Database Servers", 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, Pennsylvania, August 2000

[5] I. Foster, "What is the Grid? A Three Point Checklist", in GRIDToday, July 20, 2002, available at: http://www.globus.org/research/papers.html

[6] I. A. Gaponenko, D. N. Brown, "CDB – Distributed Conditions Database of the BaBar Experiment", in CHEP proceedings, Interlaken, Switzerland, September 2004

[7] Gartner, Inc. Monthly Research Review, December 2001, available at: http://www4.gartner.com/1_researchanalysis/mrr/1201mrr.pdf

[8] A. Hanushevsky, "The Next Generation Root File Server", in CHEP proceedings, Interlaken, Switzerland, September 2004

[9] G. Huang, "Managing Antiterror Databases", in Technology Review, June 2003

[10] M. Massie at al, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience", Parallel Computing, May 2004

[11] S. Patton at al, "Support in the BaBar Objectivity Database for Multiple Federations", in CHEP proceedings, Beijing, China, September 2001

[12] A. Rajasekar, "Storage Resource Broker – Managing Distributed Data in a Grid", in Computer Society of India Journal, October 2003, available at: http://www.npaci.edu/dice/srb/Pappres/Pappres.html

[13] ROOT I/O homepage: http://root.cern.ch

[14] J. Shiers at al, RD45 – "A Persistent Object Manager for HEP", CERN/LHCC 96-15, Feb 22, 1996, available at:

http://wwwasd.web.cern.ch/wwwasd/rd45/reports/lcrb_mar96
[15] R. Whiting. “Tower of Power”. In Information Week, February 11, 2002, available at: http://www.informationweek.com/story/IWK20020208S0009
[16] Xrootd homepage. http://xrootd.slac.stanford.edu