Available on CMS information server **CMS CR 1998/008**

CMS Conference Report

27 March 1998

Automatic Reclustering of Objects in Very Large Databases for High Energy Physics

K. Holtman, I. Willers

CERN, Geneva, Switzerland

P. van der Stok

Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract

In the very large object database systems planned for some future particle physics experiments, typical physics analysis jobs will traverse millions of read-only objects, many more objects than fit in the database cache. Thus, a good clustering of objects on disk is highly critical to database performance. We present the implementation and performance measurements of a prototype reclustering mechanism which was developed to optimise I/O performance under the changing access patterns in a high energy physics database. Reclustering is done automatically and on-line. The methods used by our prototype differ greatly from those commonly found in proposed general-purpose reclustering systems. By exploiting some special characteristics of the access patterns of physics analysis jobs, the prototype manages to keep database I/O throughput close to the optimum throughput of raw sequential disk access.

To be presented at IDEAS ’98, Cardiff, July 8th. – 10th. 1998

1 Introduction

Several next generation high energy physics experiments at CERN will involve the storage of physics measurements in huge object databases. For example, the CMS experiment will store 1 Petabyte (10^3 Terabytes or 10^{15} bytes) of physics measurements per year starting in 2005 [1], and the COMPASS experiment plans to store 200 TB of data per year starting in 1999 [2]. These data volumes push beyond the limits of current database systems [3]. The bulk of the data, which is of a read-only nature, will be stored on tape robots, with a large disk farm (tens to hundreds of Terabytes) acting as a cache, supporting jobs running on a large CPU farm. On the software side, the system will be based on object technology, in particular on an object database.

Physics analysis is the analysis of stored measurements by teams of physicists. Because of the large data volumes involved, the I/O performance of physics analysis jobs is an important area of concern. In this paper, we discuss a prototype reclustering mechanism which aims to optimise throughput from the disk farm. By exploiting the fact that, in a physics analysis job, the order of traversal for the objects of a single type T is invariant under the selection of subsets of these objects, the prototype manages to keep the database throughput close to the optimum throughput of raw sequential disk I/O. The reclustering is automatic in the sense that it does not need any direct hints from the analysis jobs, and that it can be invoked automatically and transparently based on a few tuning parameters. The prototype is built as a layer on top of the Objectivity/DB database [4].

Many proposed general-purpose object database reclustering mechanisms, for example [5] and [6], optimise the clustering of objects under a workload consisting of small unrelated transactions, where each transaction typically accesses tens of objects. These mechanisms make reclustering decisions based on statistics like object access frequency and link traversal frequency, which are aggregated over many small jobs. Our prototype makes reclustering decisions based on either logs of, or real-time observations of, the exact access patterns of individual jobs. General-purpose reclustering algorithms usually optimise the mapping of objects to database pages or chunks of database pages in order to increase the database cache hit rate. Our prototype aims to optimise both cache hit rate and the pattern of disk reading produced by cache misses. To optimise both, it does not map objects to pages, but to *collections*, which are ordered sequences of objects, each sequence being mapped to pages which are placed sequentially on disk.

2 Data access patterns in physics analysis

A physics analysis job typically performs some calculations over a large set of *events*. An event is an abstraction which corresponds to the occurrence of collisions between particles inside a physics detector. The object database stores a number of read-only objects for each event, as shown in figure 1. Among themselves, these objects form a hierarchy. At the lowest level of the hierarchy is a *raw data* object, which stores all the detector measurements made at the occurrence of the event. For the CMS detector, the raw data object has a size of 1 MB. The objects above that store interpretations of the raw data in terms of physics processes. At each consecutive level in the hierarchy, objects can be thought of as holding summary descriptions of the data in the objects at the level below. At the top of the hierarchy is an *event tag* object of 100 bytes which stores only the most important properties of the event.

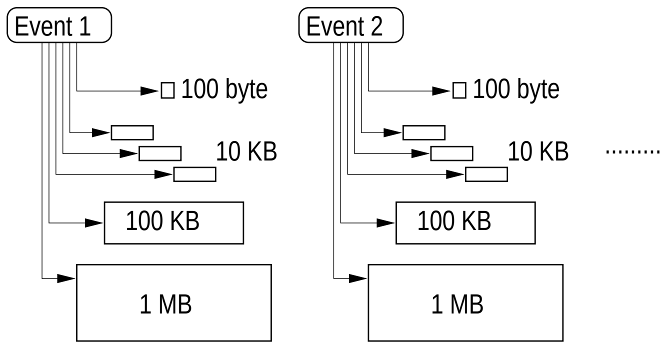

Figure 1: Hierarchy of objects for each event

Given an event (which is generally represented by an event ID), the physics analysis jobs locates one of the corresponding objects through a mechanism provided by the reclustering prototype. This mechanism accepts requests like 'locate the object of type T which belongs to event 2', and returns a handle to (a replica of) the data in this object. In the high-level object model, the processing of this request is equivalent to the traversal of the *has_a* association between an 'event 2' object and a 'T of event 2' object. However, it should be stressed that this *has_a* relation is not present as a single physical object association in the underlying commercial object database.

Each single event corresponds to isolated collisions between particles. In a physics analysis job, events are treated completely independent from each other. There is no special correlation between the time an event collision took place and any other property of the event: events with similar properties will be evenly distributed over the set of all events. A job typically computes some function $f(e)$ for each event e in an event set S. To compute $f(e)$, some of the objects which belong to event e are accessed. Different types of functions f, the results of which are processed in different ways, can be used. In some jobs, f will compute a complex new object which is stored with the event for later analysis. In other jobs, f returns one or more floating point values, and the output of the job is a set of histograms of these values. Finally, f can be a boolean function called a 'cut predicate', with the output of the job being a subset $S' \subset S$ containing all events in S for which the cut predicate returns true.

We assume that, whenever a job traverses an event set S, the individual events are always visited in the same order, which we will call the 'order of reading'. We also assume that the order of reading is unchanged, for the events which are left, if a subset $S' \subset S$ is taken and traversed. In this paper we only consider automatic reclustering services for physics analysis frameworks which satisfy this constraint.

An important part, spanning many jobs, of typical physics analysis effort is concerned with reducing a large initial event set S_1, containing say 10^9 events which correspond to all measurements made by the detector so far, to a smaller set S_n of say 10^4 events whose corresponding particle collisions all displayed some particular interesting property. This reduction of the event set is done in a number of phases. In each phase, a cut predicate p_i is developed, which is then applied to the events in the set S_i of the current phase to compute the event set S_{i+1} of the next phase. In each phase, the development of the cut predicate p_i is an iterative process, in which one keeps refining the predicate until its effect on the event set S_i under consideration are both desirable and well-understood. During this refinement process, a number of jobs will be run over the event set S_i.

The object database access patterns for the subsequent jobs in a physics analysis effort can be approximated as in figure 2, where the bars represent the number of objects of a certain type read by each subsequent job. Each 'step' in the staircase pattern corresponds to a single phase. Figure 2 shows the access patterns on three different types of objects. We see that in the first few phases, when the event set is still large, the jobs only reference the small (100 byte) tag objects for each event: the physicist will put off accessing the larger objects until all possibilities of eliminating events using the smaller objects have been exhausted.

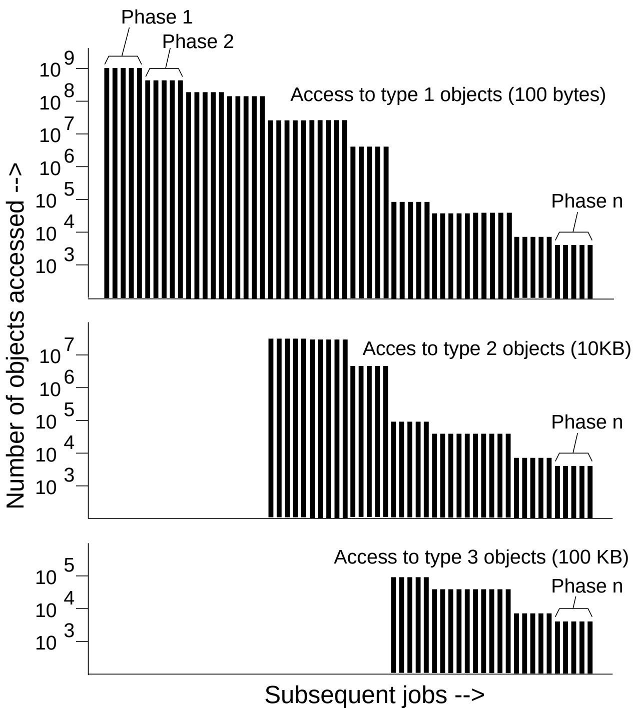

Figure 2: Numbers and types of objects accessed by subsequent jobs in a simple physics analysis effort

It should be noted that figure 2 is in many ways an idealised picture. The widths and heights of the subsequent steps in the staircase pattern will vary wildly for different analysis efforts. Some analysis efforts will split and merge event sets between phases. Some cut predicates will read a variable number of objects for each event: after having read an object of type X, a cut predicate could determine that it does not need to read the type Y object to compute its results. Also, many analysis efforts, all working on different event sets, will be in progress at the same time, so that each type of object will have many independent, but maybe overlapping, access patterns.

We therefore developed the reclustering operations in our prototype to be robust under much more complicated progressions of access patterns than shown in figure 2. In fact, our prototype will optimise any sequence of jobs accessing objects of certain types in a fixed reading order, as long as the access patterns change only occasionally and abruptly.

3 Performance measurements

The effectiveness of reclustering is demonstrated by comparing the performance of prototypes without and with reclustering. We also show some additional performance measurements which guided or backed up our design.

3.1 Performance of prototype without reclustering

We measured the performance of a prototype without reclustering against a simple analysis effort and found the I/O throughput to become very low after a few phases. In our tests, we used events which all had a single object of 8 KB associated with them. The size of 8 KB was chosen because it matched the default page size of the object database we used, and because the handling of objects with sizes around 10 KB was considered most to be the most illuminating case: access to the 100 byte objects in figure 1 could conceivably be optimised by putting all objects in a RAM cache, and for the 100 KB and 1 MB objects the impact of disk seeks on performance would not be as critical.

We put the 8 KB objects sequentially on disk, in the order of reading. The database page size was also 8 KB, so that we had a single object per page. The database was on a disk array containing 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. There was no striping of data across disks, and for every job, none of the data to be read was present in the operating system or database cache.

Figure 3 shows the results of the test run. Each pair of bars represents a single job. The height of the black bar in front represents the size of the event set over which the job was run. The height of the grey bar behind it represents the (wall clock) running time of the job. All jobs were I/O-bound. The jobs were run on a machine which was nearly empty, but not completely empty: this explains the slight variation in runtimes between subsequent jobs in the same phase.

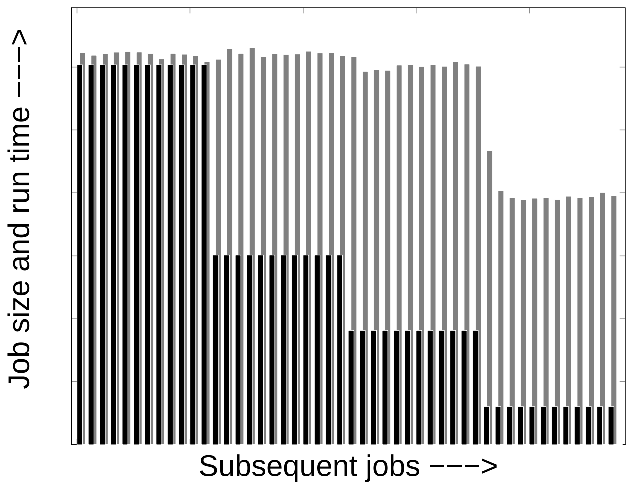

Figure 3: Performance of prototype without reclustering (black bars are number of objects accessed, grey bars are job run time)

In the first phase of the analysis effort, we ran 12 subsequent jobs over an event set containing all events, so that all stored objects were accessed. These jobs produced a nearly sequential database page reading pattern on disk, the only perturbation being an occasional excursion due to the reading of a database-internal indexing page. The jobs in this phase achieved an I/O throughput of about 5 MB/s, not noticeably lower than the maximum throughput of raw sequential I/O for these disks.

In the second phase, we ran 12 jobs over a set containing only 50% of the events, which were randomly chosen. We see that the runtime of these jobs does not drop at all, even though the jobs read only half of the objects (and therefore only half of the database pages). For the next phase, in which the jobs read 30% of the objects, we see the same effect. Only the jobs in the last phase, where 10% is read, are somewhat faster.

3.2 Discussion

The, initially surprising, lack of any speedup in the second and third phases of figure 3 can be understood by considering the time it takes for the disk to seek to the database page holding the next requested object if a few objects are skipped. When only a few objects are skipped, only a few pages are skipped (we have one object per page), and there is a large probability that the requested page is on the same disk track as the previous page that was read, so that the disk head does not need to move to get in the right position. The fastest possible seek time then equals the time needed to wait until the rotation of the disk platter brings the requested page under the disk head, and this time is equal to the time it takes to read all intermediate pages.

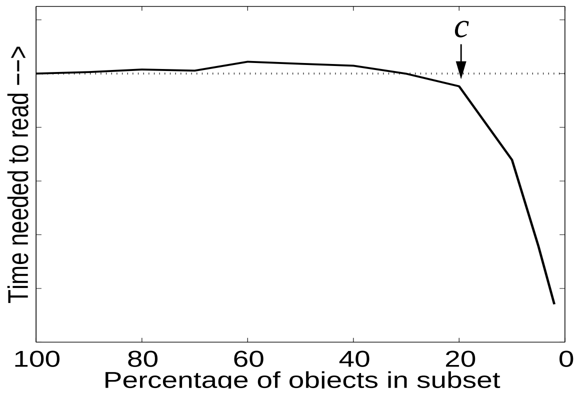

Figure 4: Time needed to read a random subset of objects in a collection of 8 KB objects, traversing the subset in the order in which the objects in the collection are clustered

We found this performance effect to be largely independent of the database page size, the striping of disks, and the brand and type of disk. Figure 4 shows the effect in more detail for 8 KB objects: for all but very small subsets, the time to read the objects in the collection is a constant, determined by the size of the collection only. The position of the cutoff point c, at which reading becomes faster for smaller subsets, varies mainly as a function of the average size of the objects in the collection. For average object sizes of 64, 8, and 1 KB, our disks had c values of 50, 20, and 2%. If c is lower, the potential benefits of using a reclustering optimisation are higher. Though c does not vary much depending on the brand of disk used, the trend is that c values are going down for more modern disks. This is because newer disks tend to have more data on a single track, leading to a higher probability that the page holding the next requested object is still on the same track.

3.3 Performance of prototype with reclustering

Figure 5 shows the performance of the prototype which implements our automatic reclustering service, running the same sequence of jobs as in figure 3. The wide grey bars in figure 5 represent the running of the 'batch' reclustering algorithm (see section 4.2), which, in this scenario, is run between jobs. The runtime of the batch reclustering algorithm is represented by the surface, not the height, of the wide bars: they are twice as wide as the other bars.

At the start of the test run, the objects in the database were clustered in the same way as for the prototype without reclustering, and the first 12 jobs again achieve a throughput not noticeably lower than the maximum throughput

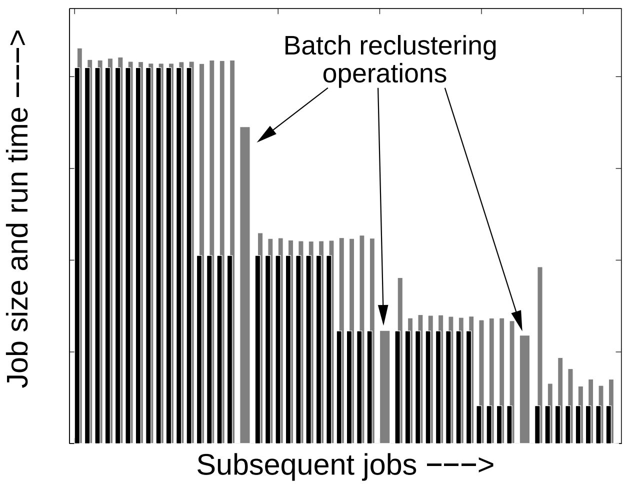

Figure 5: Performance of prototype with automatic reclustering (black bars are number of objects accessed, grey bars are job run time, wide grey bars represent running of 'batch' reclustering algorithm)

of sequential I/O. For the next 4 jobs, the running time is again similar to that in figure 3. Then, the first 'batch' reclustering operation is run. The operation examines the logs of the access patterns of the preceding jobs and reclusters the database to optimise for these patterns, as shown schematically in figure 6. The mechanisms behind batch reclustering is covered in more detail in section 4.2. Here we just note that the effect of batch reclustering is that the running time of the next 8 jobs, which access exactly the same data as the preceding 4 jobs, has improved: it is nearly proportional to the amount of data accessed, yielding a throughput close to that of sequential I/O. The spikes in job runtime immediately after the running of the last two reclustering operations are caused by the emptying of the operating system write cache (which was filled by the preceding reclustering operation) during the run, we did not pause between jobs.

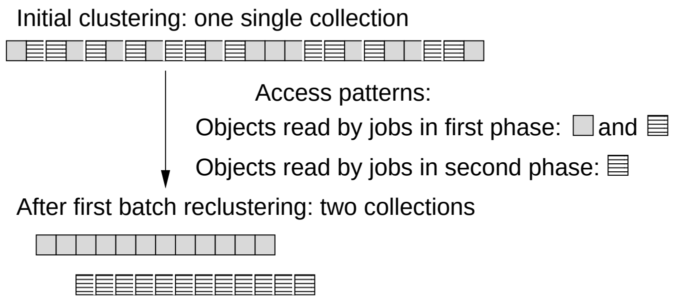
Initial clustering: one single collection
Access patterns:
Objects read by jobs in first phase: and
Objects read by jobs in second phase:
After first batch reclustering: two collections

Figure 6: State of the database before and after the first batch reclustering operation in figure 5

3.4 Read-ahead optimisation

After the first batch reclustering operation in figure 5, the jobs of the second phase will traverse only one of the two collections shown at the bottom of figure 6. Another job in a new, independent analysis effort may however need to read all data from both collections. As both collections are individually clustered in the reading order, such a job would have a logical object reading pattern as shown on the left in figure 7. If such a logical reading pattern were performed directly on a disk array, the throughput would be very low due to the many long seeks between collections.

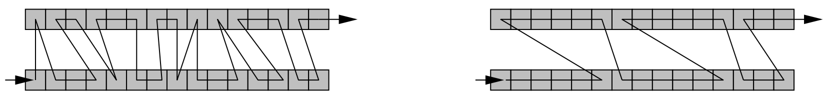

Figure 7: A logical access pattern to two collections (left) and the physical pattern after read-ahead optimisation (right)

To achieve near-sequential throughput, the logical pattern needs to be transformed into the physical pattern at the right of figure 7. To make this transformation without changing the iteration logic of the job itself, object data needs to be read ahead into some form of buffer memory. In tests, we found that such a read-ahead optimisation was not performed by the commercial object database [4] on which we built or prototype, nor by the operating system (we tested both Solaris and HP-UX), nor by the disk hardware (for various commodity brands). We therefore implemented the optimisation ourselves: the collection accessor class was extended to read ahead objects into the database cache, thus producing the physical pattern in figure 7.

It should be noted that the above case, in which we had to optimise the physical access pattern ourselves, is different from the case in which two independent processes are each traversing a single collection. In that case, both operating systems we tested will automatically keep the number of long seeks due to context switches at an acceptable level by allocating relatively long timeslices to each process. We therefore only needed to optimise access patterns by hand at the level of a single process.

Measurements (see figure 8) showed that, on our disks, when 800 KB worth of objects were read ahead for each collection, the I/O throughput approached that of sequential reading.

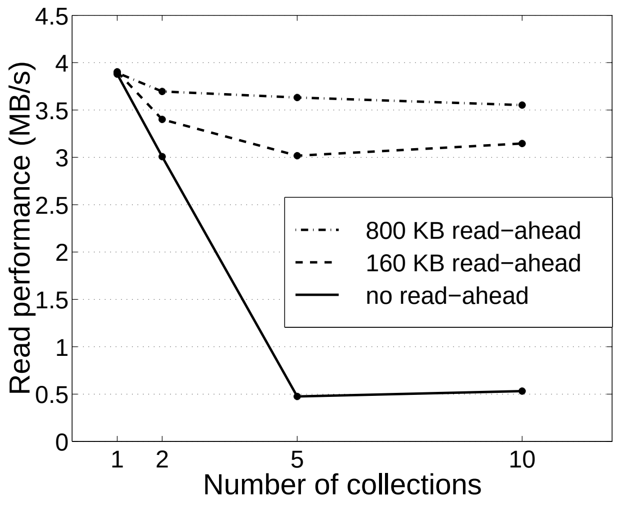

Figure 8: Read performance when simultaneously traversing multiple collections located on the same disk, with different read-ahead strategies

By sacrificing a modest amount (given current RAM prices) of memory, the read-ahead optimisation prevents a performance degradation if two or more collections produced by batch reclustering are accessed simultaneously. This gives us significant freedom in batch reclustering. In addition, the read-ahead optimisation allows us to manage the clustering of different types of objects in an independent way: we can maintain different sets of collections on a type by type basis, and optimise the reading of objects of one type without taking into account a possible interference due to the reading of objects of any other type. Of course, there is a scalability limit on the number of collections which can be traversed at the same time. For our current prototype, overheads grow too large above 20 collections. This has some consequences for batch reclustering, which are discussed at the end of section 4.2.

4 Types of reclustering

Our prototype performs two different types of reclustering, called 'on-the-fly' and 'batch'. Each type is invoked at a different time, and reorganises objects in a different way. Both have in common that they consider collections of objects of a single type only. If a physics analysis effort accesses many types of object per event, each of these types is reclustered independently. Thus, during a job which accesses three types of objects, three different reclustering operations may be going on independently.

4.1 On-the-fly reclustering

The basic operation of on-the-fly reclustering is shown in figure 9: the objects in a larger collection which are requested by a job are copied to a new collection. Note that in both individual collections, the objects are clustered in the order of reading. On-the-fly reclustering is invoked automatically, transparently to the job itself, whenever it can be done with a price/performance factor better than a certain threshold set by the user. In this factor, the price is the time needed for the copying of the selected objects, and the performance is the gain in runtime for a single subsequent job which selects exactly the same objects. Typically, on-the-fly reclustering is only invoked when there is an immediate large benefit, say if the performance outweighs the price by at least a factor 4.

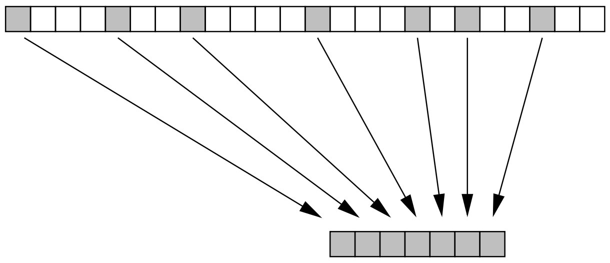

Figure 9: On-the-fly reclustering of selected objects to a new collection

After on-the-fly reclustering, there are two copies of each selected object. This is not a problem as far as our application is concerned: inconsistencies cannot emerge because the objects hold read-only data. In our prototype, the existence of two copies is invisible to the analysis jobs: if a job requests some object which belongs to an event, it will simply get the object identifier of one of the copies. However, the existence of two copies does pose an optimisation problem: if the job requests a copied object, which of the copies should be read to get the highest I/O performance? This optimisation problem is solved, again transparently to the job, by the object access and indexing mechanism discussed in section 5.

An on-the-fly reclustering operation will allocate extra disk space in order to increase efficiency. This space can be recovered, however, without losing the efficiency, by running a batch reclustering operation later. An example of this is shown in figure 10.

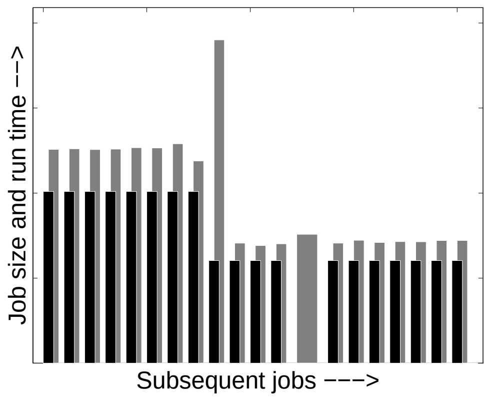

Figure 10: On-the-fly reclustering operation, triggered by the occurrence of a new access pattern, followed later by a batch reclustering operation

4.2 Batch reclustering

A batch reclustering operation will re-arrange all objects of a particular type T to create an optimal clustering for the access patterns of recent jobs. Also, any duplication of objects caused by on-the-fly reclustering operations will be eliminated. A batch reclustering operation is typically executed whenever there are some free system resources to spare. In our implementation, the database is not locked during batch reclustering: physics analysis jobs can still access objects of type T while reclustering is in progress.

Whenever a physics analysis job accesses some objects of a type T, a log is produced which records the exact set of objects of type T accessed by the job. To recluster the objects of type T, a batch reclustering operation will first process the recent logs to generate a view of all recent access patterns to objects of type T in terms of possibly overlapping sets. Such a view is shown on the left in figure 11. The view in this figure corresponds to a situation in which the access logs show exactly two different access patterns for objects of type T, produced by jobs in two independent analysis efforts. In the set view, four different regions are present. After computing the set view, the batch reclustering operation will create a new collection for each region in the view, each collection containing all objects in its region. This is shown on the right in figure 11. To create the new collections, the currently existing collections with objects of type T are accessed. After creation, the collection index is updated to point to the new collections, allowing new jobs to use them. Finally, after any existing jobs which are still using the old collections have run to completion, the old collections are deleted, thus producing a database without any duplication of objects. Our batch reclustering engine contains an optimiser which will suppress the reclustering of data from old collections, and the subsequent deletion of these collections in cases where this reclustering action would produce little or no performance gain for the access patterns under consideration.

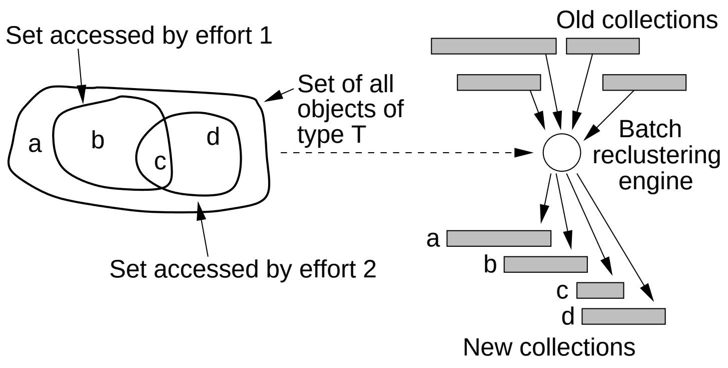

Figure 11: Batch reclustering

After batch reclustering, the collections can be used by subsequent jobs as shown in figure 12. A job in a completely new analysis effort will usually have to read objects from all four collections, and could invoke on-the-fly reclustering on any four of the collections while doing so.

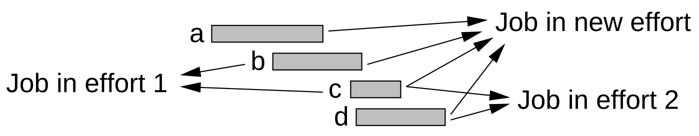

Figure 12: Use of batch reclustered collections by jobs in different analysis efforts

One important limitation of the batch reclustering scheme is that it will not scale well in the number of different access patterns for which reclustering is done. In the worst case, n different patterns will produce 2^n independent collections after batch reclustering. For our current prototype, we found that going beyond switched access to 20 collections per type produced overheads which were too large. Thus, the batch reclustering scheme in our prototype is limited to optimising a maximum of 4 independent access patterns.

The question of how reclustering can best be done for more than 4 independent patterns is a subject for future research. With more than 4 patterns, a reclustering scheme will have to make a tradeoff between I/O throughput and database size, a tradeoff which we avoided having to make for batch reclustering.

5 Access to reclustered data

When an analysis job is accessing reclustered data of a certain type T, two distinct mechanisms are involved: the *access engine* and the *optimiser* (figure 13).

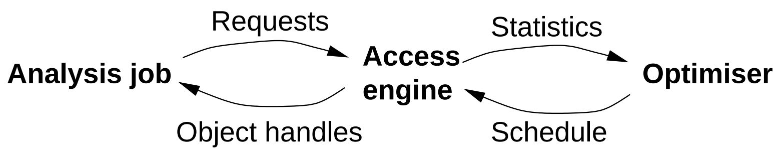

Figure 13: Communication between different parties involved in access to reclustered data

The access engine works on the level of individual objects: it receives the subsequent requests, of the form 'locate (a copy of) the object of type T belonging to event 2' made by the analysis job, locates these objects, takes care of reading them efficiently into cache memory, and returns their object handles to the job.

The optimiser works at the job level, creating a schedule for the access engine. Whenever the access engine receives a request for an object of which multiple copies exist in different collections, it will consult the schedule to determine which copy should be read in order to maximise I/O throughput. The schedule also controls possible on-the-fly reclustering actions.

5.1 The access engine

The interface between the physics analysis job and the access engine (and through it, the optimiser, see figure 13) is a very narrow one. Access engines are instantiated on a per-job and per-type basis: the access engine is bound to a type T on instantiation and receives notification of the start and end of its corresponding job. Each request from the job only includes an *event index number* identifying the event for which an object of type T is requested, and the job will not issue its next request before an object handle for the current request is returned. This narrow interface makes it necessary for the optimiser to use statistical methods for predicting future requests from the job, so that they can be optimised, even though some information about future requests is usually available inside the job. We chose for a narrow interface, rather than a wide one in which the job would give some advance information to the scheduler, to make our prototyping efforts more meaningful as an exercise in exploring the possible limits of reclustering schemes which are fully transparent to the user.

To cope with the lack of advance information about access patterns, the access engine works in two phases. Statistics gathered in the first, short, phase are sent to the optimiser, which returns a schedule to optimise I/O throughput in the second, longer phase. The complete sequence of events is as follows:

- *Initialisation*: Locate all collections of objects of type T.
- *Phase 1*: (First few hundred requests from the analysis job) Locate the subsequently requested objects in the collections. If a requested object has copies in multiple collections, choose a copy with a simple tie-breaking algorithm. Meanwhile, gather statistics for the optimiser.
- *Transition*: Send statistics to the optimiser. Receive schedule. Perform on-the-fly reclustering for the data requested in phase 1, if necessary, according to instructions from the optimiser.
- *Phase 2*: Handle all subsequent requests, performing on-the-fly reclustering, if necessary, according to instructions from the optimiser. If a requested object has copies in multiple collections, choose one according to the schedule.

This two-phase approach works well because events with similar properties are distributed evenly over the database. This makes it possible to make a good schedule based on the statistics for the few hundred initial requests.

The event index numbers, which identify objects in requests to the access engine, are to be assigned by the job which creates the objects initially. An obvious choice for these numbers are the *event numbers* which are assigned to events when they are measured by the particle physics detector, but this is not strictly necessary. In our prototype, an important constraint is that the sequence of event index numbers must be increasing in the order of reading. This constraint allows for very fast collection indexing calculations in the access engine, and also simplifies the merging of collections when reclustering. An index into a collection is simply a sequence of pairs, one for each object in the collection, with each pair containing the object ID of its corresponding object and the number of the event to which the object belongs. The pairs in the sequence are themselves clustered together sequentially, in the order of reading. The access engine maintains a single pointer into the sequence, which is advanced to keep pace with the progress of requests from the job.

5.2 The optimiser

The optimiser computes a schedule based on statistics gathered during phase 1 of the running of the access engine. These statistics consist of

1. the number of collections of objects of type T located by the access engine
2. a view of the ways in which the sets of objects in these collections overlap, and the sizes of the overlapping and non-overlapping parts,
3. for each overlapping or non-overlapping part, the percentage of the objects in this part which were requested by the analysis job.

Figure 14 shows an example of such statistics, in a graphical representation. The figure shows 4 collections, with 6 distinct (non)overlapping areas, each area labelled with the percentage of objects requested by the analysis job.

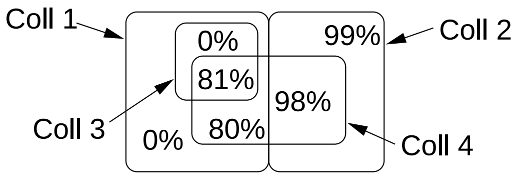

Figure 14: Example of statistics used by the optimiser

Using these statistics, the optimiser will compute a set C of collections, such that the collections in C together cover all areas in which some objects were requested by the job, and such that the sum of the sizes of the collections in C is minimal. For the statistics in figure 14, this computation yields $C = \{\text{Coll } 2, \text{Coll } 4\}$.

If, during running phase 2 of the access engine, a requested object has copies in multiple collections, two tie-breaking rules are used to optimise I/O throughput. In order of priority, these are:

1. reading from a collection in C is preferred,

2. reading from a smaller collection is preferred.

In the normal case, that is if the ‘0%’ statistics about usage of parts not in C are correct predictions for the whole job, the second rule above will not be used by the access engine: following the first rule only it will just traverse all collections in C. Because of the read-ahead optimisation used, the time needed for this traversal is proportional to the sum of the sizes of these collections, which is the quantity minimised by the optimiser when calculating C.

In calculating C, the optimiser is solving an instance of the *set covering problem*, which is NP-complete, but for which good approximation algorithms exist [7],[8]. In our prototype, the instances of the problem were so small that we could always afford to compute the optimal solution using a straightforward branch-and-bound algorithm.

The optimiser also uses the statistics to compute whether it is possible to perform an on-the-fly reclustering operation which satisfies the price/performance criteria (see section 4.1) set by the user. In the schedule for the access engine, the instructions about on-the-fly reclustering take the form of a possibly empty set R of areas in the statistics. Whenever the job requests an object which is contained in an area in R, the access engine will recluster it by copying it into a new collection. In our prototype, we implemented a straightforward calculation which determines if any non-empty R will produce a reclustering operation which satisfies the price/performance criteria set by the user. If there are many sets R which do, then the one which reclusters the largest number of objects is chosen for the schedule. If none do, then the R in the schedule will be the empty set. Unfortunately, this straightforward calculation takes too long (more than a few seconds) in some cases. These cases can be easily identified before the start of the calculation, and we plan to run an approximation algorithm for these cases in our followup prototype.

6 Conclusions

We have described the prototype of a specialised automatic reclustering system, which exploits some specific properties of disk-bound physics analysis jobs to maintain an I/O performance near the optimal performance of sequential disk I/O. The properties which were exploited are the high number of objects requested in each job, the read-only nature of the objects, the fact that events with similar properties are distributed evenly over the database, the fixed reading order, and the existence of sequences of jobs with the same access pattern.

Our prototype is based on a mechanism for clustering objects into collections, and accessing these collections with a read-ahead optimisation. The read-ahead optimisation allows us to manage the clustering of different types of objects in an independent way, and also makes it possible for the batch reclustering operation to conserve the database size while preserving optimal throughput. The objects are retrieved through a fast access engine, which uses a schedule to optimise throughput. The schedule only needs to be computed once for every job, and this allows the use of fairly complex computations in constructing the schedule.

We have shown performance measurements for the prototype and contrasted them with the performance of a prototype without reclustering. We discussed a scalability limit in the number of different access patterns for the batch reclustering scheme used by our prototype, and identified the tradeoff between I/O speed and database size which will have to be made to scale beyond this limit. Subjects for future research are to increase the scalability in the number of access patterns, and an extension of the optimiser to cover the migration and replication of collections between different storage levels.

Acknowledgements

The research reported on in this paper was done in the CMS computing group, in the context of the RD45 collaboration. It was supported by grants from the Stan Ackermans Institute at Eindhoven and by the CERN technical and doctoral student programme.

References

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

[2] The COMPASS proposal, Addendum 1. CERN/SPSLC 96-30 SPSLC/P297 Add. 1, the COMPASS collaboration, 20 May 1996.

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

[4] Objectivity/DB version 4.0.2. Vendor homepage: http://www.objy.com/

[5] William J. McIver Jr., Roger King: Self-Adaptive, On-Line Reclustering of Complex Object Data. SIGMOD Conference 1994: 407-418

[6] Jia-bing R. Cheng, A. R. Hurson: Effective Clustering of Complex Objects in Object-Oriented Databases. SIGMOD Conference 1991: 22-31

[7] V. Chvatal. A greedy heuristic for the set-covering problem. Math. of Oper. Res., 4:233–235, 1979.

[8] T. Grossman and A. Wool. Computational experience with approximation algorithms for the set covering problem. Euro. J. Operational Research, 101(1):81-92, August 1997.