A Cache Filtering Optimisation for Queries to Massive Datasets on Tertiary Storage

Koen Holtman
CERN – EP division
CH - 1211 Geneva 23, Switzerland
Koen.Holtman@cern.ch

Peter van der Stok
Eindhoven University of Technology
Postbus 513, 5600 MB Eindhoven, The Netherlands
wsstok@win.tue.nl

Ian Willers
CERN – EP division
CH - 1211 Geneva 23, Switzerland
Ian.Willers@cern.ch

Abstract

We consider a system in which many users run queries to examine subsets of a large object set. The object set is partitioned into files on tape. A single subset of objects will be visited by multiple queries in the workload. This locality of access creates the opportunity for caching on disk. We introduce and evaluate a novel optimisation, cache filtering, in which the 'hot' objects are automatically extracted from the files that are staged on disk, and then cached separately in new files on disk. Cache filtering can lead to complex situations in the disk cache. We show that these do not prevent effective caching and we introduce a special cache replacement algorithm to maximise efficiency. Through simulations we evaluate the system over a broad range of likely workloads. Depending on workload and system parameters, the cache filtering optimisation yields speedup factors up to 6.

1 Introduction

Massive datasets are used in a number of scientific endeavours, for example in High Energy Physics (HEP), earth and climate studies, and astronomy. Such datasets are typically produced by high-bandwidth scientific instruments (particle physics detectors, satellites) or by simulations. With scientists taking advantage of lower media and hardware costs, the sizes of such datasets are continuously growing. For example, the BaBar experiment is storing about 200 Terabytes of physics data per year [6], and the CMS experiment plans to store 1000 Terabytes (1 Petabyte) per year starting in 2005 [3]. Storing a large dataset on tape is some 10 times cheaper than storing it on disk [4]. For massive datasets, the use of tertiary storage remains the only cost-effective option in the foreseeable future.

The work reported on here is part of a larger research project aimed at exploring database technology options for the storage and analysis of massive high energy physics datasets [9]. Our design was driven by high energy physics requirements. The resulting optimisations can be applied in many areas where large datasets are used. In section 6 we investigate performance for both physics workloads and more generic workload scenarios.

We consider large datasets which are structured as sets of individually addressable data objects. In high energy physics, such an object typically has a size between 1 and 100 KB and describes some aspect of a single high energy particle collision. Datasets will often grow while being used, but once an object is written to the set, the object becomes read-only. A typical query computes a function $f(o)$ for all objects o in some subset S of all objects in the dataset. We assume that this object subset S has been computed and stored (as a set of object identifiers) in advance of query execution, for example with a query on an 'object summary' or 'tag' database which is on secondary storage. We also assume that the I/O profile of queries is dominated by the reading of the objects $o \in S$, and ignore any I/O connected to the processing of the function results. We assume that the order in which the function results $f(o)$ are computed is not significant to further processing: this allows for massive parallelism in processing and creates important degrees of freedom in I/O scheduling.

2 Baseline system

In this paper we compare two systems: the *baseline system* and the *cache filtering system*. We created the baseline system by synthesising elements from currently existing and proposed systems for storing and analysing massive datasets [2] [5] [8]. This section describes the baseline system, except for staging and cache replacement algorithms, which are discussed in sections 4 and 5. We also introduce some terminology. For reasons of style, we refer to tertiary storage as 'tape' and to secondary storage as 'disk' from now on.

2.1 Storage organisation

The system stores a large set of individually addressable objects. To create files on tape, this object set is partitioned with an *initial clustering algorithm* into *chunks*, with each chunk stored in a *file* on tape. These files, the *original chunk files*, are the only files present in a freshly initialised system. The file is the unit of transfer between disk and tape. We define the staging of a file as a copy operation: staging of the initial chunk file of chunk C will create a *second* file containing all objects in chunk C on disk. This is illustrated in figure 1.

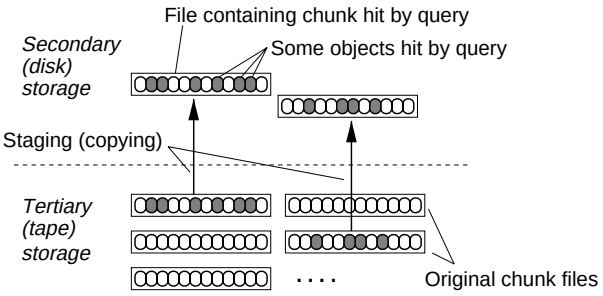

Figure 1: Storage organisation in the baseline system

In the baseline system, there are at most two files for every chunk C, one file on disk and one file on tape, and these files always contain the complete set of objects in chunk C. In the cache filtering system (see section 3), there can be many files for a chunk C on disk, and these files can also contain subsets of the objects in chunk C. Thus, the terminological distinction between a chunk and a file becomes important.

2.2 Query execution and subqueries

Execution of a query, which computes a function $f(o)$ for all objects o in some set S, proceeds as follows. First, all chunks $C_1, \ldots, C_n$ which contain objects in S are identified. We say that these chunks are *hit* by the query. It is assumed that identifying the chunks which are hit is a fast operation: usually the object identifiers in the representation of S contain the chunk number within them. Then, for every chunk C_i, a *subquery* i is created that computes $f(o)$ for every $o \in S \cap C_i$. A subquery thus always works at the level of a single chunk. If a file containing the chunk C_i, from which the subquery needs to read objects, is not already on disk, the subquery will block until this file is staged. Staging is shown in figure 1, the objects which are hit are grey.

2.3 The initial clustering

We call the assignment of objects to chunks, and initial chunk files to tapes, the *initial clustering* of the dataset. We assume that this initial clustering remains fixed over the dataset lifetime.

The construction of good initial clusterings for various data types and workloads is an active area of research [2] [10]. Such research is not necessarily aimed at datasets on tape, but often applies to any storage level. Initial clustering algorithms for n-dimensional spatial or spatio-temporal datasets usually interpret the objects in the dataset as points in some n-dimensional space. They then try to put objects which are close together in this space in the same chunk, and chunks with close-by objects near each other on the same tape. This is a natural approach when queries analyse sets of objects located in a particular region of space. For other types of datasets and workloads, one generally tries to map objects into a similar n-dimensional space, using object attributes to construct object coordinates.

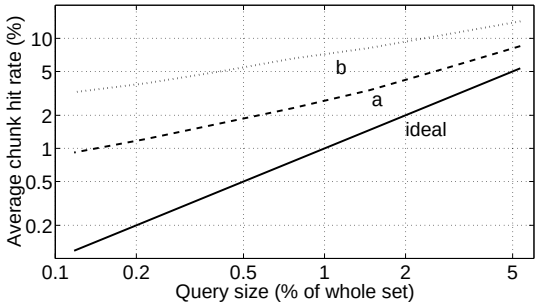

Figure 2: Performance characteristics of three initial clusterings

Figure 2 shows the characteristics of three possible initial clusterings. The curves plot the percentage of chunks which is hit by a query of a certain size. In the ideal case (the ideal curve), a query on $n\%$ of the objects hits exactly $n\%$ of the chunks. In practice, this ideal case can only be achieved if the number of possible queries is very low and very predictable. Creating many, very small chunks to improve initial clustering efficiency is self-defeating: with many small chunk files, the tape system will loose too much performance due to per-file overheads [1].

In figure 2, the curves (a) and (b) above the ideal curve show the performance of more realistic combinations of initial clusterings and query workloads. Curve (a) was constructed as a 'best realistic case', by simulating queries on a two-dimensional dataset. These simulated queries always inspect sets of objects lying within a rectangle in the space, and the the initial clustering algorithm 'tiles' the dataset space into 200 chunks. Curve (b) was constructed for a case where queries and initial clustering have a greater mismatch: it is more representative of high energy physics workloads.

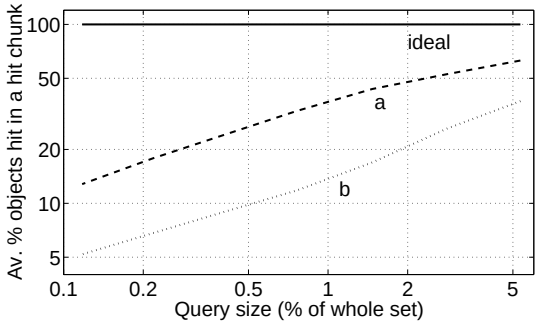

Figure 3: Object hit rate for three initial clusterings

From the chunk hit rate curves in figure 2, one can compute curves for the average object hit rate *inside* the hit chunks: this is the size of the query divided by the sum of the sizes of the hit chunks. Curves for this object hit rate are plotted in figure 3. We observe that both the realistic clusterings (a) and (b) often yield a low object hit rate inside the files being retrieved from tape. If these files are cached completely

on disk, we are wasting a lot of disk space on storing the unneeded 'cold' objects which also happen to be in the hit chunk files. Based on this observation, we developed the *cache filtering* optimisation.

3 Cache filtering system

The cache filtering system, shown in figure 4, takes the baseline system (figure 1) and adds *cache filtering* to it. In a cache filtering action, we copy 'hot' objects from a larger file on disk to a smaller file. This is illustrated in figure 4, the hot objects are the grey ones. When a smaller file has been created, the original larger file on disk can be deleted. The end result is that disk space previously occupied by the cold (non-queried) objects is freed, while all hot (queried) objects are still present. Thus, we can effectively cache more hot objects on disk. As shown in figure 4, cache filtering can be done recursively: if a new query yields a smaller set of hot objects, a still-smaller file can be made, and the larger file can be deleted.

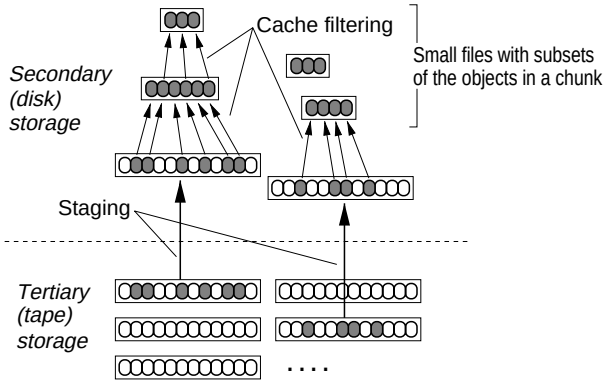

Figure 4: Storage organisation in the cache filtering system

Cache filtering is done at the chunk level: a small file created in cache filtering always contains a subset of the objects in one chunk. This strict chunk-level separation makes indexing and scheduling tasks much more manageable.

3.1 Implementation of cache filtering

Cache filtering is performed by subqueries, while they read their needed objects from disk and feed them to the query function. The actions of a subquery, after it is created, are as follows. First, it determines whether all objects it needs are present on disk. Note that cache filtering, combined with cache replacement, can create a situation where only some of the objects needed by the subquery are on disk. If not all needed objects are on disk, the subquery will block, waiting for the original chunk file, which contains all objects it needs, to be staged.

Once a subquery has established that all objects needed by it are on disk, it invokes an optimiser to compute a schedule which controls subsequent actions. This schedule specifies the set of files on disk which should be read by the subquery.

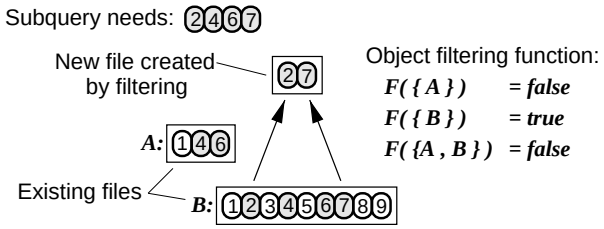

Figure 5: Example of a cache filtering function

This set of files is chosen so that the files together contain all objects needed by the subquery. If many choices are possible, the set for which the sum of the file sizes is smallest is chosen. We minimise the sum of the file sizes to minimise disk efficiency losses because of sparse reading, and, more importantly, to yield the best possible input for the cache replacement algorithm (section 5), in which file access statistics play an important role.

The cache filtering actions of a subquery are governed by a *cache filtering function* F. This function yields, for every specific object read by the subquery, the (boolean) decision whether to filter the object to a new file. The function is computed by an optimiser just before the subquery starts its iteration over objects. By using the cache filtering function, the subquery can make sophisticated object-level decisions without performing expensive calculations for each object encountered. As its input, the function takes, for every particular object, the set of all files on disk in which this object is currently present. Figure 5 shows an example of a cache filtering function, in this case there are two existing files (A and B) on disk which contain some objects hit by the subquery, and a third file is filtered from them. The function values $F(\{A\}) = false$ and $F(\{A, B\}) = false$ prevent the objects already in the small file A from being duplicated on disk.

With n files containing objects needed by a subquery on disk, there are $2^n - 1$ possible inputs for a cache filtering function (2^n minus one because the empty set can never occur as an input). The function can yield *true* or *false* for any of these inputs, so there are $2^{(2^n-1)}$ possible filtering functions. Finding the the best filtering function ('best' being defined by a reasonable costing function) is an NP-complete problem [7]. n is usually below 10, but we have seen it grow much larger in some of our simulations. Nevertheless we saw that our optimiser could almost always find the best filtering function (for our costing function), using a branch-and-bound search of the solution space after a problem transformation, in under 4 seconds of CPU time. When the optimiser cannot complete the search in the alloted 4 seconds, it aborts the search and the best function found up till then is used to control filtering.

Our optimiser calculates the cost of a filtering function as

*2 * 'the size of the filtered file that is created'*

+ 'the smallest sum of sizes of any set of files which could be read after filtering, if the same subquery were executed again, to obtain all objects needed by the subquery'.

The functioning of this cost function is best understood in economic terms. The first term represents the cost of the filtering action, the *investment* needed to create a new cache configuration. The second term represents the *running costs* of the new configuration. These are the costs of being able to service, from disk, any possible future queries on the same object set, that is the costs of storing the minimal set of necessary files on disk. The best filtering function minimises both the investment and the running costs. The multiplication factor of 2 determines how the two should be balanced. It reflects a property of the workload: the lower the factor, the longer we can expect a new investment to yield savings in running costs. In a tuning exercise we found that a value of 2 worked well for our workloads, while a factor of 4 worked almost equally well.

As an example, if there is only one existing file with objects for a chunk on disk, there can be two possible filtering functions: one specifying no filtering at all, the other one specifying that all objects needed by the subquery should be filtered. With the above costing function, the function which filters the objects is chosen as the best one if the subquery hits less than $1/3$ of the objects in the existing file.

4 Staging policy

At any point in time, many subqueries can be blocked, waiting for for their particular original chunk file to be staged from tape. Note that multiple subqueries, belonging to different queries, may be blocked waiting for the same file. Our file staging mechanism makes sure that all subqueries which are waiting for a particular file can run after the file is staged, and before it is deleted from disk again by cache replacement. This so-called *query batching* can lead to dramatic tape I/O savings [11], especially for workloads with many concurrent large queries.

The staging policy has to make a tradeoff between (1) minimising mount and seek operations and (2) optimising the order in which subqueries are serviced. To investigate the tradeoff between (1) and (2), we used simulation to determine the performance of various staging policies, under the (unrealistic) assumption that there are no mount and seek overheads. This assumption eliminated the effects of staging policies on (1), so that we could study the consequences of various policies on (2) in isolation. We found, to our surprise, that the choice of staging policy had very little effect on performance in these simulations: even a purposely bad policy (least work first) would only decrease performance with a factor 1.2 compared to the best policy found.

Based on the above results, we implemented a simple staging policy, both for the baseline and the cache filtering system, which ignores the considerations of (2) and just aggressively minimises tape mounts and seeks (1). The policy cycles over all tapes in the system in a fixed order. When the next tape is reached, and a tape drive becomes available for reading, the stager inspects the currently blocked subqueries to see if any of them need a file on that tape. If so, the tape is mounted. Then, all files which have blocked subqueries waiting for them are staged in the order of their position on tape. This results in a sequential pattern of seeking and reading on the tape. When the last file has been staged, the tape is rewound and unmounted. Note that this policy works with any number of tape drives. The fixed cycling order ensures that subqueries are never blocked indefinitely.

5 Cache replacement policies

Our cache replacement policies make decisions at the file level, as true object-level decisions would be too expensive to compute. Also, object-level cache replacement might lead to extreme fragmentation of data on the disk farm, degrading disk performance to that of random disk access. Such a fragmentation risk is absent with our file-based replacement policies: related hot objects are kept packed together in a few files, and these files can be accessed using sequential or sparse reading.

Through simulations, we compared and developed a number of cache replacement policies. We found that classical LRU, in which the Least Recently Used file is deleted when more space is needed, works very well with our workloads for the baseline system. For the cache filtering system, we settled on a novel replacement policy which is introduced below.

5.1 Usage Based Cooling (UBC) cache replacement

Our workloads (section 6.1) contain queries which are larger than, or about as large as, the size of the available disk cache. These queries can cause 'thrashing' effects, in which most of the cached data is replaced from the cache by newer data before it can be hit again by a subsequent query. To guard against thrashing, we designed a policy which assigns different replacement rates to files, depending on the circumstances in which they are used. The basic idea is that, if a file is staged from tape, or created by a subquery that was blocked on tape, it should be replaced from cache soon, except when it is also used, not much later, by a subquery that was not blocked on tape.

Below we present the UBC algorithm, with tuning parameters optimised for the cache filtering system, for which we show performance results in section 6. In UBC cache replacement, any use of a file adds a 'heated token' to the file. A heated token has an initial temperature t_i and a cooling rate c. When it is t units of time old, its temperature will have dropped to $\frac{t_i}{c \cdot t+1}$. This cooling function reflects a physical cooling process, similar decreasing functions would also work. Tokens added in different types of use get different t_i and c values. Specific values were chosen in a tuning exercise; we found that performance is not very sensitive to

values: for example where we mention a factor of 40 below, we found that a factor of 200 would work almost as well.

The coldest file is replaced first. We define the temperature of a file as the temperature of its hottest token plus half the temperature of its second-hottest token. This (somewhat arbitrary) temperature definition was found to work best in a tuning exercise, just taking the temperature of the hottest token also works well. If a subquery reads a file, it adds a 'standard' token with $t_i = 1$ and $c = 1$. An additional token twice as hot ($t_i = 2$) and cooling twice as slowly ($c = \frac{1}{2}$), is added to a file if immediate re-execution of the subquery would cause the file to be read again. (Some different files will be read on re-execution if cache filtering is done.) The effect of this is, for example, that a new file created by cache filtering lasts about four times as long as the file(s) read to create it. The cooling rate c of an added token is multiplied by 40 if the subquery was blocked on the staging of a file. This faster cooling rate causes an object which has been freshly staged to remain in the cache for a short while only, unless the object is quickly accessed again by a subsequent query. If five queries are run subsequently on a small set of objects, the first query stages these objects and adds fast-cooling tokens to their files, and the second query, if run quickly enough afterward, before the files are replaced from cache, will add slow-cooling tokens. These keep the files in cache for a much longer time, protecting them against replacement by files staged when processing subsequent large queries.

We experimented with a 'weighted' UBC replacement policy, in which tokens added to larger files cool faster, and found that a switch to such a policy had little effect on performance. Interestingly, when applying UBC to the baseline system, we found that it performed worse than LRU, in fact it performed about as bad as random cache replacement. Conversely, LRU performs reasonably well in the cache filtering system, especially for larger cache sizes, but not as well as UBC.

6 Performance evaluation

We have made heavy use of simulation experiments in developing and evaluating our system architecture. Apart from the expense and difficulty of doing large-scale experiments with real hardware, we decided to use simulation because it is the only practical way to ensure that our design is valid over a large parameter space. This large parameter space reflects the large uncertainties about the details of data analysis in the future physics experiments that we aim to support.

Due to space limitations we cannot exhaustively discuss all results we obtained. We limit ourselves to discussing performance over the likely ranges of four important parameters which have particularly strong performance effects. These parameters are the workload type, the average number of times a query is performed on the same object set, the initial clustering efficiency, and the size of the disk farm used for staging and caching. All other parameters are fixed at single likely values. Results are in section 6.3.

We use discrete event simulation on a 200-chunk 'slice' of a large query processing system. This number of 200 is high enough so as not to bias our results towards small systems, and low enough to keep the simulation time manageable. Even so, in exploring the parameter space to investigate performance and design options, we used about 600 CPU hours running some 25000 simulations.

6.1 Simulated workloads

To drive our performance evaluation we generated multi-user workloads with the following properties. A user will pick an object set and then run a number of queries on this set. The user waits for each query to finish before submitting the next query. The average number of queries run by a user on a single object set is a workload parameter, which we vary from 2 to 32 in section 6.3.

In every workload, the sizes of the object sets picked by the users vary over a very broad range: from 0.03% to 6% of the complete dataset, with an average of 0.34%. This broad range is a very fundamental property of high energy physics workloads.

A user working on small object sets is expected to have a short think-submit-wait cycle, with most or all of the queried objects remaining in the disk cache. We expect such a user to submit more queries per day than one working on large object sets, where queries are tape bound and take hours or days to run. Our workloads reflect this expectation as follows: queries of size $n/2$ have exactly twice as much chance of occurring as queries of size n.

We bound the amount of query parallelism in the system with the constraint that all users together never have more than 10 queries submitted concurrently. On average, in the period that one user is running queries on a chosen object set, about 12 other users will pick new object sets and start running queries on them.

We consider two types of workloads: generic workloads and physics workloads.

In a generic workload, users pick object sets independently and at random. The workloads do not include any 'hot spots': all objects are equally likely to be in an object set picked by a user. The absence of hot spots is somewhat unrealistic: we expect most workload–dataset combinations to have them. We have performed simulations which included such hot-spots, and found that including them speeds up the baseline system more than the cache filtering system. We chose to report on simulations for generic workloads without hot spots in section 6.3, because we found no compact way of reporting on the exact parameters of our simulated hot spots.

In a physics workload, we have very specific dependencies between the different object sets picked by users, these dependencies reflect the user-level and intra-user dependen-

cies for analysis efforts in high energy physics. Only the largest object sets in our physics workloads are picked independently. All smaller sets are picked by taking a subset of an object set about twice as large, which was visited in a recent query (not always by the same user). These specific dependencies very much improve cache efficiency: not only are queries highly correlated, so that less space is needed to contain the object sets in many queries, but the first query on a newly picked object set will also generally find all required objects already in cache.

Number of chunks in slice: 200
Chunk size: 1 GB
Tape system size: 200 GB
Disk space for staging/caching: 4–40 GB
Object size: 100 KB
Speed of disk versus tape: factor 50
Tape robot: Storagetek Powderhorn
Tape drive(s): Storagetek Redwood SD3
Tape size: 10 GB
Tape speed for sequential reading: 11.1 MB/s
Tape change time: 19 sec
Seek to from front to end of tape: 42 sec

Table 1: Size and hardware parameters of the simulated system slice

6.2 Simulated hardware

The simulated hardware parameters are shown in table 1. The tape parameters are those of a system recently taken into production use at CERN. With our tape reading policy, which generally causes the reading of many files from a mounted tape, these parameters yield an average mount/seek overhead of about 10 seconds per file on tape. For the majority of simulated workloads, the system was tape-bound, with the disk farm only lightly loaded. We assumed an I/O capacity of 500 MB/s for the disk farm, reflecting the MB/s price ratio between disk and tape.

6.3 Performance results

We compared the performance of the baseline system, with LRU cache replacement, and the cache filtering system, with UBC cache replacement, over a range of workload parameters. Tables 2 and 3 show the speeds (in MB/s average throughput of queried objects to queries) of the baseline and cache filtering systems. Speeds are tabulated for the two workload types, different average numbers of queries a user runs on the same object set (Av qry/oset), and two initial clustering qualities expressed as the average percentage of objects hit per chunk (Av % o hit/chunk, with references to figure 3 in brackets). The disk farm size is 20 GB, or 10% of the size of the dataset on tape. The (Speedup) column shows the speedup of the cache filtering system over the baseline, the (Eq Cx) shows the factor by which the disk cache size in the baseline would need to be increased to get the same speedup.

Av qry/ oset	Av % o hit/ chunk	Base line MB/s	Cache filt. MB/s	Speed up	Eq Cx
2	30 (a)	3.9	4.9	1.3	1.9
2	13 (b)	1.9	2.5	1.3	2.5
8	30 (a)	7.4	13	1.7	1.6
8	13 (b)	2.5	7.1	2.9	3.5
32	30 (a)	10	29	2.9	1.9
32	13 (b)	2.5	15	6.0	3.9

Table 2: Performance for generic workloads

Av qry/ oset	Av % o hit/ chunk	Base line MB/s	Cache filt. MB/s	Speed up	Eq Cx
2	30 (a)	13	16	1.3	1.3
2	13 (b)	4.1	9.4	2.3	2.5
8	30 (a)	40	59	1.5	1.2
8	13 (b)	6.9	28	4.1	4.8
32	30 (a)	88	157	1.8	3.9
32	13 (b)	9.0	51	5.8	4.7

Table 3: Performance for physics workloads

The speedup factors due to our novel cache filtering optimisation range from 1.3 to 6 for the parameter combinations in the tables. Over a wider range of parameters we have seen a consistent speedup, up to a factor 6.5, for cache sizes from 4 GB to 20 GB. Above 20 GB, the baseline system is sometimes faster. In experiments with initial clustering efficiencies worse than those of the initial clustering (b), we found speedups larger than 6.

As expected, the physics workloads, with their greater locality of access between different object sets, yield a much greater speed than the generic workloads. The graphs in figure 6 provide an alternate view of how various workload parameters affect the system speeds.

7 Related work

To our knowledge, there is no other work on tertiary storage querying optimisations similar to cache filtering. Caching at a granularity below the chunk level when data moves to higher levels in the storage hierarchy occurs in many systems, see for example [5]. Our use of a true object-level granularity, coupled to optimisers which work at the level of files (object sets), seems to be new. Our previous work [7] uses techniques similar to cache filtering in a secondary storage-only reclustering system. In fact, the system developed in [7] served as a proof of concept, which gave us confidence that object-level filtering and caching could feasibly be introduced in a tertiary storage system.

Many tertiary storage data analysis systems in use in science allow users or administrators to optimise performance through the creation of new, smaller datasets which contain

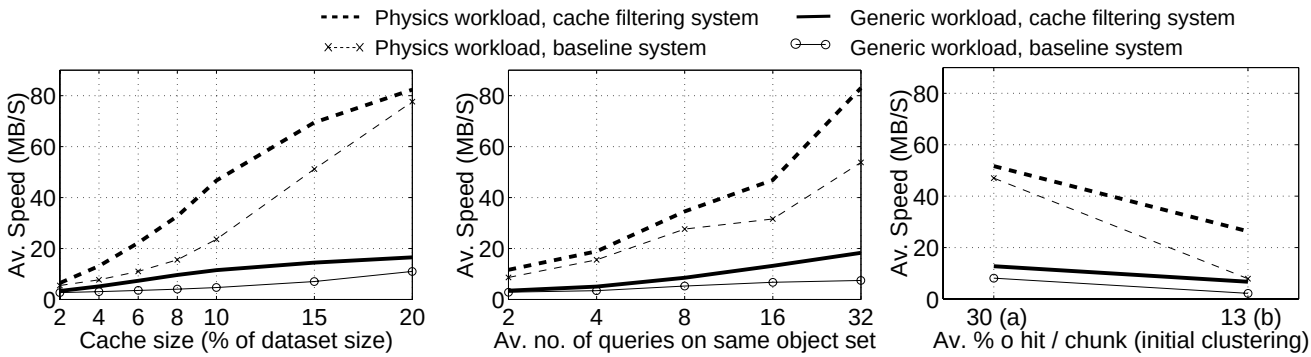

Figure 6: Dependence of different average speeds on on several parameters. Every speed value shown in a graph is the average speed over all combinations of the parameter values on the x-axes of the other two graphs.

some selected objects from the complete dataset. Queries can then be redirected to these new datasets, or are redirected automatically. Such strategies are known as 'view materialisation' or, in high energy physics, the creation of 'data summary tapes'. They are similar in intent and effect to cache filtering. However, we know of no existing tertiary storage data analysis systems in which creation of such smaller datasets, the picking of objects for them, and their eventual deletion have been automated to a large extent.

An obvious way of extending the cache reclustering system is to add a mechanism for migrating some of the small files to tape, rather than just deleting them from the disk cache. We have experimented with such an extension, but failed to find worthwhile speedups except for very small disk farms, with sizes of 4% or less than that of the data set size.

8 Conclusions

For the massive datasets used in a number of scientific endeavours, the use of tertiary storage remains the only cost-effective option in the foreseeable future. To address the needs of its future physics experiments, CERN is actively pursuing research into database technology options. We have developed a novel cache filtering optimisation, which improves cache efficiency by extracting the 'hot' objects from staged files. We also developed a 'usage based cooling' cache replacement algorithm which works well in combination with cache filtering. We have analysed the performance of both a baseline system and our cache filtering system over a large parameter space, identifying four parameters with a particularly large influence on system performance. We found that cache filtering produces speedup factors up to 6. Cache filtering speeds up a system especially if the initial clustering efficiency is low. The results obtained here will serve as a basis for future R&D at CERN.

References

[1] L. Bernardo, H. Nordberg, D. Rotem, A. Shoshani, Determining the Optimal File Size on Tertiary Storage Systems Based on the Distribution of Query Sizes. Proc. 10th Int. Conf. on Scientific and Statistical Database Management, Capri, Italy, (1998).

[2] L. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, A. Shoshani. Efficient Organisation and Access of Multi-Dimensional Datasets on Tertiary Storage Systems. Information-Systems, vol.20, no.2, p.155–183, (1995).

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

[4] J. Gray, G. Graefe, The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb. SIGMOD Record 26(4): 63–68 (1997).

[5] R. Grossman, D. Hanley, X. Qin, Caching and Migration for Multilevel Persistent Object Stores. Proc. of 14th IEEE Symposium on Mass Storage Systems 127-135 (1995).

[6] A. Hanushevsky, M. Nowak. Pursuit of a Scalable High Performance Multi-Petabyte Database. 16th IEEE Symposium on Mass Storage Systems, San Diego, California, USA, (1999).

[7] K. Holtman, P. van der Stok, I. Willers. Automatic Reclustering of Objects in Very Large Databases for High Energy Physics, Proc. of IDEAS '98, Cardiff, UK, p. 132-140, IEEE (1998).

[8] L. Lueking, Managing and Serving a Multiterabyte Data Set at the Fermilab DØ Experiment. Proc. of 14th IEEE Symposium on Mass Storage Systems (1995).

[9] The RD45 project (A Persistent Storage Manager for HEP). http://wwwinfo.cern.ch/asd/rd45/

[10] S. Sarawagi, M. Stonebraker, Efficient organization of large multidimensional arrays. Proc. of 10th Int. Conf. on Data Engineering, p. 328–336 (1994).

[11] J. Yu, D. DeWitt, Query pre-execution and batching in Paradise: A two-pronged approach to the efficient processing of queries in tape-resident data sets. Proc. of 9th Int. Conf. on Scientific and Statistical Database Management, Olympia, Washington (1997).