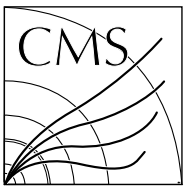
Available on CMS information server

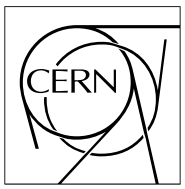
CMS NOTE 1997/074



The Compact Muon Solenoid Experiment

CMS Note

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland



8 September 1997

Prototyping of CMS storage management

K. J. G. Holtman

CERN, Geneva, Switzerland

Abstract

We report on a nine-month prototyping project concerning storage management in the CMS offline system. The work focused on the issue of using large disk farms efficiently. We discuss various hard disk performance characteristics which are important for physics analysis applications. It is shown that the layout of physics data on disk (clustering) has a significant impact on performance. We develop a storage management architecture which ensures high disk performance under a typical physics analysis workload.

Submitted to the Eindhoven University of Technology as a diploma thesis in the post-graduate programme on software design.

Keywords: CMS, prototyping, offline software, physics analysis software, ODBMS, storage management, hard disks, clustering, reclustering.

This document was submitted to the Eindhoven University of Technology as a diploma thesis in the post-graduate programme on software design.

Summary

At CERN, the European laboratory for particle physics, the fundamental structure of matter is studied using particle accelerators. Accelerated particles are collided head-on inside large detectors, which measure the collision products. The measurements are studied using large computer systems. One of the next-generation particle physics detectors, called CMS (Compact Muon Solenoid), will produce 1 Petabyte (1.000.000.000 Megabytes) of data each year, starting from 2005. This data volume pushes the limits of current database technology. The data storage and analysis software will be based on object technology, in particular on an object database. On the hardware side, CERN plans to use tape robots, large hard disk farms, and large CPU farms, all connected by a fast network.

The goal of the design project was to collaborate on finding methods for efficiently storing, managing, and retrieving the CMS detector data. Early on in the project, it was decided to focus on the issue of using large disk farms efficiently.

First, the relation between disk access patterns and disk efficiency was studied, in particular with respect to the types of access done in physics data processing. Measurements for different scenarios were made on current disk hardware, and were extrapolated to future hardware. The results of these performance studies, which were often very counter-intuitive, were then fed into a design phase. In this phase, a number of complementary storage management and optimisation mechanisms were produced. Together, these mechanisms keep performance high while database access patterns change. The designs were validated by making prototype implementations of the parts which were critical to performance.

Management Summary

This documents reports on research and prototyping work for CMS storage management and optimisation, in particular with respect to the efficient use of large disk farms in physics analysis jobs.

Chapter 4 reports on disk performance measurements important for physics analysis applications. It is shown that the layout of physics data on disk (clustering) has a significant impact on performance. In developing the CMS physics analysis system, a significant amount of work will need to be devoted to the creation of mechanisms for optimising the clustering of data. The required clustering optimisations are not provided by the object database, nor by any other commercial hardware or software component in the system.

Chapter 5 reports on prototyping activities which were performed to test the usefulness of the IRIS Explorer software framework for creating CMS physics analysis software. It is concluded that IRIS Explorer is not useful for this.

Chapter 6 reports on prototyping activities connected to storage management, in particular with respect to the clustering services shown to be needed in chapter 4. Chapter 6 contains an analysis of the requests which will be made on the object data store by physics analysis applications, and presents the global design of a storage management mechanism which would fulfil the stated CMS needs.

Results obtained in this project indicate that it will be possible to create a storage management system which maintains high performance, while meeting the flexibility requirements in the CMS computing technical proposal. For analysis efforts which repetitively access the same dataset, it will be possible to automatically optimise the placement of data on disk, to a level at which access speeds are close to the maximum disk access speeds as specified by hardware vendors.

In this project, the tests for finding performance and scalability problems were done on a medium-size hardware configuration (6 processor machine with two disk farms), using the Objectivity/DB database on top of the UNIX file system. Tests with larger hardware configurations and more software layers may reveal additional problems, which are not addressed by the optimisation services presented in chapter 6.

Contents

Chapter 1

Introduction

The CMS (Compact Muon Solenoid) detector is a next-generation particle physics detector which will be built at CERN. The detector will be ready in 2005, and has a planned operational lifetime of 15 years. The detector will produce 1 Petabyte (1.000.000.000 Megabytes) of permanently stored physics data each year. Teams of physicists will search and process this data to extract new physics results. For every Petabyte of raw data, some 0.1 to 0.2 Petabytes of derived data will be produced and stored by various physics analysis efforts. Unlike the raw data, derived data can exist in multiple versions.

The CMS data store requirements pose several key issues. The large data volume means that the data storage and processing system uses a significant amount of hardware (tape robots, disk farms, CPU farms), so that hardware failures can be expected daily if not hourly. Also, jobs which process the data have to be executed on a massively parallel platform, if they are to finish in reasonable time. To achieve the desired database throughput, a storage hierarchy will have to be used. At the bottom of the hierarchy will be tape robots which can hold all data, but have low throughput and high latency. At the next level is a disk farm, which has a much higher throughput (about a factor of 100 higher), and much lower latency, but which can only hold about 0.2 Petabytes of data. A large amount of memory based on RAM chips will be at the top of the hierarchy. Methods have to be developed for managing the migration and replication of physics data through the hierarchy. A final key issue is that the performance characteristics of 2005 hardware are not yet known. Some technology tracking is being done, but it cannot account for dramatically new developments. Much of the development work takes the most conservative technology predictions as a basis.

To address the key issues connected to the management of the data store, several prototyping activities are being done by CMS, often in collaboration with other groups having similar storage needs. The prototyping work reported on in this document focused mainly on the issue of data management at the disk farm level of the storage hierarchy. The prototyping efforts aimed at reconciling the data access needs of physics analysis jobs with the performance characteristics of disk drives, in such a way that the greatest possible throughput is achieved.

Chapter 2 contains environment and background information relevant for the work. Chapter 3 states the project goal, and discusses the project organisation which was chosen, and the management of risks in the project. Chapter 4 reports on measurements of disk performance characteristics which were made to support the subsequent design and prototyping work. Chapter 5 reports on prototyping activities which were done to test the usefulness of the IRIS Explorer software framework for creating

CMS physics analysis software. This chapter concludes that not using IRIS Explorer is the better alternative design decision, and we thus did not use Explorer in subsequent prototyping activities. Chapter 6 reports on our design and prototyping activities for CMS storage management and optimisation. It also describes the storage management design which was made. The first part of the chapter presents an analysis of the nature of physics data processing, and explains how this analysis led us to choose certain design alternatives. The rest of the chapter reports on design activities which were steered, not so much by a requirements analysis effort, but by prototyping efforts aimed at discovering various relevant properties of hard disks, and the operating system and database kernel layers above them. It presents performance measurements done on prototype implementations, and shows how these measurements led to subsequent design decisions. Chapter 7 has conclusions.

Chapter 2

Environment

2.1 CERN

At CERN, the European laboratory for particle physics, the fundamental structure of matter is studied using particle accelerators. The acronym CERN comes from the earlier French title: ”Conseil Europeen pour la Recherche Nucleaire”. CERN currently has the largest accelerator in the world, the LEP (Large Electron Positron) accelerator [1] which is a ring with a circumference of 26.7 km. The successor of LEP is called the LHC (Large Hadron Collider) [2]. The LHC startup is scheduled in 2005. Currently, a large share of CERN's resources goes into the design and construction of the LHC accelerator, and of its two main detectors, ATLAS (A Toroidal LHC Apparatus) [3] and CMS (The Compact Muon Solenoid) [4].

2.2 Particle physics

In accelerators like LEP and LHC, particles can be accelerated to near-light speed, and collided head-on. Such high-energy collisions happen inside detectors, which detect some of the products of the collision (particles and energy quanta) which emanate from the collision point. Figure 2.1 shows an example of such a collision, which is commonly referred to as an ‘event’.

By studying the types, speeds, and directions of the collision products in the event record, physicists can learn more about the exact nature of the particles and forces which were involved in the collision. Because of the large amount of data involved, events are studied with large computer systems.

In many physics analysis efforts, a large amount of time (and computing power) is spent in taking a large set of events, and narrowing it down to a much smaller set of interesting events.

For example, to learn more about Higgs bosons, one can study events in which a collision produced a Higgs boson which then decayed into four charged leptons. (A Higgs boson cannot be observed directly, only its decay products can be observed.) A Higgs boson analysis effort can therefore start with isolating the set of events in which four charged leptons were produced. Not all events in this set will correspond to the decay of a Higgs boson: there are many other physics processes which also produce charged leptons. Therefore, subsequent isolation steps are needed, in which ‘background’ events, in which the leptons were not produced by a decaying Higgs boson, are eliminated as much as possible. Background events can be identified by looking at other observables in the event record,

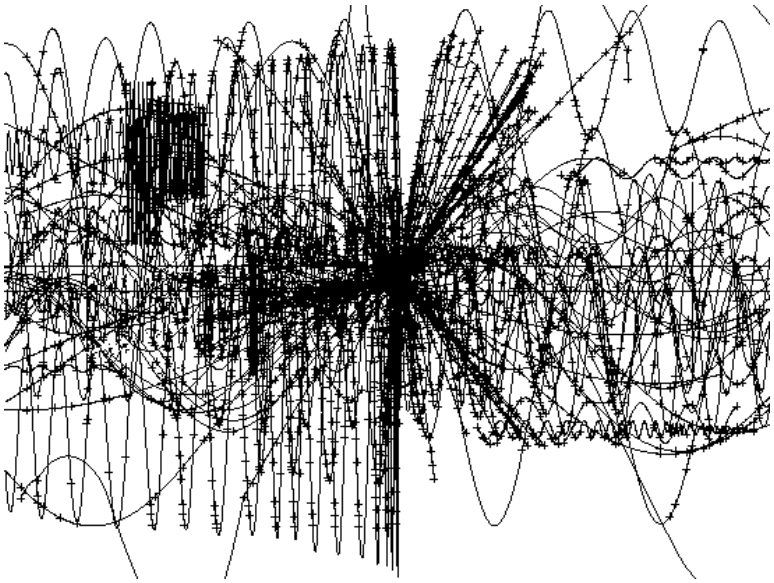


Figure 2.1: A CMS event (simulation)

like the non-lepton particles which were produced, or the speeds at which various particles left the collision point. Once enough background events have been eliminated, properties of the Higgs boson can be determined by doing a statistical analysis on the set of events which are left.

The process of narrowing down an event set is done in multiple steps, where each step is referred to as a 'cut'. The data reduction rate of the cutting process can be enormous. The final event set in the above example may contain only a few hundreds of events, selected from the 4×10^{14} events which occurred in one year in the CMS detector. This gives a data reduction factor of about 1 in 10^{12}.

The study of matter with accelerators is part of the field of high energy physics (HEP). A lot of the technology, including software technology, used in the HEP field is developed specifically for HEP.

2.3 CMS

The CMS detector (figure 2.2) is one of the two main detectors of the LHC accelerator. It is being designed and built, and will be used, by a world-wide collaboration which currently consists of some 130 institutes, which contribute funds and manpower. The institutes will also be the users of the detector when it is finished. CERN is one of the institutes in the CMS collaboration.

In normal operation, the LHC accelerator will let two bunches of particles cross each other inside the CMS detector 40.000.000 times each second. At the highest power levels, there will be about 20 collisions of two particles in each bunch crossing. In CMS terminology, an 'event' corresponds to a bunch crossing with collisions, and the combined measurements of the collisions products, done by the detector elements in the CMS, are called the 'raw event data'. The size of the raw event data for a single CMS event is about 1 MB.

There are two main systems in CMS data processing: the online system and the offline system (figure 2.3). The online system is a real-time system, which has the task of selecting (filtering out) the 100 most interesting events out of the 4×10^7 events in every second. These 100 most interesting events

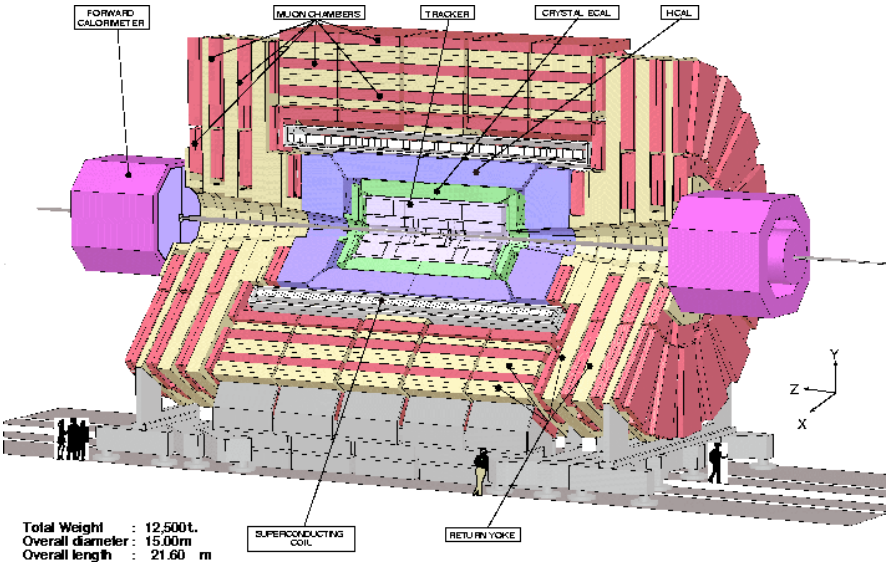


Figure 2.2: The CMS detector

will be recorded on long-term storage maintained by the offline system. The output of the online system is thus 100 MB/s data stream, containing raw events of about 1 MB each.

The detector takes data for about $1/3$ of the time of each year, which corresponds to about 10^7 seconds. Thus, in a year, 10^9 events will be recorded in the offline system, which corresponds to 1 PB (Petabyte, or 10^{15} bytes) of data. The total running time of the LHC will be about 15 years. The LHC data volumes are the largest of any known project in the time frame involved [5]. One of the big challenges in CMS computing is to invent methods and techniques which scale to the Petabyte level.

2.4 CMS offline data processing

The stored events will be analysed by about 20 groups of physicists in the CMS collaboration, using a large computing system known as the offline system. The activity of physics analysis is an example of a data mining [6]. Note however that physics analysis has been done long before the term data mining became popular.

Some parts of physics analysis are highly CPU-intensive, other parts rely heavily on the I/O bandwidth which can be achieved when accessing precomputed results. It is expected that the physicists in the collaboration will be able to use whatever computing power the offline system makes available to them: an increase in computing power means an increase in the potential for interesting physics discoveries, because analysis jobs can look for more subtle effects.

The CMS offline system will be based on a persistent object store. Physicists will access event data via an Object Database Management System that will be automatically available from the CMS software. Neither the tapes nor disk files should be accessed explicitly.

The offline system will rely on massive parallelism and special optimisation techniques to get the most out of standard hardware. The hardware will be upgraded through time to profit from new advances in computing technology. Performance could grow with several orders of magnitude from 2005 to 2020,

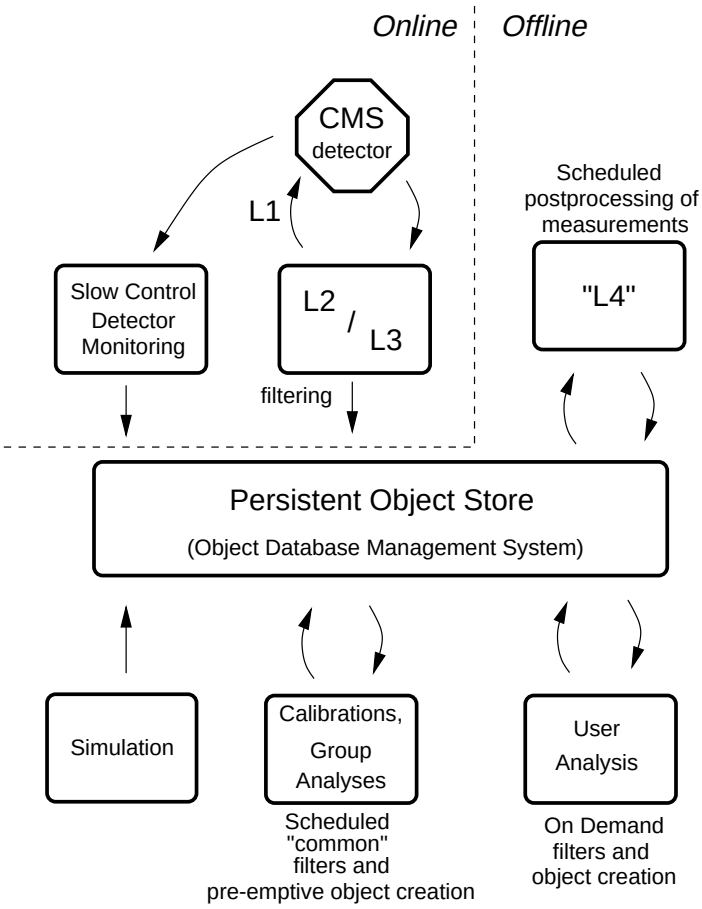


Figure 2.3: The CMS online and offline systems

and the system architecture and database model will have to take this into account.

In addition to the central CMS data processing system at CERN, there may be a small number of regional computing centres around the globe, which host smaller, but still considerable, CPU and storage capacities. Processing tasks are automatically scheduled to be executed at the most convenient location.

2.4.1 System dimensions

The following are estimates for the initial offline system in 2005, taken from [7]. Exact figures cannot be given: these will depend on the particular price/performance ratios for hardware at that time.

Processing power. The offline system will have about 10^7 MIPS of processing power, and will heavily rely on parallelisation to achieve the desired speeds.

Storage capacity. The system will have a robotic tape store with a capacity of several PB, and a disk cache of several hundred TB (Terabyte, or 10^{12} bytes).

I/O throughput rates. For the central data store at CERN, the maximum integrated throughput from the disks to processors will to of the order 100 GB/s, whereas the integrated tape-disk throughput may

be of the order 1 GB/s (32 PB/year) spread over some tens of devices.

The table below lists estimated data rates between the persistent object store and the different online and offline system components in figure 2.3.

Component name	Data rate from/to Persistent Object Store	
Slow control	$\sim$ 0.1 MB/s	(during detector operation)
L2/L3	$\sim$ 100 MB/s	(during detector operation)
L4	$\sim$ 110 MB/s	(keeping up with detector operation)
Simulation	$\sim$ 2 MB/s	(occasional writing over the year)
Calibrations, group analyses + User analysis	$\sim$ 100 GB/s	(continuous use of whatever resources can be obtained)

Note that the data rate for the two analysis components far outweighs all other data rates. Thus, if one is concerned with efficiently using the available hardware resources, only these two components need to be considered.

With respect to storage management and optimisation, the 'Calibrations, Group Analyses' component is the least problematic one of the two. This component typically runs large batch jobs which will process all data in a huge dataset, writing another huge dataset. Jobs will mostly be tape-bound, and will read data in a sequential way. The techniques for optimising the data flow for such jobs are relatively simple, and well-understood in the physics dataprocessing community.

The 'User analysis' component, on the other hand, will depend on storage management techniques which are novel in physics data processing. Jobs run by this component are typically short, sometimes even interactive, and will be concerned with analysing increasingly narrow subsets of all event data. These jobs produce semi-random data access patterns which are subject to gradual change. To keep the performance for these jobs high, the persistent storage manager will have to dynamically reorganise (re-cluster) data on disk to match the changing patterns.

2.4.2 Types of data

We can recognise a number of different types of data in physics analysis. All types of data below will be stored in the persistent object store of the offline system.

Raw data. For every event, the raw data is the record of all detector measurements done for this particular event. No processing has been done on the raw data apart from some lossless compression. In CMS, the raw data for a single event will have a size of about 1 MB. In experimental terms, the raw data for an event is a direct record of an observation, which serves as a basis for later interpretations. After having been recorded, it will never be changed.

Calibration data. The calibration data is used in interpreting the raw data. Calibration values, which can be queried for each event, record things like the exact geometric position of detector elements inside the CMS detector, and factors needed to interpret the output of various analog-to-digital converters, as recorded in the raw data blocks. Unlike the raw data, the calibration data records interpretations and judgements made by humans, usually with the aid of statistical analysis programs. When the interpretation process is refined, new versions of the calibration values can be stored in the database. Calibration values are not stored on an event-by-event basis, every value will be stored with a validity interval which will span many events. Querying of the calibration values will usually be on an

event-by-event basis.

Reconstructed data. The reconstructed data for an event represents an interpretation of the raw data and calibrations for an event in terms of physics phenomena. There are several types of reconstructed data. For example, the set of *reconstructed tracks* for an event is a set of particle trajectories which can be observed in the raw data, by connecting the 'points' measured by the individual detector elements. A reconstructed track record for an event has a size of about 100 KB. The record is produced with a *track reconstruction algorithm*. Good track reconstruction algorithms are very CPU-intensive. It is expected that the majority of CPU resources in the CMS offline system will be devoted to running track reconstruction algorithms. Another type of reconstructed data are reconstructed *jets*. A jet is a collection of tracks with the same origin and about the same direction, which is produced by a physics interaction at the quark level.

Event summary data. An event summary object has a size of about 10 KB, and summarises important features of the event. It is largely based on the reconstructed data. The record may for example contain information about those reconstructed tracks which were produced by particles with very high energies.

Event tag data. An event tag object has a size of about 200 bytes, and contains a very compact summary of the nature of an event.

In the CMS data model [7], the distinction between the different types and sizes of reconstructed and summary data is not very rigid. It is possible that some physics analysis effort is best done with event summary data blocks of 50 KB, and the offline system should take this possibility into account. At the database interface level, there will be no distinction between reconstructed, summary, and tag data: all will be accessed through the same mechanism.

As seen in section 2.2 in physics analysis a sample of events is narrowed down to a smaller set of interesting events in a number of steps, where each step is referred to as a cut. For Higgs analysis, with a data reduction factor of 10^{12}, the online system will account for a reduction factor of 4×10^5, the offline system for a factor of about 2.5×10^6. The first cuts in the offline system will make use of the small event tag objects, later cuts will gradually access larger objects for each remaining event. The final cuts may even need the raw event data itself.

2.4.3 Object databases

The CMS offline system will be based on a persistent object store. In the CMS computing strategy [7], which was developed in close collaboration with the RD45 project (see section 2.5), the choice was made to implement this persistent object store on top of a commercial object database management system (ODBMS). The ODBMS should be compliant with the emerging ODMG [8] standard for object databases. This choice is in line with the general strategy of using commercial software as much as possible, rather than developing software in-house. The existence of the ODMG standard, for which multiple vendors are making implementations, ensures that CMS will not be bound to a single object database vendor. This is important for two reasons. First, it ensures that the continuity of the CMS data store and its data management software is not dependent on the survival of a single vendor or product line until 2020. Second, it improves CERN's position in negotiating the price of the database license.

The Objectivity/DB object database [9] was chosen as the basis for all prototyping efforts in CMS in the 1996-2000 timeframe. The final choice for a production database will be made later. Objec-

tivity/DB was found to be the product best suited for use by CMS: its internal architecture can cope with extremely large databases, and it has facilities for building distributed systems. Also, the vendor, Objectivity Inc, has been shown to provide good support for its products. CERN currently has the status of an Objectivity beta testing site.

From a C++ programmer's viewpoint, an object database offers the service of creating and managing *persistent objects.* A persistent object has the property that, unless explicitly deleted, it will continue to exist after the termination of the program which created it. Aside from that, persistent objects can have all the features one can expect in a normal C++ object. They can inherit from other objects, and can have private and public data members, methods, and virtual functions. A non-persistent object can maintain references to persistent objects, and a persistent object can maintain references to other persistent objects. Compared to programming for a relational database, programming for an object database has the advantage that an object oriented data design will map naturally onto the object database facilities. There is no need for code which 'flattens' the object structure into something like relational database tables.

The ODMG standard for object databases contains a language-independent model for objects in the database. This object model defines, among other things, persistent object naming and identity, inheritance, locking, and the relations between objects which can be maintained by the database. The ODMG standard defines language bindings for C++, Smalltalk, and Java. These language bindings define facilities for object creation, naming, manipulation and deletion. Individual vendors can extend the facilities offered by the language bindings to areas which are not covered by the standard. Objectivity, for example, extends the C++ language binding with a 'clustering' mechanism, by which the application programmer can, to some extent, control the physical placement of objects on the database media. An optimal physical placement of objects is important to get high performance on reading.

2.5 Organisational environment

The project was performed in the CMC (CMS computing) group of the ECP (Electronics and Computing for Physics) division of CERN. This group contributes to the global computing work in the CMS collaboration. This arrangement has a matrix organisation structure (figure 2.4): the ECP division is on the vertical axis, and the CMS collaboration is on the horizontal axis, extending beyond CERN.

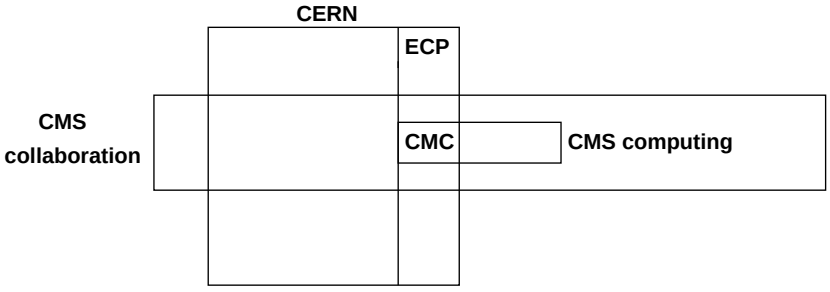


Figure 2.4: Position of CMC group in the organisation matrix

In the field of storage management for HEP, CMS is not the only experiment with Petabyte requirements. To pool the efforts whenever possible, a joint project between the LHC experiments, called the RD45 project (subtitle: A Persistent Storage Manager for HEP) [10], was created. In addition, to providing a common R&D forum for the future experiments at CERN, the RD45 project has ties

with a number of particle physics experiments in preparation outside of CERN. An example is the BaBar experiment at SLAC (The Stanford Linear Accelerator Center) [11], which starts in 1999 and will store 100 TB of event data per year, using Object Database technology. Another research project with ties to RD45 is the CMS Caltech/CERN/HP joint project [12], which is constructing a large-scale prototype of a CMS regional computing centre. Figure 2.5 shows the structure of the RD45 project.

Another important joint project is the LHC++ (Libraries for HEP Computing) project [13], which addresses the production of HEP-specific software libraries, and the licensing of commercial software. The organisational structure of the LHC++ project is similar to that of the RD45 project.

The activities in the design project were done in the CMS computing group, and the main goal was to contribute to the CMS computing milestones as recorded in [7]. A secondary goal was to contribute to the RD45 milestones, as part of the CMS contribution to RD45. Due to the strong overlap between the CMS computing and the RD45 milestones, many work items in the design project contributed to both milestones.

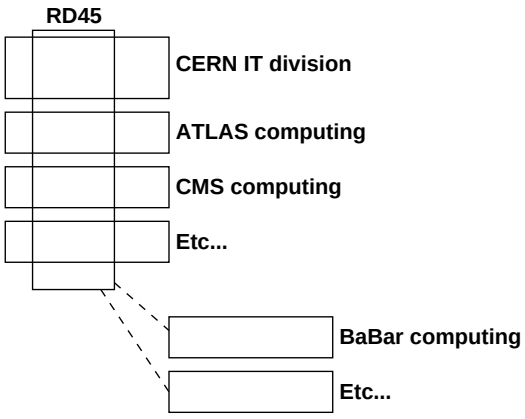


Figure 2.5: Structure of RD45 the project

2.6 CMS software process

The CMS software process differs from ‘textbook’ software processes in a number of ways.

First, there is the large timescale involved: the offline software has to be ready in 2005, and will be used at least up to 2020. The large lead time has made room for ambitious requirements. The LHC data volumes are the largest of any know project in the time frame involved [5], and this makes it necessary for the software process to actively push the limits of storage management technology.

The CMS software process [7] recognises the following (overlapping) phases in offline software development:

1995 – 1998	Research, selection and testing of commercial software components Initial prototyping of parts of the system
1998 – 2000	Research, selection and testing of commercial software components Development of full prototype of the system
2000 – 2002	Selection and testing of commercial software components Development of first version of the Operational Phase software Choice of the final the software environment, including commercial components
2002 – 2004	Development of production version of the Operational Phase software Deployment of production operational phase software
2005 – 2020	Maintenance of operational phase software (Maintenance involves re-optimising to exploit new developments in hardware)

It must be noted that in the current phase of the software process, a detailed overall design does not yet exist. Also, the system requirements have not been fixed: requirements are stated in terms of ‘the system must use the available 2005 hardware effectively’, rather than ‘the system must supply

n MIPS of processing power'. In the end, the 2005 system performance will be a function of the price/performance ratios of 2005 hardware, the CMS hardware budget (which is more or less fixed), and the effectiveness of (the optimisation algorithms in) the software.

Second, the dispersed nature of the CMS collaboration also makes the software process different from textbook cases. Though there is a core software team at CERN, other development activities are done in smaller teams, or by individuals, in other European countries, and in the US. The flat nature of the collaboration implies that there can be no centralised control structure: decisions are made on the basis of consensus. The core team at CERN can play a coordinating role at best.

Third, HEP software has traditionally been developed in FORTRAN, and the field is currently in the middle of making a slow transition to the use of object technology and a higher reliance on commercial software components. Many people in staff and management are only slowly making the shift to the new design and programming paradigms. As a result, there exists no clear, unified view on the potentials and limitations of object technology, and on the potentials and limitations of the chosen software components. As a result, a lot of time is spent in investigating and reporting on issues which would, in a more mature environment, have been identified beforehand as minor, non-critical, or irrelevant. It is recognised though that at least some of this is an inevitable by-product of traversing the organisational learning curve.

The CMS software strategy is to exploit commercial software components as much as possible. An important part of this strategy is to provide early directions and feedback to vendors of such software components, to ensure that the CMS needs are met in the long term. Providing early feedback can be particularly valuable in the ODBMS market. This market is still small, while it is expected to grow rapidly. Also, standardisation efforts for ODBMS systems are still underway. Feedback is mostly provided through the RD45 and LHC++ joint projects.

Chapter 3

Project Organisation

3.1 Project goal

The goal of the design project is to do research and prototyping for CMS storage management and optimisation. Various access and optimisation services will be designed and prototyped *on top of* the basic mechanisms of Objectivity/DB [9], in order to evaluate if and how an ODBMS can be used for implementing functions specific to the CMS computing technical proposal [7].

The prototypes will also aim to integrate Objectivity/DB with various other software components in the LHC++ [13] library. The goal of integrating with these other components is twofold: first, one wants to know if the component is a good choice among various options, and second, one wants to provide feedback and directions to the component authors. The activities are performed in close collaboration with the RD45 project [10].

3.2 Project planning

The project planning needed to address two major project risk factors: technological risks and organisational risks.

The technological risks are caused by a number of technical uncertainties. First, the research activities will (naturally) explore unknown terrain, and the problems which may be encountered cannot be known beforehand. Second, the prototyping activities deal with commercial software components which have not yet been evaluated fully, and for which unexpected integration problems may arise. The technological risks cause large uncertainties in the time budget for different project activities.

The organisational risks stem from the characteristics of the CMS software process. There is no ‘hard’ requirements document, and, in the 1997 phase, there is also no detailed software design. As a result, the design project goal is stated in broad, loose terms.

To address the risks, the planning follows the spiral model [14]. Risks are identified, and questions corresponding to these risks are formulated. The answers to these questions are used to address the risks. We use a cyclic prototyping approach with a variable number of cycles (figure 3.1). Each cycle should take no longer than 10 working weeks, though the actual time from the start to the end of a cycle may be a bit longer, because some overlap in the phases is allowed.

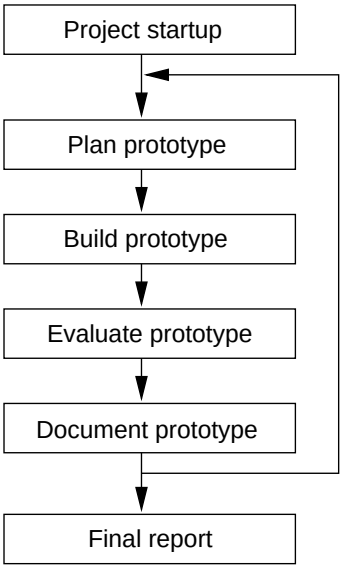


Figure 3.1: Project plan with cyclic prototyping

The planning of each prototype happens at the start of a cycle, not earlier. This makes it possible to take the latest information about the needs of the CMS collaboration into account when committing resources. Thus, frequent course corrections can be made, lowering the organisational risk that the project drifts away from the CMS mainstream.

The plan for each prototype clearly defines

- the functionality of the prototype,
- the software components which are tested or integrated by the prototype,
- a set of prototype evaluation metrics.

There is no requirement that the ‘Build prototype’ phase ends with a working prototype. If software integration problems turn out to be very big, then it is possible to end the building phase with a negative result, and a report on integration problems.

For every prototype, any bugs in software components, and integration problems between components, will be reported to the relevant party or parties. Depending on the nature of the problem, relevant parties can be the RD45 project, the LHC++ project, or the support groups of the various software component vendors.

The prototype documentation phase delivers the prototype source code, a discussion of relevant design decisions, and a report on the evaluation of metrics. The prototype will in general be throw-away prototypes.

The project startup phase has the following goals:

- Gather knowledge about the organisational environment at CERN
- Gather knowledge about the methods in High Energy Physics, in particular with respect to computing
- Gather knowledge about the CMS software process
- Study relevant documents produced at CERN, in particular the CMS Technical Proposal [4], the CMS Computing Technical Proposal [7], and relevant RD45 reports [10]
- Learn to program for the Objectivity/DB [9] ODBMS, and study in particular those parts of the ODBMS which relate to performance
- Gather relevant hardware performance figures
- Study existing data management techniques developed in the High Energy Physics community and outside it, in wider the field of data mining.

The ‘final report’ phase of the project has, of course, the goal of writing the final report required by the OOTI course.

Chapter 4

Disk Measurements

In the project startup phase, it was found that not all hardware performance figures which were relevant for CMS storage management could be found in literature. It turned out that the standard sources, like [15], [16], [17], [18] and [19], did not contain enough information to accurately predict performance in some important disk-bound HEP data access scenarios. Also, it was found that none of the CMS and RD45 members of CMS were already performing, or planning to perform, a detailed study of disk-bound data access.

It was therefore decided to add an extra ‘disk measurements’ phase to the project, directly following the startup phase. The structure of the ‘disk measurements’ phase (figure 4.1) was similar to that of a single prototyping cycle.

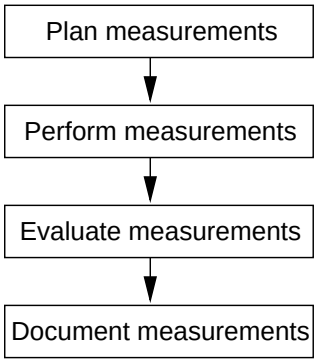


Figure 4.1: Planning for disk measurements phase

4.1 HEP data access scenarios

As seen in section 2.4.2, physics analysis jobs refer to reconstructed objects of events. The jobs in successive stages of a physics analysis process refer to less and less events. This leads to a data access pattern as in figure 4.2: the selectivity in reading data grows over time (the grey blocks represent objects which are being read by the jobs at a certain time).

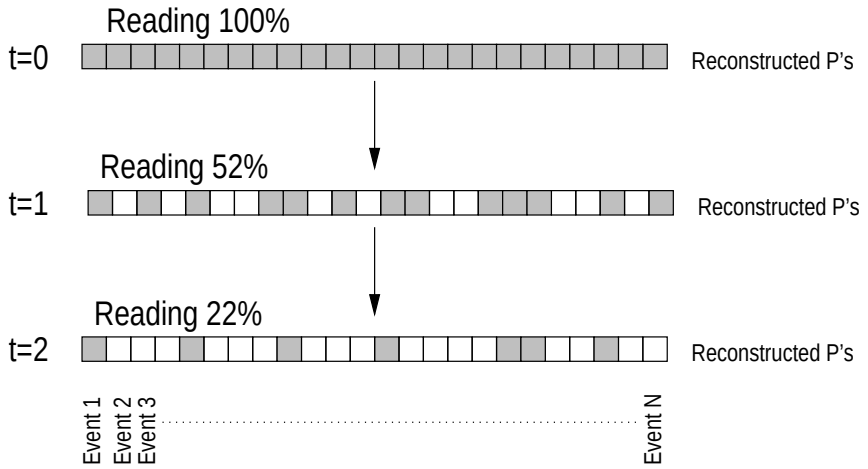


Figure 4.2: Increasing selectivity over time

One obvious way to optimise disk access performance for the $t = 0$ scenario in figure 4.2 is to cluster all objects on disk in the order in which they will be read. In this scenario, the job will perform a *sequential read* over the disk.

Now, the question arises what will happen to the performance in the $t = 1$ and $t = 2$ scenarios, assuming that

- the data is not reclustered, but kept on disk as it was in the $t = 0$ scenario
- the $t = 1$ and $t = 2$ jobs read events in the same order as the $t = 0$ jobs.

Under these assumptions the $t = 1$ and $t = 2$ jobs will perform a *selective read* (figure 4.3): the objects are read in a sequential 'left to right' order, but some objects are skipped.

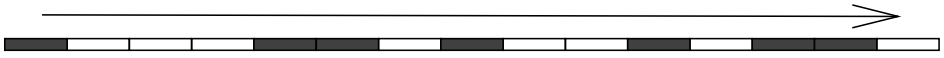


Figure 4.3: Selective reading

It turns out that literature does not answer the question of how to compute the disk performance for selective reading scenarios. The importance of these scenarios in HEP was the motivation for the disk measurement phase in the design project.

We measured the disk performance of selective reading for various selectivities and object sizes. For background and validation, we also measured sequential and random reading scenarios, even though the resulting curves, which are important in their own right, could also have been calculated using literature alone.

All measurements were performed on disks (2.1-GB 7200-rpm fast-wide SCSI-2, Seagate ST-32550W) which can be considered typical for the high end of the 1994 commodity disk market. All measurements are of raw disk performance, without any optimisation by an operating system cache.

4.2 Sequential and random reading

We measured the raw disk performance for sequential and random reading. The results, translated to performance rates for various average (reconstructed) object sizes, are in figure 4.4. Note that this figure predicts the overall system performance only in the case that the disks are the bottleneck. For object sizes below 1000 bytes, the CPU usage requirements of Objectivity/DB will more often be the limiting factor, at least in the case that only one process is reading from disk.

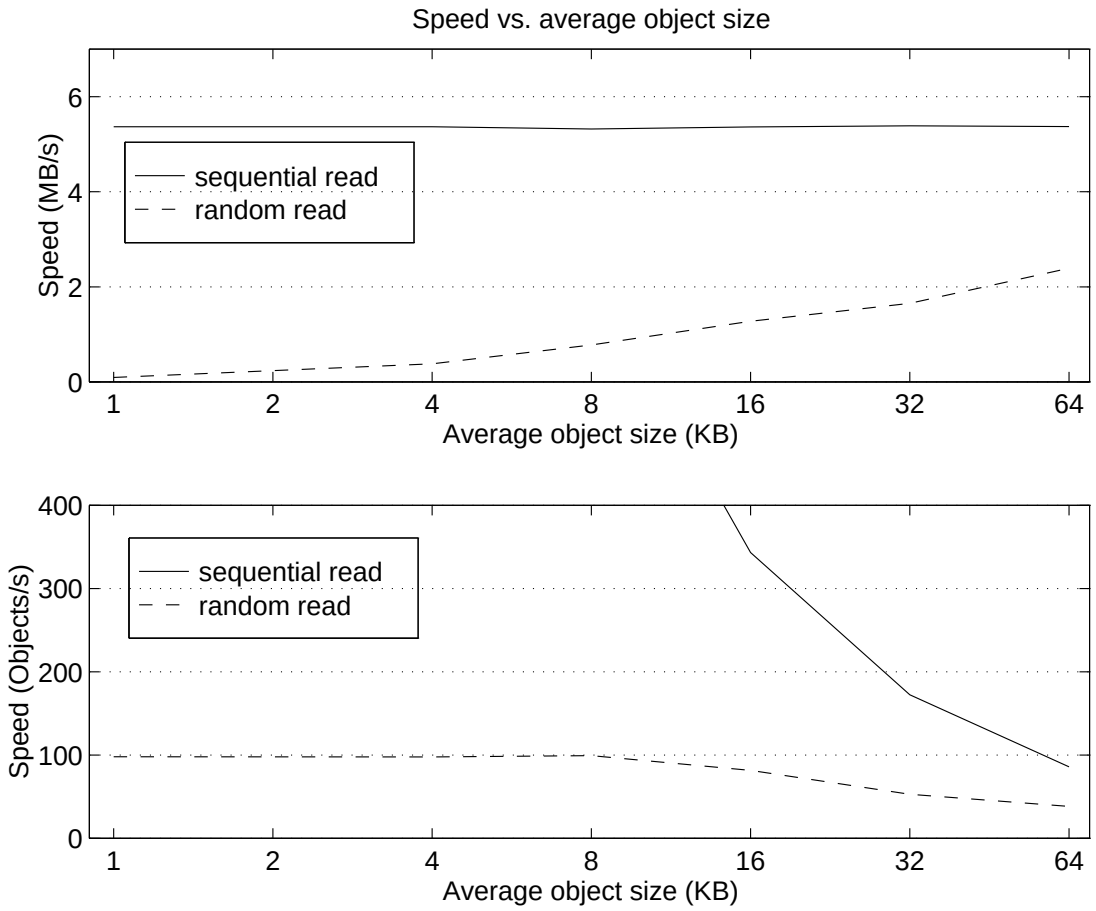


Figure 4.4: Performance for sequential and random reading scenarios

Observing the results in figure 4.4, we see that the random read speed is much lower than the sequential read speed. For example, it is about a factor 7 lower for 8 KB objects. As can be seen in the Objects per seconds plot, the random read time is completely dominated by the hard disk seek time for small object sizes.

As far as the design of a storage manager is concerned, these graphs illustrate the importance of good clustering, especially for (reconstructed) objects smaller than 8 KB. If the clustering is bad, the database performance will, in the worst case, degrade to that of the random read scenario.

After an analysis of hard disk technology trends ([20], [21], [22]) we found that the large gap between the sequential and random scenarios will grow even larger in future. Extrapolating trends, we can predict that the gap will grow from a factor 7 to a factor 20-30 for 8 KB objects, and from a factor 50

to a factor 150-250 for 1 KB objects,

The size of the gap has a big impact on the design parameters for storage management and optimisation mechanisms: as degradation to a random read scenario is very costly, considerable resources can be invested to avoid such degradation.

A system which spends 90% of its resources performing optimisations which avoid a degradation to the random read scenario may end up being faster than a system without such optimisations.

4.3 Selective reading

To determine the performance of selective reading for various average object sizes, we need to take into account that Objectivity/DB does its reading at the database page level, not at the object level. If, on average, every page holds 5 objects, and the object selectivity is 10%, then this results in the reading of 51% of all database pages. The relation between the page selectivity S_{pg}, the object selectivity S_{obj}, and the number of objects per page N_{pg} is as follows:

$$S_{pg} = 1 - (1 - S_{obj})^{N_{pg}}$$

We thus first measured the raw disk performance for various values of S_{pg} and various page sizes, end then extrapolated the results for different combinations of S_{obj} and N_{pg}.

In measuring the raw performance associated with various S_{pg} values, we first used the default Objectivity/DB page size of 8 KB.

The left hand graph in figure 4.5 shows the performance for various S_{pg} values, with 8 KB database page sizes (8 KB is the default page size in Objectivity/DB). Observing the graph, we see that the performance decreases rapidly when the reading of pages becomes more selective. Also, the curve only levels out when the performance level of the worst-case random read scenario is reached.

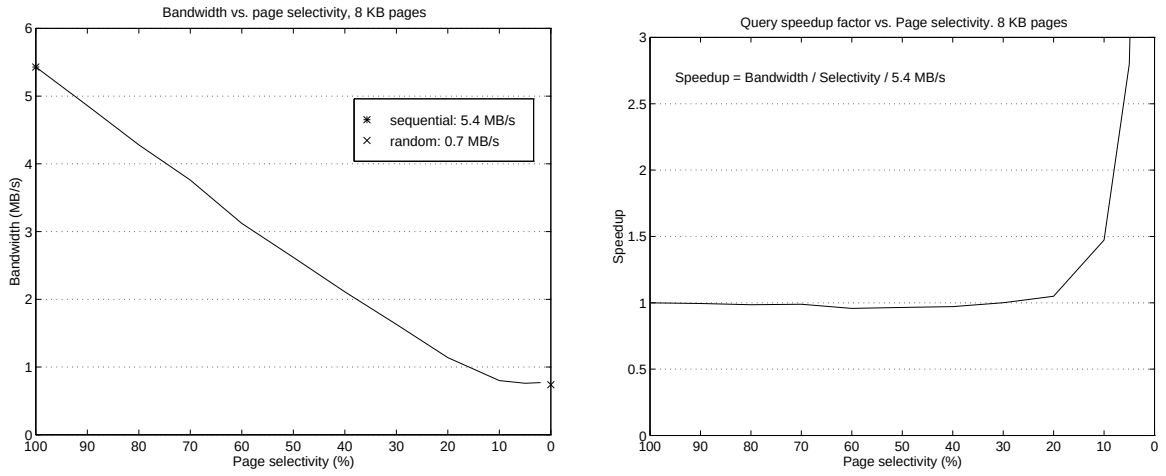


Figure 4.5: Performance for selective reading of 8 KB pages

To interpret these performance measurements, it is useful to plot them as a speedup curve. The right hand side of figure 4.5 shows the speedup when going from the sequential reading of all data, as in the $t = 0$ scenario in figure 4.2, to the selective reading of part of the data, as in the $t > 0$ scenarios.

We can conclude that (at least for this disk and this page size), selective reading is only interesting as an optimisation technique if the page selectivity S_{pg} is less than 15%.

By extrapolating the S_{pg} results for different combinations of S_{obj} and N_{pg}, we get the curve in figure 4.6. This curve shows, for different object sizes, the selectivity at which the analysis job becomes faster than the sequential reading analysis job in the $t = 0$ scenario. Note the double logarithmic scale.

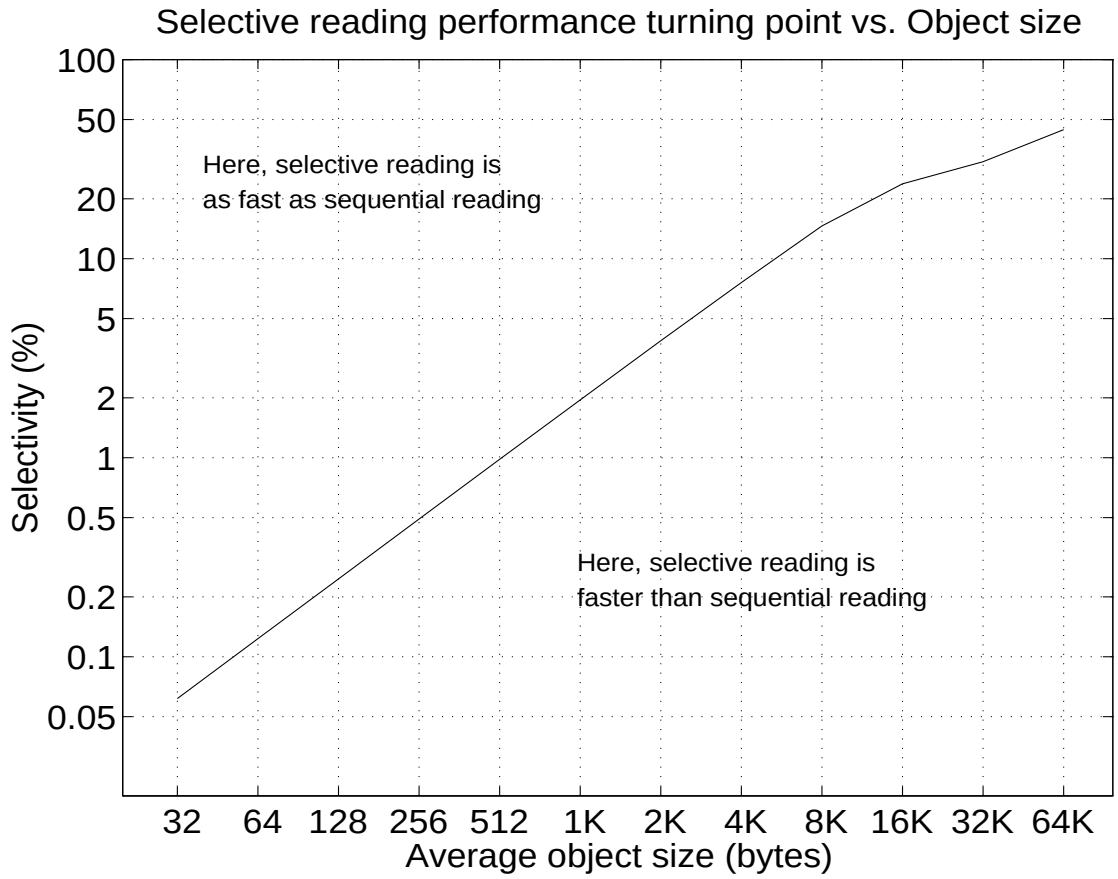


Figure 4.6: Selectivity at which jobs start to outperform sequential reading

The curve in figure 4.6 is based on raw performance measurements for a database page size of 8 KB. Raw performance measurements for other page sizes showed that the curve does not change much if an other page is chosen. Also, measurements on another type of disks, a study of disk technology, and a comparison of the disk specifications supplied by different manufacturers, indicated that this rather negative results applies to all types of commodity disk hardware: the curve in figure 4.6 may shift a bit for other disks, but it does not change fundamentally. For a RAID array [23], in which the data is striped across multiple disks, the curve will also be about the same: striping improves both sequential and selective reading performance with the same factor. Finally, a study of disk technology trends showed that, barring radically new hardware innovations, the curve will even move down, to smaller selectivities, in future. Note however that the commodity/desktop market, which is expected to drive innovation, is largely dominated by sequential reading: there is little market pressure to improve random and selective reading.

4.4 Conclusions and followup

4.4.1 Impact on design

In the end, we can conclude that, at least for small object sizes, hard disks are best treated as little tape drives in disguise. For the majority of disk-bound physics analysis jobs, selective reading is ineffective as an optimisation mechanism: an implementation with selective reading will not outperform an implementation which simply reads sequentially through all data, discarding the unwanted data on the fly.

Of course, this does not mean that selective reading mechanisms are useless. Compared to sequential reading, selective reading will use less CPU resources. If the reading of database pages goes via a network, selective reading will save network resources when S_{pg} is low enough, which is roughly for object sizes bigger than 1 KB. Also, selective reading will never perform worse than random reading, and will usually perform better.

If the goal is to make the $t = 1$ and $t = 2$ jobs in figure 4.2 faster than the $t = 0$ job, selective reading is useless. Any other technique which is solely based on skipping over unwanted data will be as least at useless: as selective reading is the technique which most closely mimics the sequential scenario, we can only expect even faster degradation toward random reading performance for other partial reading techniques. To make the duration of a job linear with the size of the requested data, one will have to physically move the unwanted data out of the way.

4.4.2 Followup in design and prototyping activities

At the end of the disk measurements phase, we were left with the conclusion that efficient storage management would have to rely heavily on the art and science of reclustering. As very little of this art existed, developing the art and science of reclustering was identified as an important goal for future design and prototyping activities.

4.4.3 Documentation of results

The results of the disk measurements phase were documented, in a form accessible to CMS and RD45, as part of [5]. The material in [5] differs from this chapter in that it places a stronger emphasis on exploring the design consequences of the measured performance characteristics, and less emphasis on the raw measurement results.

The results were also reported in a number of talks at the end of the phase. They were used to explain a some unexpected performance breakdowns observed in prototypes developed by other members of the RD45 project.

Chapter 5

IRIS Explorer

IRIS explorer is a data visualisation framework, which is used by various communities in science and industry, for example the computational fluidics community, to make 3D visualisations of complex datasets. It is currently not in use in the HEP community, but its introduction is considered as part of the LHC++ [13] strategy. IRIS explorer was originally developed by Silicon Graphics, but is currently being maintained by NAG (The Numerical Algorithms Group Ltd) [24], which also supplies various numerical libraries to CERN.

To produce a 2D or 3D graph or picture from some data set, an IRIS explorer user can build a 'map' (see figure 5.1), which is a program in Explorer's graphical programming language. Explorer programming consists of selecting modules, setting various parameters in the modules, and drawing data flow paths between the modules. The last module in a data flow chain will generally be one which produces a picture in a separate window.

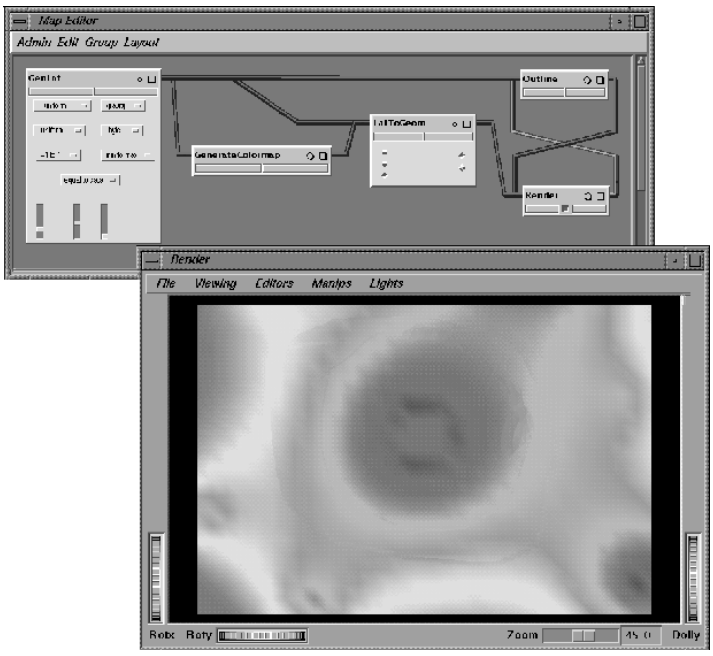


Figure 5.1: IRIS explorer map editor, with a map of five modules, and a picture window

One of the strengths of Explorer is that it allows the visualisation program to be developed in an explorative way: the user interface makes it very easy to tune the various module parameters, and to change or extend the map, in order to isolate or enhance certain features of the data. If a change is made, the Explorer framework will automatically perform the recalculations which have to be done to update the display. Thus, working with explorer is in some ways similar to working with a spreadsheet application.

Explorer offers a few hundred modules for the use to choose from when constructing a map. Some of the modules are part of the base package, others have been developed over time by various user

communities.

Within the LHC++ project, two observations were made with respect to the use of Explorer. First, by building some HEP-specific modules, it would be possible to make Explorer useful as a HEP data visualisation tool. Second, as event filtering can be modelled as a data flow computation, it would be possible to write Explorer modules for event filtering. This would allow physicists to do both filtering and visualisation under a single unified GUI.

Though the use of Explorer looked attractive, in early 1997 the LHC++ project had not yet validated the above observations by performing prototyping experiments. As a contribution to LHC++, it was decided to collaborate in such experiments as part of the design project. If successful, the prototype could be re-used as a GUI for further prototypes in the design project.

5.1 Prototype

Below, we cover the prototype which was built by covering the main definitions in the prototype plan.

Functionality. The prototype provides an event filtering chain toolkit based on Iris Explorer modules. The modules have the following functions:

- The *source module* can be used to select a test beam run, and it outputs all events in that run
- A *filter module* only forwards those events which match the filter predicate of the module. The filter predicate is a C++ expression
- The *end module* displays the number of events which are left at the end of the filter chain.

Figure 5.2 shows an event filtering chain built from the prototype modules. The source module (called RunList) is at the front of the chain, followed by three filter modules, containing three different filter predicates, followed by the end module, which displays a count of 16 events selected by the filter.

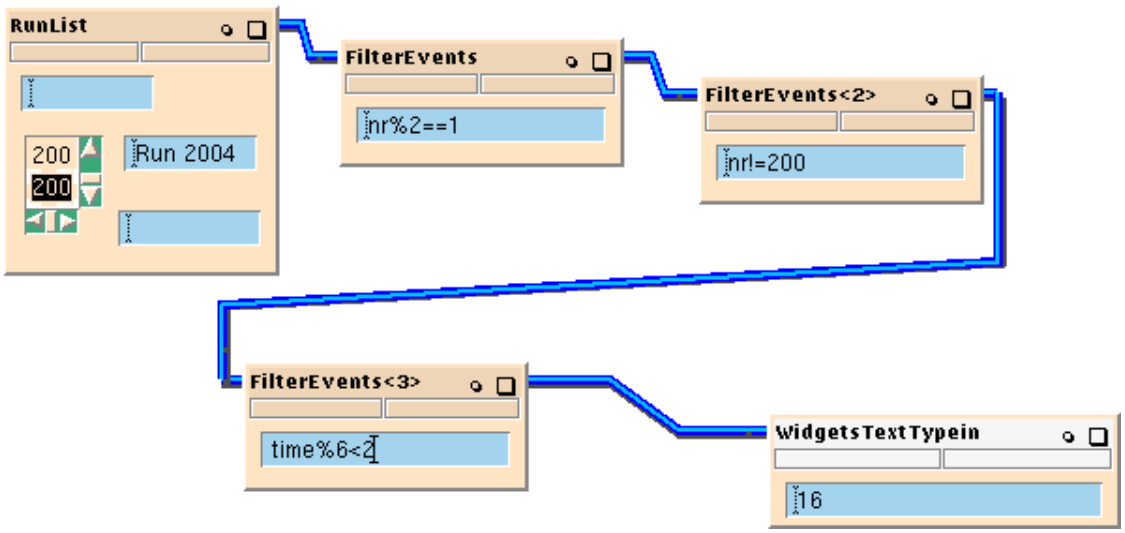


Figure 5.2: Screenshot of the IRIS Explorer filtering prototype

On the implementation side, a requirement is that, for efficiency reasons, all event database access has to be done by a single explorer module. Another requirement is that the filtering has to be implemented by dynamically compiling and loading the C++ filter predicates. (This requirement verifies the feasibility of dynamic compiling and loading, which plays an important role in the system design of the CMS computing technical proposal [7].)

Software components For the source module, an already existing LHC++ prototype module is used. For the end module, a standard Explorer module is used. The filtering module makes use of the following software components IRIS explorer, Objectivity/DB, Rogue Wave tools.h++, standard SunOS C++ compiler, SunOS dynamic linking and loading facilities. The event database model is the database model of the CMS testbeam prototype.

Evaluation metrics Metrics are: the interactive response time of the filter application, and the usability of IRIS Explorer as a software framework for developing CMS (filtering) applications.

5.2 Design

Most of the prototype design parameters were already fixed by the prototype requirements. To meet the requirement that all event database access is done by a single module, we used a mechanism in which the data flowing between the modules was not a collection of events, but a description of the selected run together with the filtering predicates so far. The filtering module at the end of the chain would, after detecting that it was at the end, take this description, turn it into compilable C++ code, and compile, load, and execute the code, thus running all filter predicates against the event database.

5.3 Conclusions

Based on the design above we were able to build a prototype meeting all requirements. With respect to the metrics, interactive response time was good, in the order of 1 second, in the case that a different run was selected, but bad, in the order of 40 seconds, in the case that a filter predicate was altered. The bad response time was due to the long time needed to compile the generated C++ code. The compiler spent most of its time processing the 750 KB of C++ header files included by the loadable code. A large explosion in header files is definitely a risk when integrating many commercial components. It seems attractive to reduce the need for recompilation by techniques which isolate the constants in the filter predicate and treat them separately: this way, a change in a constant would not require a costly recompile.

To meet the Objectivity/DB derived efficiency requirement that all database access is done in a single module, we had to use the Explorer dataflow facilities in an a-typical way. Though we had little problems implementing this a-typical use pattern, it also prevented us from exploiting much of the refined services offered by the Explorer framework. While the programmer of a more typical Explorer module could have left all control flow decisions and data dependency administration to the Explorer framework, we were forced to implement these things ourselves inside our modules.

Software frameworks, like IRIS Explorer, are designed to take the implementation of complex but common administration tasks out of the hands of the application programmer. However, as we have

seen, due to database usage constraints, Explorer could not take such tasks out of our hands in our case. We basically ended up using Explorer not as a framework, but as a GUI toolkit. As Explorer is not the nicest GUI toolkit around (for a start, it provides a C, not a C++ API), it can be concluded that using Explorer to build event filtering software is more trouble than it is worth. Providing an event filtering package based on another GUI toolkit, with a coupling to an Explorer-based visualisation package, seems a more promising approach.

Chapter 6

Storage Management and Optimisation

This chapter reports on the design and prototyping efforts with respect to storage management. These activities were done in two prototyping cycles. The first prototyping cycle, which roughly corresponds to sections 6.1, 6.2 and 6.3, was mainly concerned with analysing the physics analysis application domain. The second prototyping cycle, which roughly corresponds to sections 6.4 and 6.5, was mainly concerned with clustering and reclustering strategies. In the first cycle, the physicist was at the focus of attention. In the second cycle, the focus of attention shifted to disk access patterns.

We have seen in section 2.4.1 that, of the two types of jobs which will take the majority of system resources, the 'user analysis' jobs were the least well understood. We therefore focused our prototyping efforts on these jobs. The jobs are disk-bound: the execution time is dominated by the time needed for reading physics objects, as described in section 2.4.2, from the persistent object store.

6.1 Analysis of layers above the storage manager

The CMS physics analysis system is a layered system (figure 6.1), with the persistent storage manager somewhere in the middle. In the project startup phase, and in the disk measurements phase of the project (chapter 4), we analysed the characteristics of the layers below the storage manager. In this section, we report on our analysis of some of the characteristics of the layers above.

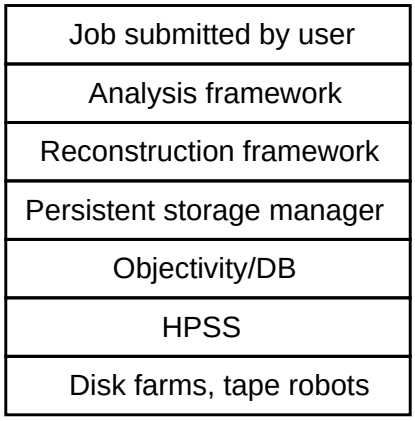


Figure 6.1: Layers in the CMS physics analysis system

The goal of this analysis effort was to get clarity about the interface between the storage manager layer and the higher layers. The leading questions were:

- What is the nature of the requests made on the storage manager
- How often are different types of requests made.

These questions are fundamentally about the *dynamic* side of storage management: how do the contents of the data store, and the demands on the data store, change through time. It was found that the CMS computing technical proposal [7] left these questions open for future research. In fact, the computing technical proposal argues that existing storage management methods and practices should *not* be

taken as a basis for answering questions about the dynamic nature of physics analysis, as the existing methods were developed more to meet the constraints of tape drives than to reflect the 'natural' way of doing physics analysis.

To answer the leading questions, we thus had to approach storage management from a very high level. In fact, we found that, to expose all possible forms of optimisation, we had no choice but to consider physics analysis as a process in which a community of humans forms a judgement, using computers as a tool.

The sections below report the results of our analysis, and on their consequences for storage management and optimisation.

6.1.1 Data dependencies in physics analysis

Figure 6.2 shows, for a single event, some typical object types in physics analysis, as described in section 2.4.2. The figure also shows the data dependencies between these objects: an arrow pointing from A to B means that the value of B depends on the value of A. Note that algorithms are also treated as objects in this figure.

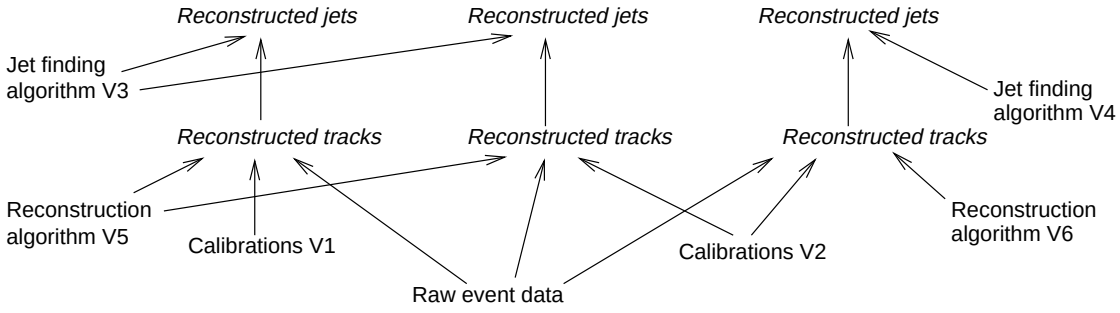


Figure 6.2: Some object types and dependencies for one event

As observed in the computing technical proposal [7], the reconstructed objects in figure 6.2 have the property that their values are uniquely determined by the values of all the objects they depend on. This means that, if an object is deleted, it can be re-computed again, provided that

- its type is still known
- its dependency relations with other objects are still known
- the values of these other objects are available, or can be re-computed again themselves.

The global system design in the computing technical proposal requires that, if an object is deleted, all above conditions are always met, so that it can be re-created on demand. The initial creation of an object will also depend on dependency data.

According to the global system design in the computing technical proposal, a user program will obtain reconstructed objects by making requests like

'give me the reconstructed tracks object with dependencies D for event e'

on the reconstruction framework layer (see figure 6.1). The request will contain all necessary data to compute the object if it is not available in storage.

The global system design therefore foresees a close integration between the reconstruction framework and the storage manager layers in figure 6.1. On receipt of a request like the one above, it will be decided, by some optimisation service which spans the reconstruction and storage management layers, whether the reconstructed object should be retrieved by the storage manager layer (assuming it is stored somewhere), or whether is should be re-computed by the reconstruction framework layer.

In the general case, if a reconstructed object is in store, retrieving it from store will be cheaper than recomputing it. This is because the data on which a reconstructed object depends is usually larger than the object itself. For example, the size of a reconstructed tracks object will be about 100 KB, but the size of the raw data on which it depends will be about 1 MB. In some cases however, reconstruction will be faster than retrieval. Examples are:

- A single reconstructed object is needed, and this object is only stored on a tape which is not in any tape drive at the moment
- A reconstructed object is only stored in a regional centre, which currently has a saturated network link, while the objects it depend on are all stored locally.

The availability of an on demand reconstruction mechanism has important consequences for storage management: it means that a storage manager can use a strategy of deleting reconstructed objects to save space. Because of the availability of sufficient dependency data, deletion of a reconstructed object can never cause permanent data loss, and deletion is transparent for the user.

It makes sense to think of the store of reconstructed objects as a *cache* which sits in front of the reconstruction service. Many of the design principles and techniques for a cache manager apply to the design of the storage manager, as far as managing reconstructed objects is concerned. The selection of objects to delete could be based, for example, on a ‘least recently used’ algorithm. This makes for a cheap service for freeing space, one that is much cheaper than garbage collection by reachability analysis, which some other object databases are forced to use.

The availability of a cheap service for freeing space again has important consequences for storage management: it means that we do not have to worry much about the cost of storing new reconstructed objects. We do not need an on-the-fly algorithm to decide whether it is worthwhile to spend space storing an object which has just been created by the reconstruction framework. We can simply always store objects which are created: if they are not used, they will be deleted soon enough. The only reason for not storing an object would be a shortage in disk resources and bandwidth for writing.

6.1.2 Physics analysis workcycle

As we have seen in section 2.4.2, a major part of a physics analysis effort consists of the construction of successive cut predicates, where each predicate separates ‘interesting’ from ‘uninteresting’ events. The construction of a single cut predicate can take a significant amount of time: from weeks to months. In extreme cases, a team of physicists can spend more than a year constructing a cut predicate. The construction of a cut predicate is an iterative process, in which one keeps refining the predicate (usually by tuning one or more of its constants), until its effects on the event set under consideration are both desirable and well-understood. The quality of a predicate is typically assessed by running it against a collection of real or simulated events, or running it against a collection of special events, of which the properties are better known than in the general case. Figure 6.3 gives a graphical overview of the physics analysis process.

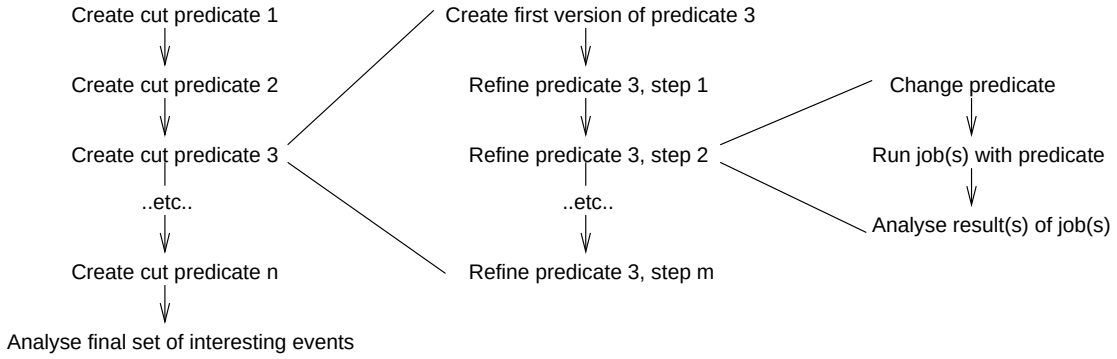


Figure 6.3: Physics analysis workcycle

During the iterative phase for a single cut predicate, the middle part in figure 6.3, one can expect database usage in which the same sets of (real or simulated) events are accessed over and over again while various constants are tuned. If the cut predicate involves some reconstruction algorithm, which is being refined at the same time, one may also see occasional reconstruction jobs with newer versions of the algorithm, and jobs comparing the results of the old and the new versions. Also, there may be jobs which isolate some set of events for closer study.

When the final version of a cut predicate is constructed, the predicate is applied to the real set of events under study, yielding a smaller event set for the next phase.

6.1.3 Granularity of access

Looking at the physics analysis workcycle, we can observe a number of things.

First, object access is not as random as in, for example, a library catalog database. Instead, during the refinement of a single cut predicate, there will be several collections of objects, which are revisited again and again by subsequent jobs. Each individual job will traverse one or more of these collections.

This ‘working set of collections’ will only change radically when refinement is completed, and work on a new predicate is begun. During refinement, we will only see small changes in the working set. For example, the working set may be extended with a new collection by running a new version of a reconstruction algorithm. Also, an analysis effort could sometimes shift focus from all objects in a collection to only part of the objects, so that a subcollection of another collection becomes part of the working set.

The storage manager should not try to make access to a single object in a collection fast, but on making access to all objects in a single collection fast. As seen in chapter 4, this means that we have to make sure that collection traversal leads to a sequential disk access patterns.

Inside a collection, the objects for all different events are independent from a physics standpoint. This means that there is no reason for a physics analysis job to put any constraints on the order in which the objects in a collection are traversed. The storage manager could use this lack of constraints to its advantage.

As far as the design of a storage manager is concerned, we will have two levels of granularity of access:

1. Access to a collection of objects
2. Access to a (compound) object which represents a single event, inside a collection

A storage manager will be able to spend considerable resources on every 'request' for a collection, because such requests only happen once or a few times for each job. This makes it possible to use storage management and optimisation algorithms with an unusually high (computational) complexity at the collection level.

At the object level, on the other hand, the storage management overhead will have to be low. As a general rule, whenever some management or optimisation mechanism can be moved from the object level to the collection level, the design should do so.

6.1.4 Sharing data and changes in data

It will often happen that different analysis efforts use the same data. Typical examples of data which could be shared are calibration constants, and reconstructed particle tracks based on a particular version of the calibration constants.

However, only data which has a read-only nature can be shared successfully. When refining a cut predicate, one needs a stable dataset against which to run subsequent versions of the predicate. If the dataset is not stable, one cannot accurately compare subsequent versions of the predicate by comparing the results of subsequent runs.

As far as storage management is concerned, this read-only nature of shared data has important consequences: as opposed to, for example, the data manager of an airline reservation system, which has to ensure that all users see all changes immediately, the CMS data manager will have to ensure that, if one user changes data, all other users will *not* observe a change.

Analysis jobs will not in general request the latest versions of some physics objects from the storage manager, they will request the same, frozen, versions again and again. This leads to an architecture in which a newer version of an object does not overwrite the old version, but is stored separately in a new location. Note that with such an architecture, we can expect much less locking and hot spot problems than with an average case multiuser database.

Of course, there has to be a service for letting users know about new versions. Such a service is best implemented at a high level indexing and notification service, not as part of the storage manager. The storage manager could however provide a user, who is considering switching to a new version, with information on whether objects for the new version have already been computed. If there are no reconstructed tracks yet for the latest version of some calibration constants, then the user may want to choose a less recent version of the calibration constants, for which the tracks have already been computed.

6.1.5 User role in optimising the system

When choosing a less recent version of some calibration constants over the latest version, the user is really making a tradeoff between quality and time: less accurate calibration constants are used (at least, assuming that later constants are always more accurate) in order to save on the CPU time it

would take to reconstruct all objects with the latest calibrations, and the space needed to store them. Such tradeoffs can be crucial for system efficiency, but note that they can only be made by the users themselves.

Thus, in order to be effective, the CMS system will have to make it easy for the users to plan and make tradeoffs like this. This can be done by providing a high level of tool support. For example, it would be very useful to have a tool which can quickly estimate the time needed to run a job if different (combinations of) versions of reconstructed objects are used.

As far as storage management and optimisation is concerned, this means that optimisation mechanisms, the effects of which can be cheaply and accurately predicted beforehand, should be preferred over mechanisms for which the effects are less predictable, or costly to predict. If predictions are not accurate, the users will loose trust in the optimisation tools, and will stop using them, leading to a system with much less sharing.

6.2 Design of jobs

Having done the analysis in section 6.1, we can now perform a synthesis step. We can make a high-level object decomposition of the physics analysis system, by defining classes to go along with each of the user-level concepts we identified. The most important classes, and their most important relations as far as storage management and optimisation is concerned, are shown in figure 6.4.

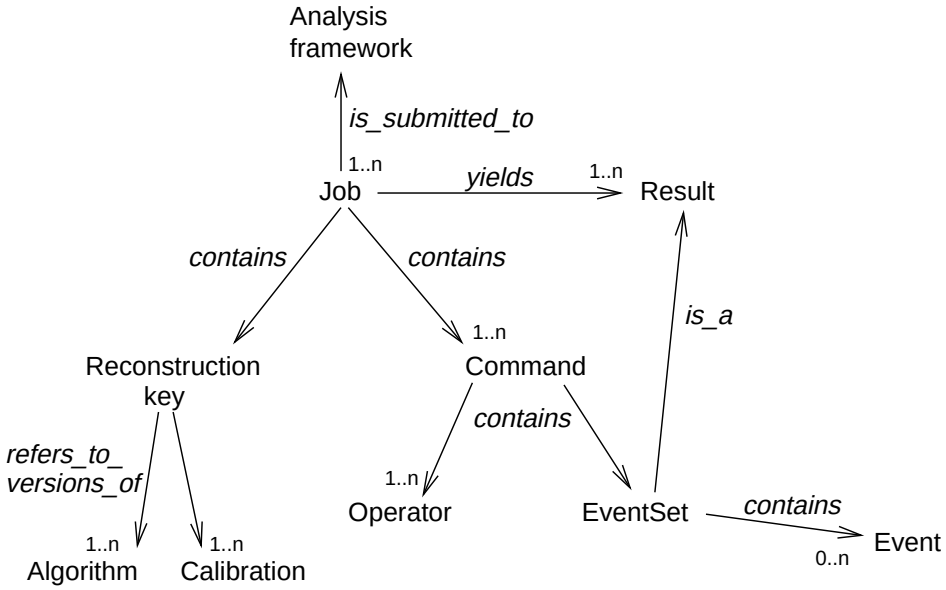


Figure 6.4: Job-centred high-level system decomposition

Of the design decisions which led to the decomposition in figure 6.4, the decisions concerned with the decomposition of the job class are most interesting. The other decompositions follow more or less naturally and inevitably from the constraints and conditions in the CMS computing technical proposal [7] and in section 6.1.

As the criterion for decomposing jobs, we chose to separate the parts which stay the same over multi-

ple jobs from the parts which change in each job. We chose this criterion because it has two strengths:

1. A user interface built with this job decomposition as a basis will have a natural separation between control elements corresponding to fast-changing data, i.e. control elements which are used often, and control elements which correspond to slow-changing data and are not used often
2. This job decomposition is most suitable as a basis for considering data management and optimisation mechanisms later on.

Note that there is also a synergy between these two strengths: the user interface will guide the user into formulating the job in such a way that it fits naturally to the capabilities of the optimisation mechanisms.

At the top level of the decomposition of a job, we separated out the reconstruction key, which represents the choices for different versions of algorithms and calibration constants made by the user at the start of a the cycle of refining a cut predicate (figure 6.3). A reconstruction key would record user choices like:

For all reconstructed tracks in the job, use version X of the track reconstruction algorithm with version Y of the calibration constants.

This would allow a command to simply request

'the reconstructed tracks'

relying on the reconstruction key to make this request unambiguous.

The purpose of the reconstruction key is only to record long-term decisions. If the goal of the job would be to compare the results of two new track reconstruction algorithms, then the versions of these algorithms would be mentioned in the job commands, not in the reconstruction key.

At the next level of job decomposition, we have one or more commands, which are decomposed into operators and event sets. A typical command would be:

make a histogram of the A_0 values of the tracks of the events in the set E, cut by the predicate P.

or, in a more mathematical notation:

$$result := histo(\bigcup_{e \in E \wedge P(e)} A_0(tracks(e)))$$

In this command, there is one event set E, all other things are members of the operator class. Our decomposition separates out the event set because it is an important type of object with respect to storage management, and one which, unlike the cut predicate, will not change often.

Our design allows multiple commands in a single job because users will often need multiple histograms to analyse the effects of a new refinement step, for example

1. the histogram of the A_0 values of the tracks of the events in the set E, cut by the predicate P
2. the histogram of the A_1 values of the tracks of the events in the set E, cut by the predicate P
3. the histogram of the X_8 values of the tracks of the events in the set E, cut by the predicate P

Producing these histograms in three subsequent jobs will require the tracks to be accessed three times. If they are produced in a single job, the tracks need to be accessed only once.

6.3 Design of collections

This section is concerned with the management of storage in terms of collections. Recall that in section 6.1.3, we identified a two-level granularity of access in physics analysis:

1. Access to a collection of objects
2. Access to a (compound) object which represents a single event, inside a collection

We noted that there will be a 'working set of collections' during the refinement of a cut predicate, a working set which only changed gradually, and that it would make sense to manage storage in terms of collections which are persistent.

6.3.1 Refining the notion of collections

Using the decomposition of jobs in the previous section, we can now refine our notion of a persistent collection. In a single command like

$$result := histo(\bigcup_{e \in E \wedge P(e)} A_0(tracks(e)))$$

we can identify access to the following collection of reconstructed track objects:

$$\bigcup_{e \in E \wedge P(e)} tracks(e) .$$

This yields the following properties for a persistent collection of objects:

- A persistent collection c consists of a set of (raw data or reconstructed) objects, corresponding to some event set E_c
- All objects in the collection have the same type (for example 'tracks' or 'jets')
- All object values in the collection have the same dependencies, they are computed with the same versions of reconstruction algorithms and calibrations

From sections 6.1.1 and 6.1.4 we also obtain the property:

- Objects in a collection can never be updated: after creation they become read-only objects.

Of course, the above refinement, on the basis of the form of a command, is not the only way possible way to refine the collection concept: it represents a design decision. Though there were strong indicators showing that this was the right decision, we could not completely justify it beforehand: there was a risk that the decision would lead into a dead ally. We therefore employed the strategy of only tentatively making this decision, and verifying, as we went along, that it did not cause any problems. This was done for some other refinement strategies too.

The main questions which need to be answered in order to verify the design decision above are:

1. How does the storage manager find the right collections for a job?
2. How do new collections get created?

3. What is the mechanism for deleting old collections?
4. If a job accesses more than one of these collections, it is possible to ensure efficient access, even though the collections are managed separately?

Early on in the prototyping cycle, we did not try to answer all these questions exhaustively. We satisfied ourselves with tentative answers first. We will not report on these tentative answers here. Instead, we will only report on the definite answers. The first question is answered in the next section. The second and third questions are answered in section 6.5, the fourth in section 6.4.5.

6.3.2 Finding the right collection

Suppose that a job will access the reconstructed tracks objects with the dependencies D, for all events in the event set E_j. Finding the right collection for this job can be with the following simple algorithm:

1. find all collections of tracks with the dependencies D
2. pick the smallest of these collections for which $E_j \subset E_c$, where E_c is the event set corresponding the collection.

To support the implementation of this algorithm, we need two things. First, we need a database index by which we can look up collections based on dependencies. The Objectivity/DB database has facilities which make it easy to build and maintain such an index. Lookups will have $O(log\ n + f)$ efficiency, where n is the number of collections in the database, and f is the number of collections found. This is certainly efficient enough.

Second, we need a mechanism to calculate $E_j \subset E_c$ for the various candidate collections. The E_c term in this expression poses little problems: we can simply store the E_c of each collection with the collection itself. Storing the E_c as a set of event identifiers with each collection will not bring a very large overhead: we expect event identifiers to have a size of 8 bytes. Obtaining the E_j term, for a job which is about to run, can be more difficult. However, we will usually be able to get at least an $E_{es} \subset E_j$ by taking the EventSet of the job object. Using E_{es} instead of E_j in the algorithm above will usually still give good results. Also, we expect that we will often be able to find a better approximation E_{cp} with $E_{es} \subset E_{ec} \subset E_j$ by doing a symbolic analysis of the cut predicate in the job command, comparing it against cut predicates in previous jobs for which we stored the event sets E_{cp} produced by their cut predicates.

Finally, provided that we store all set contents in a sorted order, the step of comparing two event sets to see if $E_j \subset E_c$ can be implemented in $O(s_j + s_c)$ time, where s_j and s_c are the sizes of the two event sets. By comparing candidate collections in smallest to largest order, the comparisons can take no longer than $O(f \cdot s_{cb})$ time, where f is the number of candidate collections and s_{cb} is the size of the event set of the best candidate collection which is eventually found. As the reading of the wanted objects in the best collection will take $O(s_{cb} \cdot s_{ao})$ time where s_{ao} is the average size of the objects stored in the collection, we can expect that, except for very large f and very small s_{ao}, running though candidate collections will not cost a significant amount of time compared to the actual reading of the data. We also expect that it will often be possible to calculate $E_1 \subset E_2$ even faster, in $O(1)$ time, using symbolic comparison of the cut predicates associated with E_1 and E_2, or by maintaining a cache of the results of earlier comparisons.

The above analysis shows that, in the worst case, the overhead of finding the right collection is small compared to the execution of the whole job. The smallness of the overhead is mainly due to the fact

that event identifiers, with a size of 8 bytes, are small. We can expect that it will be feasible to use random access memory, rather than slower disks, to hold the representations of all but the largest event sets associated with jobs and collections. With event sets in RAM, the overhead of choosing the right collection will be negligible, and it will be possible to offer a service to physicists by which the execution time of a proposed disk-bound job can be predicted very quickly.

In this section, we have shown that there is a simple, robust, and efficient enough basic algorithm for finding the right event set. We argued that symbolic analysis techniques will often be able to make the basic algorithm much faster. Our goal was to show an existence proof of a collection finding service, we do not propose that the basic algorithm is used, without any optimisations, in the real CMS storage manager. Completely different collection finding mechanisms should also be considered, for example a mechanism which does not try to find the best collection beforehand, but which aims to switch to the best collection on the fly, using knowledge about the event set the job has accessed so far. Also, a larger degree of sharing between different physics analysis efforts could by achieved by using a collection finding algorithm which does not choose the single best collection, but the best set of collections $E_{c_1} \cdots E_{c_n}$, such that $E_j \subset (E_{c_1} \cup \cdots \cup E_{c_n})$.

6.4 Design of collection data clustering

In this section, we report on our design and prototyping efforts with respect to the clustering of the objects contained in collections on disk. In chapter 4, we identified clustering issues as a major risk factor. We saw that breakdowns from the speed of a sequential reading scenario to the speed of a random reading scenario could come quickly and unexpectedly. Because of the lack of knowledge about what would, and what would not cause such a breakdown, we chose a design and prototyping strategy in which each step could be carefully checked by making measurements to detect a possible breakdown. We decided to approach the problem in the following way.

1. Make a list of simplifying assumptions, which allow one to develop a storage management mechanism in which jobs will always yield sequential reading performance
2. Design such a mechanism and test, by prototyping, if the performance is indeed according to the sequential reading scenario
3. Choose one of the simplifying assumptions, drop this assumption, and check, by prototyping, if the performance is still according to a sequential reading scenario. If not, fix the performance breakdown by adding additional optimisation mechanisms. Repeat until all simplifying assumptions are dropped.

Of course, successful termination of step 3 in this strategy could not be guaranteed beforehand. It could be possible that we would encounter a breakdown in step 3 which could not be fixed, even not by restarting the process at step 2. But even in case of failure, this strategy would at least leave us with an accurate pinpointing of the reason why a breakdown to random reading was inevitable. Another important strategy is that it does not introduce optimisations unless it is shown beforehand that they are really required, thus keeping the system as simple as possible.

In the end, it turned out that the strategy did successfully terminate, without us having to go back to step 2. In the next sections, we will describe our steps 1 and 2. Each iteration through step 3 will be described in a separate subsequent section.

6.4.1 Initial simplifying assumptions

We made the following simplifying assumptions to ensure that we could build an initial prototype with sequential reading:

1. There will only be one job running on the system
2. All collection elements are on disk, not in RAM and not on tape
3. There will be only one disk farm in the system
4. The job will access only one collection
5. The job will not be parallelised
6. The job does not do CPU-intensive computations, it is disk-bound.

6.4.2 Initial design

Producing sequential reading under the above assumptions is easy. We made the following design decisions:

- Subsequent jobs should always request the objects in a collection in the same order
- The objects in the collection are clustered in this order on disk

To validate these design decisions, we designed and implemented the following classes:

Event. Objects in this class are persistent. An event object represents a single event. Its object identifier (OID) in the database acts as a unique identifier for the event. The object identifier is an 8 byte value.

EventList. Objects in this class are persistent. An EventList stores an EventSet object from the design in figure 6.4. The list elements are ordered: iteration over an EventList will always visit the events in the same order, which is the order in which they were stored in the EventList.

RecObj. Objects in this class are persistent. A RecObj (reconstructed object) stores reconstructed data about a single event. This is an abstract class, various derived classes exist to store different types of reconstructed data.

Collection. Objects in this class are persistent. A Collection stores a set of RecObjs, clustered in the order in which they were added to the collection. The Collection has an iteration service which allows for selective reading. If a job requests the RecObj for a particular event from a Collection, the collection will return either a handle to this stored RecObj, or a status code indicating that no RecObj was stored for that particular event.

After implementing these classes using Objectivity, the running of test jobs showed that we did indeed get sequential reading access patterns. To validate that access patterns were indeed sequential, we developed a tool which could trace and visualise the file system calls done by the object database. We also did test runs on large datasets, measuring the actual performance. As expected, performance was indeed according to the sequential reading scenario.

6.4.3 Multiple jobs

We first dropped the simplifying assumption that there would only be one job running on the system.

To test the performance effects of running multiple jobs together on the same hardware, we ran multiple copies of our test implementation in parallel on a 6-processor machine. Every job reads a different collection, but all collections are on the same disk. The resulting performance graph, which shows the combined throughput for all jobs, is shown in figure 6.5.

As can be seen, there is no performance breakdown: the disk access pattern produced by running multiple sequential reading jobs together is still sequential enough to maintain high performance. This is an important result: it shows that we can optimise each job individually, leaving the efficient scheduling of the disk access operations performed by the different jobs to the operating system. Of course, we do have to take into account the possibility that the operating system optimisations might break down if the individual jobs start to access data in a less sequential way.

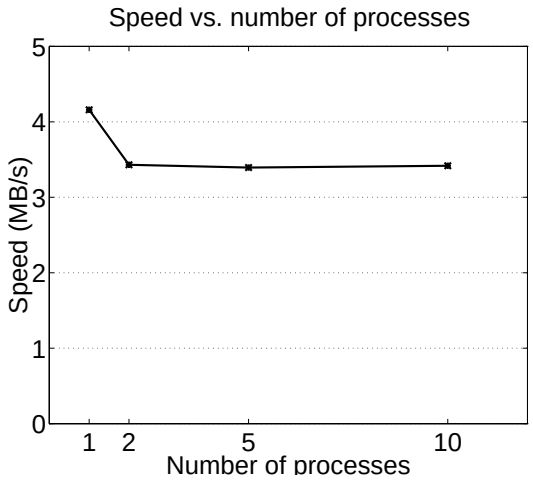


Figure 6.5: Performance effects of running multiple jobs together.

6.4.4 Multiple disks

To see if the simplifying assumption that there was only one disk could be dropped, we repeated the test performed in the previous section on a system which had two disk farms connected to it, with multiple disks in each disk farm.

For the first test, we put all collections on a single disk farm. For the second test, we put half on the collections on the first disk farm, the other half on the second disk farm. The results of the tests are shown in figure 6.6. As can be seen in this figure, there is no performance breakdown: the use of two disk farms nicely doubles the overall throughput, except of course in the case of one job reading one collection, which is on a single disk farm.

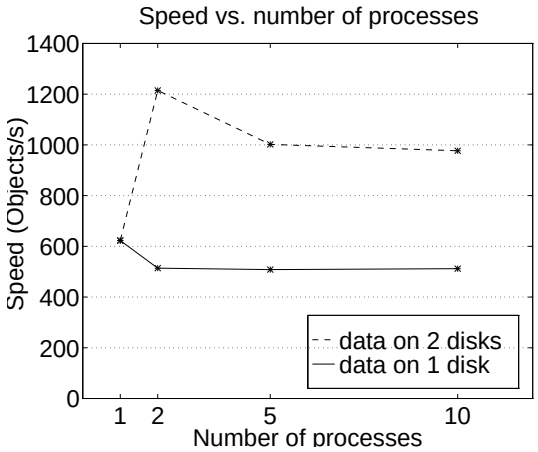


Figure 6.6: Effects of using two disk farms

6.4.5 Multiple collections in one job

To test the effects of dropping the simplifying assumption that a job only reads one collection at the same time, we did tests with a single job reading many collections of 7 KB objects in parallel. First, the job reads the first object in every collection, then the second object in every collection, and so on. The resulting performance curve is shown in figure 6.7. Here, we can see a definite performance breakdown.

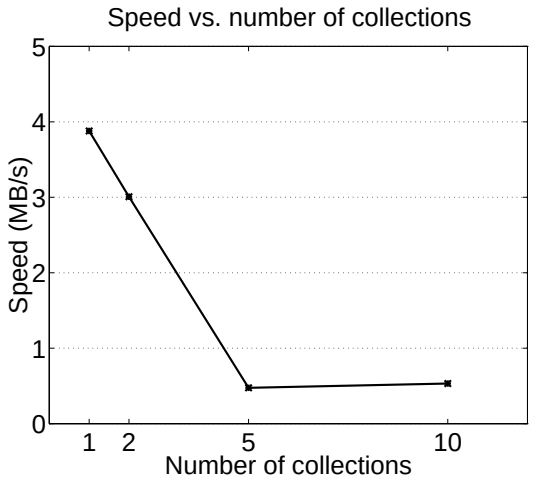


Figure 6.7: Reading multiple collections in parallel

Following our strategy in section 6.4, we tried to fix this performance breakdown by adding an additional optimisation mechanism. By tracing the disk read system calls performed by the database on the operating system, we could determine that the pattern of reads performed by the database jumped wildly over the disk. The database did not bunch subsequent page reads in the same collection together.

An example of a system call pattern we measured is on the left hand size in figure 6.8. This graph shows the database behaviour for a job which is reading three collections in parallel. Apparently, neither the operating system, nor the disk controller were able to recognise the regularity in this pattern and schedule the appropriate read-aheads which would have reduced the number of disk arm movements to be made.

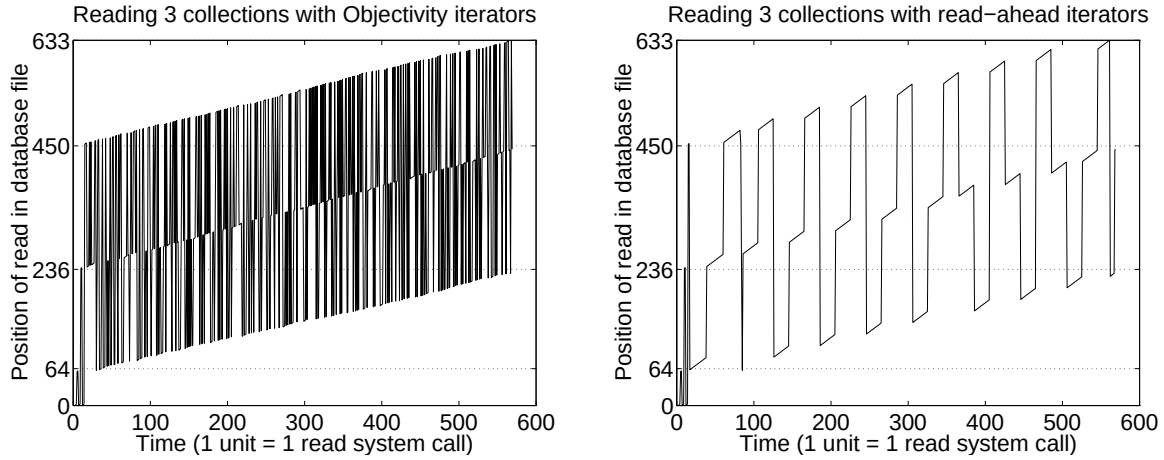


Figure 6.8: Read patterns produced by the database

To solve this performance breakdown, we implemented a small read-ahead layer on top of the database. We implemented this layer by refining the iterator class provided by the Collection class for accessing the collection contents. We created a new iterator class, with the same interface, in which every Nth call to the 'get the next object' function causes the next N objects to be read from disk. Internally, the new iterator objects maintain two of the old iterator objects: one for reading ahead, and one for offering the regular iteration services to the calling job. The objects which are read ahead with the first iterator will be found in database cache memory when requested through the second iterator. Thus, all disk reads happen in bursts of N objects.

The right hand side of figure 6.8 shows the database access pattern if the new read-ahead iterators are used. Reading now takes place in several sequential chunks, separated by seeks to other collections. Figure 6.9 shows the resulting performance curves, for a read-ahead of 20 objects and for a read-ahead of 100 objects. With a 100 object read-ahead, there is no noticeable performance degradation anymore: the time spent in disk drive arm movements is negligible. In the case of reading 10 collections in parallel, the 100 object read-ahead will need 8 MB of object database cache memory to work. This 8 MB is a small enough amount by today's standards.

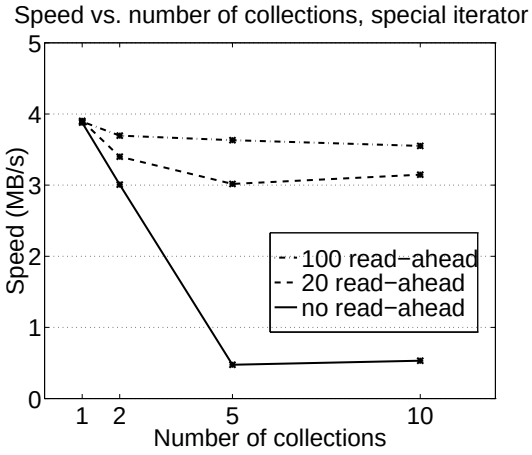


Figure 6.9: Fixing the performance breakdown with read-ahead iterators

By fixing the performance breakdown with our special iterators, we are now in a situation in which we can optimise the access pattern for each collection individually. The optimisations do not have to take the possible reading of other collections at the same time into account.

6.4.6 Parallelising jobs

We have seen in section 6.2 that a typical command in a job performs a calculation like

$$result := histo(\bigcup_{e \in E \wedge P(e)} A_0(tracks(e)))$$

This calculation can be parallelised quite naturally by partitioning E into subsets $E_1 \cdots E_n$ and calculating

$$result := histo(\bigcup_{e \in E_1 \wedge P(e)} A_0(tracks(e))) \oplus \cdots \oplus histo(\bigcup_{e \in E_n \wedge P(e)} A_0(tracks(e)))$$

where the $\oplus$ operator 'adds' two histograms. Each subjob could calculate a single

$$histo(\bigcup_{e \in E_i \wedge P(e)} A_0(tracks(e)))$$

The final 'adding' of all histograms is a cheap operation. As far as data dependencies are concerned, we have complete freedom in partitioning E into subsets.

The results in the previous sections show how this partitioning can be done without causing a performance breakdown: we should cut the event sets into parts which cause the subjobs to perform sequential reading. If we partition an event set E with a size of 10^6 events into ten subsets $E_1 \cdots E_{10}$, then E_1 should contain the 10^5 events which would be read first by a single-process implementation of the job, E_2 the 10^5 events which would be read after that, and so on.

To account for variations in the execution time of subjobs, it would be best to cut a job into at least five times as many subjobs as there are processors, and to use a processor farming approach for

executing the subjobs, keeping the processor farm loaded with, say, 1.5 times as many subjobs as there are processors. We have seen in sections 6.4.3 and 6.4.4 that running multiple disk-bound (sub)jobs, which all do sequential reading on different data, on a 6 processor machine does not cause any performance breakdown problems.

To ensure that the read-ahead mechanism in section 6.4.5 can keep avoiding performance breakdowns, every E_i for a subjob should refer to at least a few hundred kilobytes of data. But this minimum poses no significant problem: parallelisation only becomes interesting for jobs which read at least tens of megabytes of data. Disk-bound jobs which read less data will be finished in a few seconds even without parallelisation.

The above makes us confident that, at least on a platform where all processors have equal bandwidth to all disks, the scheduling of the efficient parallel execution of a job will be straightforward. If completely symmetric bandwidth is not feasible because of technology or cost constraints, it would be best to arrange disk and CPU farms as in figure 6.10, giving each disk farm a dedicated CPU farm for executing subjobs which refer to data on that farm. In this arrangement, the event data would be divided in some fashion over all disk farms, but all data for a single event would be on a single disk farm. Of course, in this case, the scheduler which divides a job into subjobs would have to take the division of data over the disk farms into account, so that each subjob only uses data on a single farm.

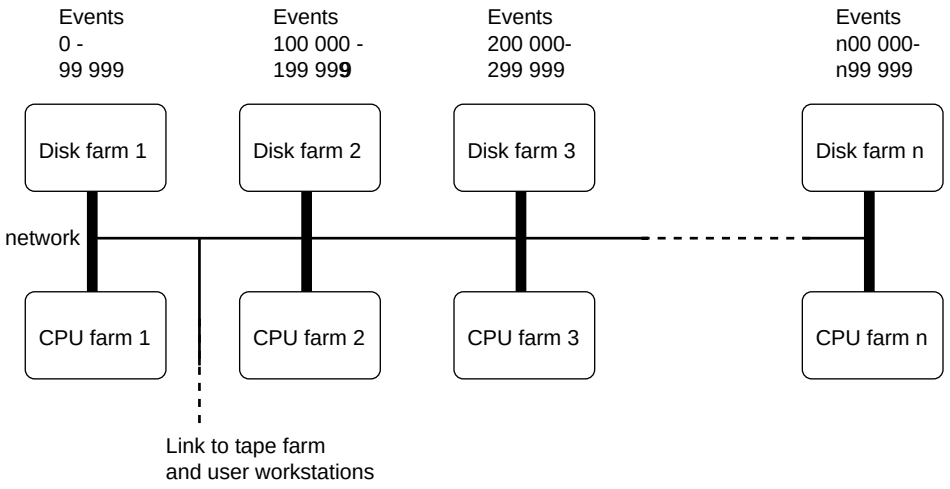


Figure 6.10: Massively parallel system with dedicated CPU farms

6.4.7 Jobs which are CPU-bound, not disk-bound

The read-ahead mechanism from section 6.4.5 will ensure that CPU-bound jobs have the same burst-like I/O behaviour as disk-bound jobs (see the right hand side of figure 6.8). We therefore expect that, as far as storage management and optimisation is concerned, CPU-bound jobs will not introduce new problems at the disk level. However, we have not verified by making measurements that CPU-bound jobs, especially parallelised CPU-bound jobs, would not cause performance breakdowns in other parts of the system: this was beyond the scope of our project.

6.4.8 Jobs which also access collections on tape or in RAM

If a job starts to access a collection on tape, the High Performance Storage System (HPSS) [25] beneath the Objectivity/DB database (see figure 6.1) will move the part of the database in which the collection resides to disk first. The job will remain blocked until the move of data to disk is completed. Thus, a job which uses data on tape will not produce a new type of disk access pattern: it will jump between not doing any disk access and producing the read pattern of a regular disk-bound job. When moving data from and to tape, the HPSS will perform strictly sequential reading and writing on both the disks and the tapes. We expect that the management and optimisation techniques outlined above will not break down for jobs which also accesses collections on tape.

The reading of collection data from RAM will be much faster than the reading of collection data from disk. Thus, a job which accesses both collections on disk and in RAM will remain disk-bound. The use of a collection in RAM will not cause a qualitative change in the disk I/O behaviour, so we expect that the management and optimisation techniques outlined above will not break down for jobs which also accesses collections in RAM.

Note that the storage management techniques we developed above are not concerned with jobs which *only* access data in RAM. For these jobs, radically different management techniques could be more optimal.

6.5 Design of collection data reclustering

In the previous sections, we talked about optimising access to existing collections. Here, we will answer the question, posed in section 6.3.1, of how new collections get created.

We recognise two different forms of creation. First, as discussed in section 6.1.1, if a job requests objects which are stored in no existing collection, these objects will have to be created by the reconstruction framework layer of the CMS system (figure 6.1). As concluded as the end of section 6.1.1, it will almost always be attractive to store the new objects in a new collection, so that they do not have to be reconstructed again.

Second, new collections could be created by rearranging (reclustering) the objects in existing collections. As concluded at the end of chapter 4, this reclustering is necessary to maintain good performance as analysis jobs become more selective in their reading.

6.5.1 Reclustering patterns

If a job is reading a collection with a selectivity below a certain threshold, it becomes attractive to copy the selected data to a new collection.

An example of this is shown in figure 6.11, in which a job with a cut predicate of $E_x > 10$ is creating a histogram. Only 60% of the objects from the collection are being read, so the storage manager has decided to copy the selected objects to a new collection. The copying can be done in parallel with creation of the histogram. If the copy is directed to different disks in the disk farm, which are not currently involved in the

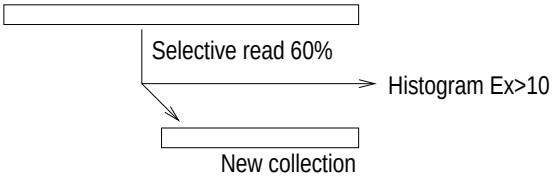


Figure 6.11: Basic reclustering pattern

reading of data, the copying will not add a performance overhead. If the same job is run again, the storage manager can find and use the new collection, as discussed in section 6.3.2.

A problem with this optimisation by copying is that extra disk space is used. We can address this problem by extending the pattern as shown in figure 6.12. Here, the objects which are not selected are copied too, to another collection. After the creation of the two new collections, the original collection can be deleted without losing any reconstructed objects. A job which requests all data in the original collection, in the order in which it was stored in the original collection, can be handled by reading the two collections at the same time and merging the data streams, as shown in figure 6.13.

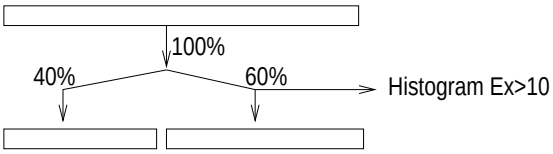


Figure 6.12: Extended reclustering pattern

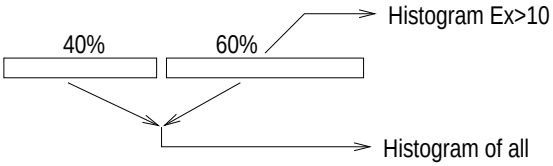


Figure 6.13: Using the new collections

Experiments with a test implementation showed that the merging of multiple collections did not cause a performance breakdown if the read-ahead iterators from section 6.4.5 were used. A collection which was cut into ten parts could be accessed with about the same speed as the original collection. During the creation of the test implementation, some performance-related bugs were found in the Objectivity/DB database. These bugs have been reported to the vendor.

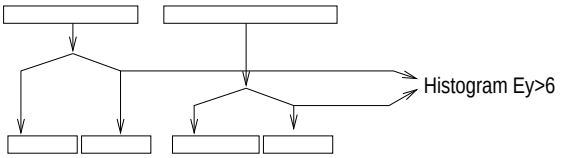


Figure 6.14: Splitting of two collections into four collections

Figure 6.13 shows a clustering of collection data that is optimal for two different jobs. If another job, with another cut predicate $E_y > 6$ is run on the data in the original collection, this can again lead to selective reading, this time on two collections. In that case, the extended reclustering pattern can be used again, leading to a splitting into four collections, as shown in figure 6.14. The end result, in figure 6.15, is a clustering of data which is optimal for three different jobs. This cutting process could be repeated to create 8 and 16 collections. Of course, the cutting process cannot be repeated indefinitely: with a cut into 32 collections, the overhead of for merging for a job which requests all data in the original collection will probably become too large.

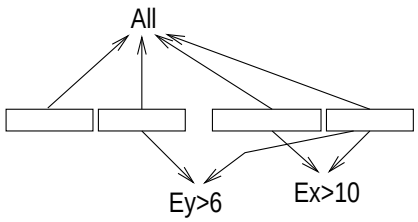


Figure 6.15: Optimal clustering for four jobs

A big advantage of this technique of reclustering is that it can be done automatically, using the actual access patterns produced by jobs. The method does not rely on guidance from human operators.

6.5.2 Managing the set of collections

In section 6.1.1, we saw that it will always be possible to recompute deleted objects using dependency data. We concluded that it would therefore make sense to manage the store of reconstructed objects as a *cache.* In terms of collection management, this means that collections are created whenever possible,

and that some scoring mechanism is used to select collections for deletion, or for migration to the slower tape store. For example, if a collection is split, we do not expect it to be deleted immediately. Instead, we expect that its score will be lowered, so that it is more likely to be deleted. We expect scores to be based on factors like

- how often the collection is used
- the degree of overlap with other collections
- how often the collection has been split.

Similar scoring mechanisms could be used for deciding on whether to merge collections.

Because of the similarities to cache management, which is a well-understood field with little surprises, we expect that the development of adequate collection management mechanisms will pose little problems. We can also expect, however, that the collection management system will have a significant number of tuning parameters, and that tuning will make the difference between a merely adequate collection management mechanism and a near-optimal mechanism. Such tuning can only be done while the system is running, from 2005 on, and it may be economically feasible to devote a significant manpower to it.

Chapter 7

Conclusions

In 1996 and 1997, the object store related work in CMS and the RD45 project was mainly concerned with identifying and investigating possible problem areas. This focusing on problem areas was done to prepare for the creation of fully functional prototypes, starting in 1998. Our project identified and investigated the problem area of using disk farms efficiently while database access patterns change. The choice for this area was at least partly coincidental: in the project startup phase, studies of the project environment and of literature revealed a number of open problems. The disk efficiency problem was only one of them, but it happened to be a problem on which no one else in CMS or RD45 was working at the time. Of course, the HEP community did have prior projects aimed at optimising disk performance for physics analysis, but the efforts found in literature (for example [26], [27]), were all concerned with taking a single job and making it as fast as possible, not with considering optimisations for a succession of jobs as we did.

After having identified the physics analysis scenarios in figure 4.2, in which database access becomes more selective through time, we measured the performance of selective reading. We discovered, to our initial surprise, that selective reading would in most cases not out-perform the currently used method of reading the whole physics data set sequentially and throwing away the unwanted data. We concluded that reclustering mechanisms were needed to maintain performance, and were then faced with the problem of developing them. Our reading of [18] indicated that the object database community has little experience with clustering, let alone reclustering, for our scenarios. No known object database offers advanced clustering or reclustering services. Clustering and reclustering optimisations, if used at all, are typically coded by hand, using application-specific knowledge. In view of this information, we decided to proceeded by closely studying the specifics of our application, physics analysis, before starting with the design of clustering and reclustering mechanisms.

The design of the clustering and reclustering mechanisms themselves was a very explorative process, in which many alternative solutions were considered and rejected. Our design was developed concurrently with the measurements supporting it. New measurements were done to either validate tentative design decisions, or to force a choice between alternative design decisions. This approach allowed us to discover the ‘dead ends’ in the design tree as quickly as possible, and it also ensured that we did not waste time on measurements which would be useless in retrospect. We believe that, with a less closer coupling between design and measurements, it would have taken us much longer to obtain the same end results.

7.1 Results and limitations

In this project, we have identified and explored the problem area of object clustering for physics analysis. We have designed storage management and retrieval mechanisms which are robust, and which maintain a high efficiency for typical physics analysis efforts, without placing unnatural constraints on the physics analysis process. The design was validated by measurements on tests implementations.

The optimisation techniques we developed exploit some specific properties of physics analysis. The most important of these properties is that subsequent jobs will access the same collections of objects again and again, with the collections only gradually changing. Another important property we exploited is the lack of data dependencies between operations on individual events. We expect that our techniques will only have limited applicability outside of physics analysis: most database applications do not share the properties above. The reclustering patterns in section 6.5.1 may be applicable in other data mining application domains, provided sufficiently efficient mechanisms for finding the right collections can be built in these domains. Our optimisation mechanisms make it possible predict of the duration of a disk-bound job beforehand. Such predictions are important because they allow users to optimise their demands on the system.

The tests for performance and scalability problems were done on a medium-size hardware configuration (6 processor machine with two disk farms), using the Objectivity/DB database on top of a UNIX file system. Tests with larger hardware configurations and more software layers may reveal additional problems, which are not addressed by the optimisation services we developed.

We did not consider the optimisation of physics analysis jobs which do a lot of writing besides reading. Performance tests have shown that the creation and copying of objects is very CPU-intensive, especially for small objects: Objectivity/DB spends significant CPU resources on various administration tasks for each object and object reference. This often causes jobs which write or copy objects to be CPU-bound, rather than disk-bound, unless they are parallelised. The developments in price/performance ratios for hardware may make this problem disappear over time however: CPU speed is improving with a factor of 1.9 every year [7], disk speed only with a factor of 1.2 [22].

7.2 Future work

Obvious items for future work are performance tests on larger platforms, tests with more software layers, and the development of more fully functional prototypes.

CERN will have an operational HPSS installation at the end of 1997, and this will allow for tests on larger data volumes, and tests spanning more of the software and hardware layers in figure 6.1.

Our prototyping efforts limited themselves to the parts of the design which were most critical for performance. More fully functional prototypes can validate our assertions, made on other grounds, about the feasibility of the collection finding service in section 6.3.2, and the collection management mechanisms in section 6.5.2. Further prototyping may also expose additional bugs and performance bottlenecks in Objectivity/DB, beyond the ones we already found.

Outside of the problem area of object clustering on disks, the problem area of data migration in a multi-level hierarchy, especially in a hierarchy including regional centres, still requires significant study by CMS and RD45. Once good migration mechanisms are found, they will have to be merged with the collection management mechanisms outlined in section 6.5.2.

Acknowledgements

I would like to thank my colleagues in CMS and RD45 for their contributions in terms of discussions, suggestions, and results on which I could base my work. I especially like to thank Ian Willers and Peter van der Stok for supervising this project and for making this project possible. Many thanks to Erco Argante for valuable discussions, and for his help in getting me started in the CERN environment. Finally, thanks to Andrés Kruse for sharing his insights in the physics analysis process.

Abbreviations

AIO	Assistant In Training
API	Applications Programmer Interface
ATLAS	A Toroidal LHC Apparatus
CERN	The European laboratory for particle physics
CMC	CMS Computing
CMS	Compact Muon Solenoid
CPU	Central Processing Unit
ECP	Electronics and Computing for Physics
GUI	Graphical User Interface
HEP	High Energy Physics
HP	Hewlett Packard
HPSS	High Performance Storage System
I/O	Input/Output
LEP	Large Electron Positron collider
LHC	Large Hadron Collider
LHC++	Libraries for HEP Computing ++
LHCC	Large Hadron Collider Committee
MIPS	Mega (10^6) Instructions per Second
OO	Object Oriented
OOTI	Post-graduate programme on Software Design
ODBMS	Object DataBase Management System
ODMG	Object Database Management Group
TUE	Eindhoven University of Technology
RAM	Random Access Memory
RAID	Redundant Array of Inexpensive Disks
RD45	Research and Development project 45 (A Persistent Storage Manager for HEP)
SAI	Stan Ackermans Institute
SLAC	Stanford Linear Accelerator Center

KB	Kilobyte (10^3 bytes)
MB	Megabyte (10^6 bytes)
GB	Gigabyte (10^9 bytes)
TB	Terabyte (10^{12} bytes)
PB	Petabyte (10^{15} bytes)

References

[1] The LEP accelerator at CERN.
General information: http://www.cern.ch/CERN/Divisions/SL/welcome.html

[2] The LHC Conceptual Design Report. CERN/AC/95-05(LHC)
General information: http://wwwlhc01.cern.ch/

[3] ATLAS Technical Proposal. CERN/LHCC/94-43 LHCC/P2
General information: http://atlasinfo.cern.ch/Atlas/Welcome.html

[4] CMS Technical Proposal. CERN/LHCC 94-38 LHCC/P1
General information: http://cmsinfo.cern.ch/cmsinfo/Welcome.html

[5] Using an object database and mass storage system for physics analysis. CERN/LHCC 97-9, The RD45 collaboration, 15 April 1997.

[6] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth. Data Mining and Knowledge Discovery in Databases. CACM Vol. 39, number 11, November, 1996.

[7] CMS Computing Technical Proposal. CERN/LHCC 96-45, CMS collaboration, 19 December 1996.

[8] R. G. G. Cattell, D.K. Barry (eds). The Object Database Standard: ODMG 2.0. Morgan Kaufmann Publishers. May, 1997.
General information: http://www.odmg.org/

[9] Objectivity/DB.
General information: http://www.objy.com/

[10] RD45, A Persistent Storage Manager for HEP.
Project homepage: http://wwwcn.cern.ch/asd/cernlib/rd45/

[11] The BaBar Detector.
Project homepage: http://www.slac.stanford.edu/BFROOT/doc/www/bfHome.html

[12] The Caltech/CERN/HP Joint Project.
Project homepage: http://pcbunn.cithep.caltech.edu/caltech.htm

[13] Libraries for HEP Computing – LHC++
Project homepage: http://wwwcn1.cern.ch/asd/lhc++/index.html

[14] B. Boehm. A Spiral Model of Software Development and Enhancement. ACM SIGSOFT Software Engineering Notes, August 1986.

[15] G. Wiederhold. Database design, 2nd ed. London, McGraw-Hill, 1983. ISBN 0-07-070132-6.

[16] J. L. Hennessy, D. A. Patterson. Computer organization and design: the hardware/software interface. San Mateo, Morgan Kaufmann, 1994. ISBN 1-55860-281-X.

[17] D. A. Patterson, J. L. Hennessy. Computer architecture: a quantitative approach, 2nd ed. San Mateo, Morgan Kaufmann, 1995-1996. ISBN 1-55860-329-8.

[18] R. G. G. Cattell. Object Database Management: Object-Oriented and Extended Relational Database Systems. Revised ed. Addison-Wesley 1994. ISBN 0-201-54748-1.

[19] C. Ruemmler, J. Wilkes. An introduction to disk drive modeling. IEEE Computer 27(3):17-29, March 1994.

[20] LHC Computing Technology Tracking Teams.
General information: http://wwwinfo.cern.ch/di/ttt.html

[21] Pasta - The LHC Technology Tracking Team for Processors, Memory, Architectures, Storage and Tapes. Status Report - August 1996.

[22] Quantum Corporation. Storage Industry History And Trends.
Web site: http://www.quantum.com/src/history/

[23] D. A. Patterson, G. Gibson, R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). Proc. of International Conference on Management of Data (SIGMOD), p. 109-116, June 1988.

[24] The Numerical Algorithms Group Ltd.
General information: http://www.nag.co.uk/

[25] The High Performance Storage System.
Project homepage: http://www.sdsc.edu/hpss/

[26] R. L. Grossman, X. Qin, D. Valsamis, D. Lifka, E. May, D. Malon, L. Price. The Architecture of a Multi-level Object Store and its Application to the Analysis of High Energy Physics Data. Laboratory for Advanced Computing Technical Report, Number LAC 94- R8, University of Illinois at Chicago. December, 1993.

[27] D. Malon, D. Lifka, E. May, R. Grossman, X. Qin, W. Xu. Parallel Query Processing for Event Store Data. Proceedings of Computing in High Energy Physics 1994, p. 239-240.