# Coordinating Simultaneous Caching of File Bundles from Tertiary Storage

A. Shoshani, A. Sim, L. M. Bernardo, H. Nordberg,
*National Energy Research Scientific Computing Division*
*Lawrence Berkeley National Laboratory*
*(Shoshani, Asim, LMBernardo, Hnordberg)@lbl.gov*

## Abstract

In a previous paper [Shoshani et al 99], we described a system called STACS (Storage Access Coordination System) for High Energy and Physics (HEP) experiments. These experiments generate very large volumes of "event" data at a very high rate. The volumes of data may reach 100's of terabytes/year and therefore they are stored on robotic tape systems that are managed by a mass storage system. The data are stored as files on tapes according to a predetermined order, usually according to the order they are generated. A major bottleneck is the retrieval of subsets of these large datasets during the analysis phase. STACS is designed to optimize the use of a disk cache, and thus minimize the number of files read from tape. In this paper, we describe an interesting problem of disk staging coordination that goes beyond the one-file-at-a-time requirement. The problem stems from the need to coordinate the *simultaneous* caching of groups of files that we refer to as "bundles of files". All files from a bundle need to be *at the same time* in the disk cache in order for the analysis application to proceed. This is a radically different problem from the case where the analysis applications need only one file at a time. In this paper, we describe the method of identifying the file bundles, and the scheduling of bundle caching in such a way that files shared between bundles are not removed from the cache unnecessarily. We describe the methodology and the policies used to determine the order of caching bundles of files, and the order of removing files from the cache when space is needed.

## 1. Introduction

Because of advances in computer technology and automated sensor system, it is possible today to set up experiments and simulations that generate tremendous amounts of data. It is common to hear of systems that collect or generate terabytes ($10^{12}$ bytes) or even petabytes ($10^{15}$ bytes) of data. While disk systems are becoming cheaper, storing terabytes/petabytes of data on disk continues to be economically prohibitive. Therefore, most of the data from such experiments/simulations are stored on robotic tape systems. This presents a challenge of how to organize and manage the data stored on tape systems.

One scientific application area that generates large amounts of data is High Energy Physics (HEP). Over the last three years, we have developed a system for coordinating the access to tertiary storage for multiple HEP clients wishing to access data from files stored on tape. The system, called STACS (STorage Access Coordination System) essentially performs 4 functions: (i) it determines, for each query request, which files need to be accessed; (ii) it determines the order of files to be cached dynamically so as to maximize their sharing by queries; (iii) it requests the caching of files from the robotic tape system in tape optimized order; and (iv) it determines dynamically which files to keep in the disk cache to maximize file usage. This system was described in [Shoshani et al 99]). However, the previous paper deals with application clients that require only a single file at a time to proceed with the analysis. For reasons explained below, it is often the case that multiple files are needed simultaneously for the analysis. This led to the work described in this paper.

To explain the problem we address in this paper, we describe briefly the type of data generated by the HEP experiments. In HEP experiments, elementary particles are accelerated to nearly the speed of light and made to collide. These collisions generate a large number of additional particles. For each collision, called an "event", about 1-10 MBs of raw data are collected. The rate of these collisions is 1-10 per second, corresponding to 30-300 million events per year. Thus, the total amount of raw data collected in a

year amount to 100s of terabytes to a few petabytes. After the raw data are collected they undergoes a "reconstruction" phase, where each event is analyzed to determine the particles it produced and to extract its summary properties (such as the total energy of the event, momentum, and number of particles of each type). The volume of data generated after the reconstruction phase ranges from a tenth of the raw data to about the same volume. Most of the time only the reconstructed data are needed for analysis, but the raw data must still be available.

Event data can be thought of as "chunks of data" which are organized into files, normally about 1 GB each. The reason for this size is that the mass storage system (which in our case is called HPSS[♣] – the High Performance Storage System developed by IBM) operates more efficiently with large files. However, we do not wish to make files too big, since files cached may contain too much unneeded data. Typically, there may be 100-500 events in a file. When a file is cached for an application only part of the events may be relevant to it. For example, if the application needs events with energy>5 GEV, then only a fraction of events in each file may qualify.

STACS was originally designed assuming that a HEP analysis client can find all the data it needs about a particular event in a single file. However, because of the size of the data, it was decided to partition the reconstructed data into "component" types. Each event has several components associated with it. In HEP these components correspond to different aspects of the physics, such as the trajectories of each of the 1000s of particles produced in the collision (called "tracks"), and the positions in space where particles split (called "vertices"). Since not all the components are needed at the same time for analysis, and the volume of the data is so large, the data is partitioned into these component types and stored separately. For analysis, the Client (user) can specify that any combination of these components can be requested *concurrently*, i.e. the files containing these components need to be in the cache at the same time to perform the analysis. This is the problem addressed in this paper.

We note that this kind of data are not unique to the HEP application area. For example, earth observation satellites [NASA web] have multiple sensors, each generating a series of images/data-chunks. For each point in time-space, there is a collection of data-chunks, some of which maybe needed at the same time for analysis. Another example is the series of images collected for gene expression data. Each image is analyzed, and several data objects can be associated with each image. Here again, several images and associated data objects may be needed at the same time depending on the nature of the analysis performed on such data.

In section 2, we introduce the concept of a file bundle, and the practical reasons for that. Section 3 describes what extensions were made to the STACS index to support file bundles. In section 4, we describe the methodology and techniques used to achieve file bundle caching coordination. In section 5, the implementation of the STACS system is described, as well as the method for monitoring the system operation. We conclude in section 6 with a summary and a discussion of future work.

## 2. File Bundles

As explained above, each event Ei has several component data-chunks associated with it: C1(Ei), C2(Ei), …, Cn(Ei). While the system is designed for any number of components n, it is typically less than 10. In general, a component Cj(Ei) can reside in any file, Fk; i.e. a given file can have a mixture of component types. However, in HEP applications, it makes sense to organize the components into files according to types, since a request by the application for certain component types would not involve caching of files with other component types. Thus, all components of type Cj (for all events Ei) go into their own series of files, i.e. every file contains only components of a certain type. This is illustrated in Figure 1.

As can be seen, if we look at each combination of files containing components of type A and files containing components of type B, they have components of certain events in common. For example, file1 and file2 in Figure 1 have components of the events {E1, E2, E3, E5) in common. We call such a file combination a *file bundle*. If, for example, an application needs component A and Component B for events E3, E5, E7, the system will bring to cache two file bundles: the file bundle (F1,F2) for events E3 and E5, and the file bundle (F3,F2) for event E7.

[♣] http://www.sdsc.edu/hpss/

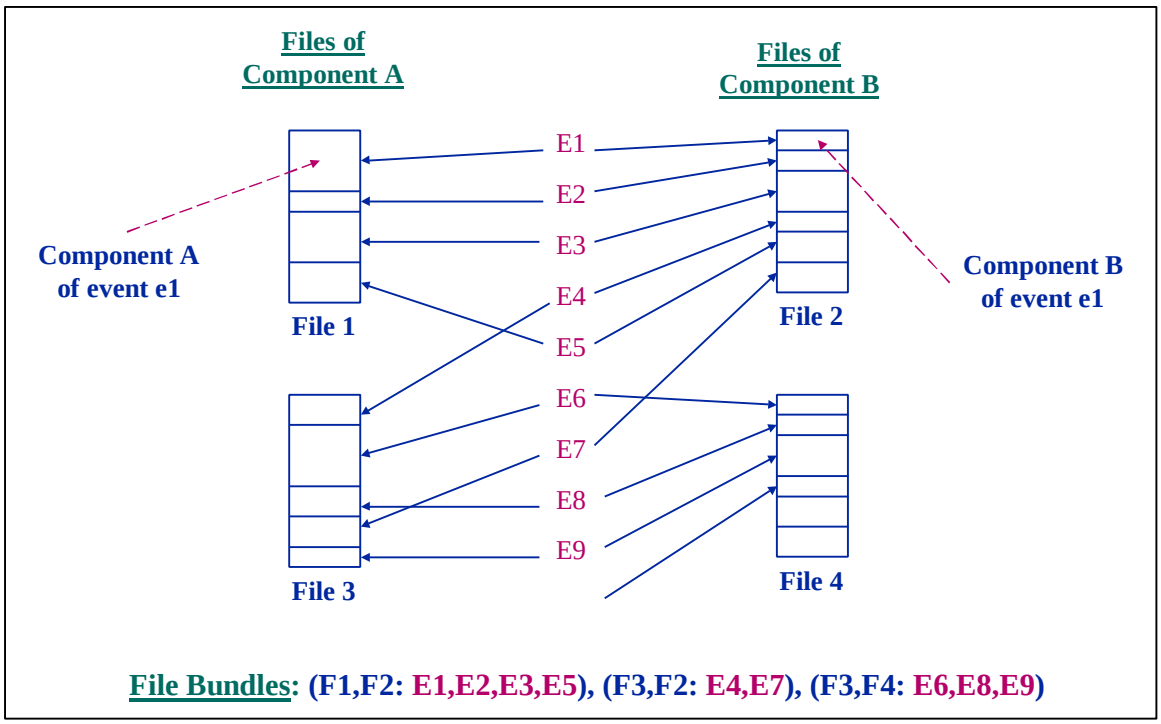


Figure 1: Components of events and file bundles.

We note that not all events may be needed out of a particular file bundle. In the above case only E3,E5 are needed out of the file bundle (F1,F2), and only E7 is needed out of (F3,F2). Therefore, a file bundle can be used by several clients, each accessing another subset of the event components.

File bundles can have *overlapping* files. In the above example F2 is overlapping. The importance of overlapping files is in managing the cache. When a given file bundle is brought into the disk cache, the system has to give a priority of keeping an overlapping file in cache till the other file(s) of other overlapping bundles are brought into cache. This is one aspect of the problem we address in the next section. We note that a file bundle is the *minimal* set of files that must be in cache for the Client (analysis code) to proceed. It is not enough to find all the files that are necessary for the query, since we may not have the space in cache to bring all the files at once for the query. Thus, we need to know the minimal file set (or the bundle), and if space permits, or a priority policy allows, more bundles can be brought in. We refer to bringing more than one file bundle at a time to cache as a "pre-fetching" policy in section 4. If the system policy permits more files to be cached per user, it is best to cache bundles that have files in common. This is also addressed in the coordination algorithms described in section 4.

Another aspect of the problem is how to determine which files to bring into cache when a new bundle is needed by the Client. The goal is to maximize file sharing between the clients. It is also necessary to avoid *deadlocks*, where 2 or more bundles are partially in cache, each waiting for the other(s) to release files.

As mentioned previously, we should not, in general, make the assumption that event components are always grouped into files according to their component types. For example, suppose that events are partitioned into components $C_A$, $C_B$, and $C_C$. We may have the components for event E3, $C_A$(E3), $C_B$(E3), and $C_C$(E3), all residing in the same file Fk. In this case the file bundle for E3 is made of the single file Fk. This implies that for a particular request, the file bundles can be of different cardinalities (but, they cannot exceed the number of components requested). Our solution makes no

assumptions on the number of files in the bundles, or whether components of different types are stored in different files. The case of files that contain only components of a certain type is, therefore, a special case that is handled as well.

## 3. Finding the Bundles

To select which files are relevant to a given query, we need an index that maps from the predicate conditions on properties specified in the query to the set of files needed to satisfy the query. Because of the large number of index entries ($10^8$-$10^9$) and the high dimensionality of the property space, we developed a specialized bit-sliced index [Shoshani et al 99]. In the initial version, the index was designed to support queries that do not specify components; that is, all the data for an event data are assumed to be in a single file. For example, the following query specifies range predicates for a dataset called star_dataset:

SELECT *
FROM star_dataset
WHERE 500<total_tracks<1000 & energy<3

The bit-sliced index associates a fixed vector position with each event. Given a query, it returns a (compressed) bit-slice, where a 1 indicates that an event qualified in this position and a 0 indicates that it does not qualify. This bit-slice is then applied to a file vector containing the corresponding file IDs. The result returns the list of files that qualifies for this query. By removing the file duplicates (using a hash map), the index generates the set of files that qualify for this query, and for each file the subset of events that are relevant to this query. An example of the result is shown below, where inside the [] we specify the file and the events in it that qualify for the query, and {} denotes a set of files.

{[F7: E4,E17,E44], [F13: E6,E8,E32], …}

Note that it is not possible to have file overlap in this case, since if this exists the event sets will be put together into a single list associated with this file.

Now, this index had to be extended to handle component specification. Suppose that events were partitioned into components called “raw”, “tracks”, “hits” and “vertices”. An example of a query is:

SELECT vertices, raw
FROM star_dataset
WHERE 500<total_tracks<1000 & energy<3

where only the components "vertices" and "raw" are requested. An example of a result is shown in the following example, where inside the [] we specify the file bundle and the events in it that qualify for the query, and {} denotes a set of file bundles.

{[F7, F16: E4,E17,E44], [F13, F16: E6,E8,E32], …}

Note that in this case file overlap between file bundles is possible.

The extension to the bit-sliced index to handle bundles was quite simple. We only had to have a separate file vector for each component. First the predicate is run against the index to generate a bit-slice as before. Then for each position in the bit-slice with a 1 in it, we pick the file IDs for the corresponding components. This forms a file bundle. We then use a hash map over the bundle IDs to remove duplicates. To support the case that different component types reside in the same file, we simply remove duplicates in the set of files of each bundle. For example, if the query has 3 components, and components A and B of some bundle reside in the same file Fj, while the component C resides in file Fk, the bundle found by the index will be (Fj, Fj, Fk). Removing the duplicate file generates a bundle with only 2 files (Fj, Fk).

Now, we turn to the main scheduling issue.

## 4. Bundle Caching Coordination

### 4.1 File weights and bundle weights

The key to scheduling the caching of file bundles, and determining what should be in cache at any one time, is the assignment of weights to files and to bundles dynamically. Various policies are possible for assigning the weights, such as the number of event components in a file, the size of a file, etc. We chose a simple and intuitive policy, where the file weight is proportional to the number of bundles that the file participates in, summarized over all the queries in the system.

Accordingly, the most "popular" files will have a higher weight. Intuitively, caching first the files with higher weights will satisfy the largest number of queries.

Specifically, the file weight is *incremented* by 1 if it appears in a particular bundle of a query. Thus, if a file appears in 3 bundles for that query, it is incremented 3 times. The Initial File Weight, IFW, assigned to a file *Fk* is obtained by summing the weight *Wij(Fk)* over all bundles *j* of a query and over all queries *i*, as shown below:

$$IFW(F_k) = \sum_{i}^{\substack{all\\queries}} \sum_{j}^{\substack{all\ bundles\\in\ queries}} W_{ij}(F_k) = \begin{cases} 1 & if\ F_k\ is\ in\ bundle\ j \\ 0 & otherwise \end{cases}$$

For example, if there are 2 queries in the system, and file Fk appears in 5 bundles for query 1 and 3 bundles in for query 2, then IFW(Fk) = 8.

To account for the dynamic case, we need to adjust the weight for every bundle that was processed by some Client. Specifically, after a bundle is processed by a Client, and the Client releases it, the system decrements the file weight by 1 for each file in the bundle that was released. Thus the Dynamic File Bundle, DFW, is:

$$DFW(F_k) = IFW(F_k) - \sum_{i}^{\substack{all\\queries}} \sum_{j}^{\substack{processed\\bundles\\in\ queries}} W_{ij}(F_k) = \begin{cases} 1 & if\ F_k\ is\ in\ bundle\ j \\ 0 & otherwise \end{cases}$$

Similarly, the DFW of files are dynamically incremented for each new query request that arrives to STACS.

The Dynamic Bundle Weight, DBW, is simply the sum of the weights of all the files in the bundle.

$$DBW(Bi) = \sum_{k}^{\substack{all\ files\\in\ bundle\ i}} DFW(F_k)$$

## 4.2 Bundle caching policy

Queries are serviced in turn according to some system policy. The order of servicing queries can vary according to administrative goals, such as Round Robin (RR) to be fair to all users, or Shortest Query First (SQF) to shorten turn around time for short queries. Care also needs to be provided that queries are not starved perpetually. In principle, query service policies can be tuned to types of users or types of queries based on priority assignment.

We allow queries to request multiple bundles in a controlled fashion. Assuming that each Client uses a single processor, it can only process one bundle at a time. However, we may still want to pre-fetch a bundle, so that the bundle is already in the disk cache when the Client has finished processing the previous bundle. This is, again, an administrative choice depending on how loaded the system is and priorities assigned to users. To accommodate parallel processing, one can permit multiple requests per user to be pre-fetched.

Currently, we use the RR policy. Service for a query is skipped if the query has all the bundles it requested satisfied (subject to pre-fetching limits) and it is still processing them.

When it is a query's turn to be serviced, the system needs to determine which bundle to cache. We use the bundle weights for this purpose. It is possible that some bundles of the query may have one or more files already in cache because another query is currently using or previously used those files.

The policy used is to cache the file bundle with the most files in cache. In case of a tie, the bundle with the highest weight is selected. In case that there is no space in cache for the selected bundle, the next eligible bundle that will fit the cache is selected.

In addition, as soon as any file is cached, the system checks if there are any bundles in cache that can satisfy pending queries as a result of this file being cached. If such a query is found, the bundle is passed to it, even if this is out of the RR order.

To avoid deadlocks, we follow a procedure similar to 2-phase-commit. First, we verify that the space is available on disk cache. If it is not available, we request to remove (purge) one or more files from disk to make space for all the needed files in the bundle (i.e. not currently in cache). When space is allocated, it is "locked out" (committed). No other query can use this space.

### 4.3 File purging policy

File purging policy is the policy that determines what to release from cache when cache space is needed. No file purging occurs until space is needed by some query.

Unlike the file caching policy that is based on bundle weights, the file purging policy is based on file weights only. This is to ensure that if a file is needed by more than one bundle, its purging is deferred as long as possible.

The current policy is simply to purge the file with the smallest weight. In case of a tie, the largest file will be purged to make space for other large files (an alternative policy for ties is to choose the file that has been in cache unused for the longest time).

### 4.4 Discussion

The technique of assigning file weights and bundle weights permitted us to use different criteria for the caching of bundles and purging of files. File caching for bundles is handled on a bundle weight basis, ensuring that the most worthy bundle (the bundle that has the most files shared by other bundles) is cached first. This way we can bring about file sharing whenever possible. On the other hand, file purging occurs on a file weight basis, keeping files in cache longer if they are shared between bundles and are still needed by active queries.

One can consider other file weight policies or combinations of policies, such as the total size of a bundle (in MBs), or whether files reside on the same tape to minimize latency. We chose to maximize file sharing between queries first. At a later stage, before the files are actually cached, we look at the queue of the files that were requested to be cached, and reorder them according to tape clustering, as described in [Bernardo et al 2000].

The techniques and policies described above have been implemented and incorporated into STACS. We describe briefly the system architecture below, and which components were affected. We follow that with an example of a log that shows the coordination of bundle caching in test operation.

## 5. STACS implementation and operation

### 5.1 System Architecture

As shown in Figure 2, STACS has 3 main components that represent its 3 functions: 1) The Query Estimator (QE) uses the index to determine which file bundles and which events are needed to satisfy a given range query. 2) The Query Monitor (QM) keeps track of the queries that are executing at any time, which file bundles are cached on behalf of each query, which files are not in use but are still in cache, and which file bundles still need to be cached. The Query Monitor consults an additional module, called the Caching Policy module, which determines which bundle should be processed next according to the policies of the system. 3) The Cache Manager is responsible for interfacing to the mass storage system (HPSS), issuing PFTP (parallel FTP) requests, and purging files from the disk cache when space is needed.

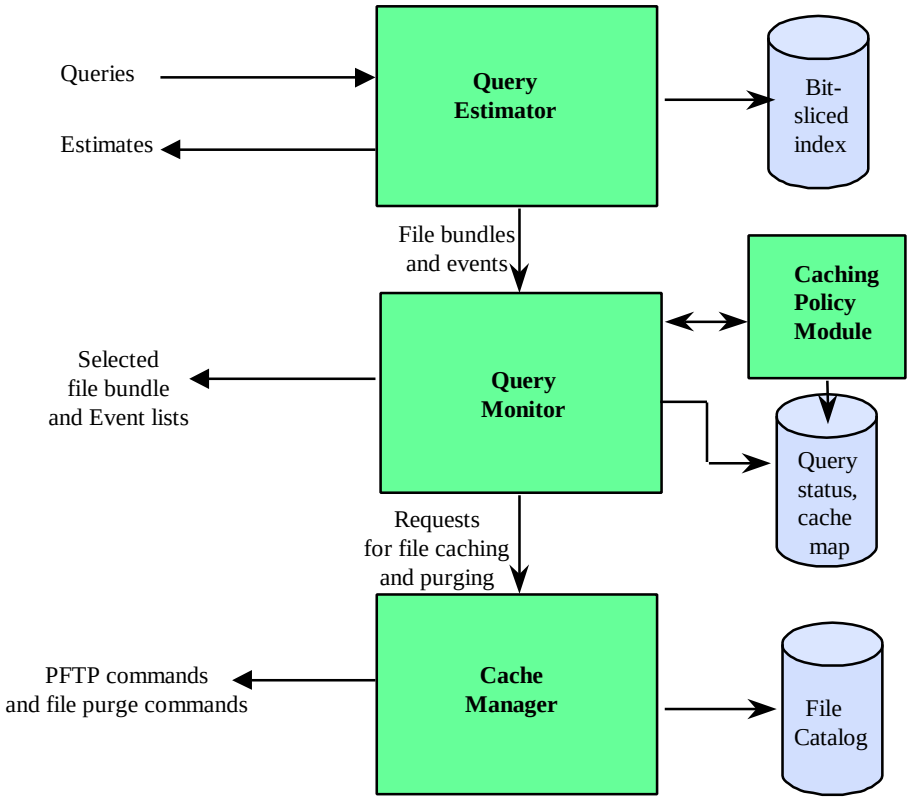


Figure 2. The Storage Access Coordination System (STACS)

The Cache Manager controls the rate of PFTP (parallel file transfers) submission to HPSS, so as not to flood it. The number of active PFTPs is set as a parameter that can be changed dynamically by the system administrator. To perform this function, the Cache Manager maintains a queue of file caching requests. It consults a file catalog in order to know where to move the cached file (i.e. which directory on disk). It also monitors the performance of each PFTP, checking for error messages, and rescheduling caching requests that failed. The details of the functions performed by this component and its implementation are described in [Bernardo et al 2000].

The communication between the STACS components and the Client modules are made through CORBA interfaces. The Client modules communicate with STACS by issuing query requests, asking for estimates of the numbers of events and files involved, and the time to execute the query. After issuing an execute request, the Client can get information about the remaining time and files that need to be cached to complete the query. All the communication to the STACS modules must support multi-threading, since we need to support multiple queries and multiple bundle and file processing concurrently. Our experience with using multi-threaded CORBA ORBs is described in [Sim et al 99].

### 5.2 The Query Monitor operation

In addition to the changes made to the Query Estimator and the bit-sliced index to handle bundles, most of the changes required to support file bundles were made to the Query Monitor and the Caching Policy module.

The Query Monitor maintains the query queue, and bundle and file status as shown in Figure 3.

The query queue is used for the round robin service of the queries. If a query is busy processing, it is skipped. Each query points to its set of bundles. Bundles are marked whether they have been processed, whether they are currently in process, or whether they are waiting to be processed. Each bundle has a bundle

weight associated with it, that changes dynamically as bundles are processed, and as new queries enter the system. The bundle weights are used to determine which bundle to process next for the query being serviced.

Each bundle points to a set of files that it contains. As explained above, the file weights are used to determine which file(s) to remove from the disk cache if space is needed. The system also maintains information on the set of files currently being processed, and therefore cannot be considered for removal from cache. The set of files in cache which are not currently in use are candidates for removal from cache. As mentioned above the current policy used when space is needed is to remove a file not currently in use that has the lowest weight. Another reasonable policy is to remove the "least recently used" file (the file that has not been read the longest time), but we found it easier to use the weight as a guide.

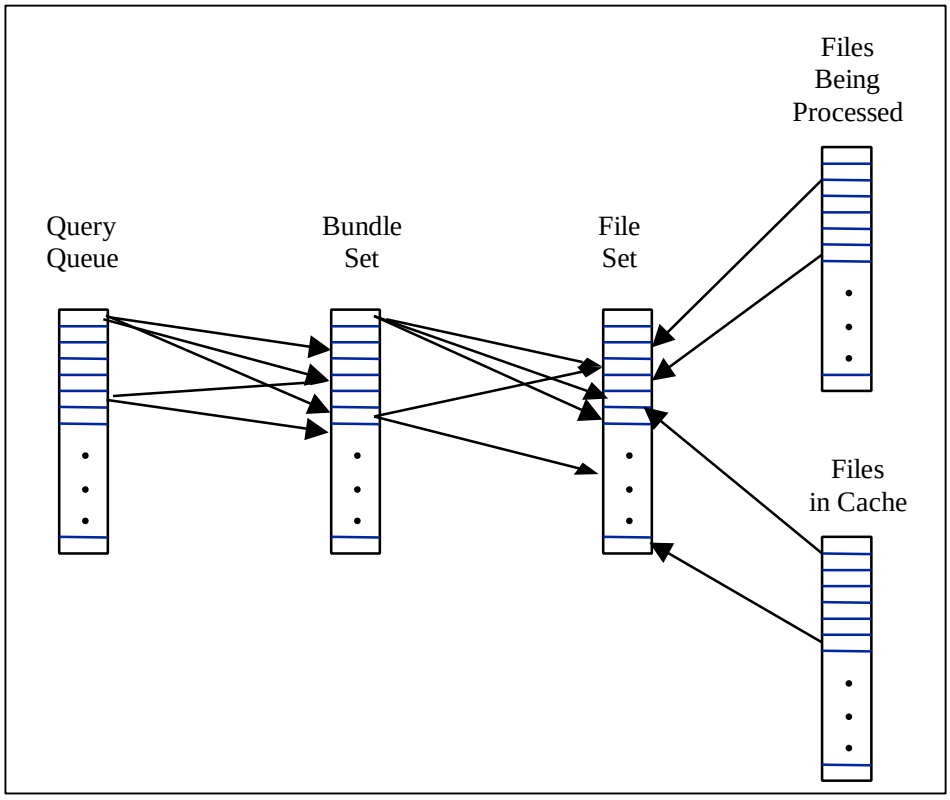


Figure 3. Queues and lists maintained by the Query Monitor

A typical sequence of events that the Query Monitor (QM) goes through for each query is shown schematically in Figure 4. The Client initiates a query by communicating with the QE (not shown in the figure below). The QE determines the set of bundles and events in each bundle that qualify for the query and issues an "execute" command to the QM. The QE then notifies the Client that it can start requesting bundles.

After an "execute" command is issued (1), the Client notifies the QM that it is ready to receive a bundle (2). The QM consults the Policy Module to determine which bundle to process next (3). Depending on the result (4), it may need to request files that are not in cache from the Cache Manager (5). When all the requested files are staged (6), it notifies the Client that it can begin processing (7). After a bundle is released by the Client (8), the QM marks it as a candidate for removal if no other query is processing this file. If space is needed, the QM requests that the selected file(s) be removed (9). When this is done (10) the QM can continue with its scheduling of bundles to be processed. This process repeats until all the bundles of a query are processed, at which time the query is removed from the query queue, and its associated bundles removed from the bundle set.

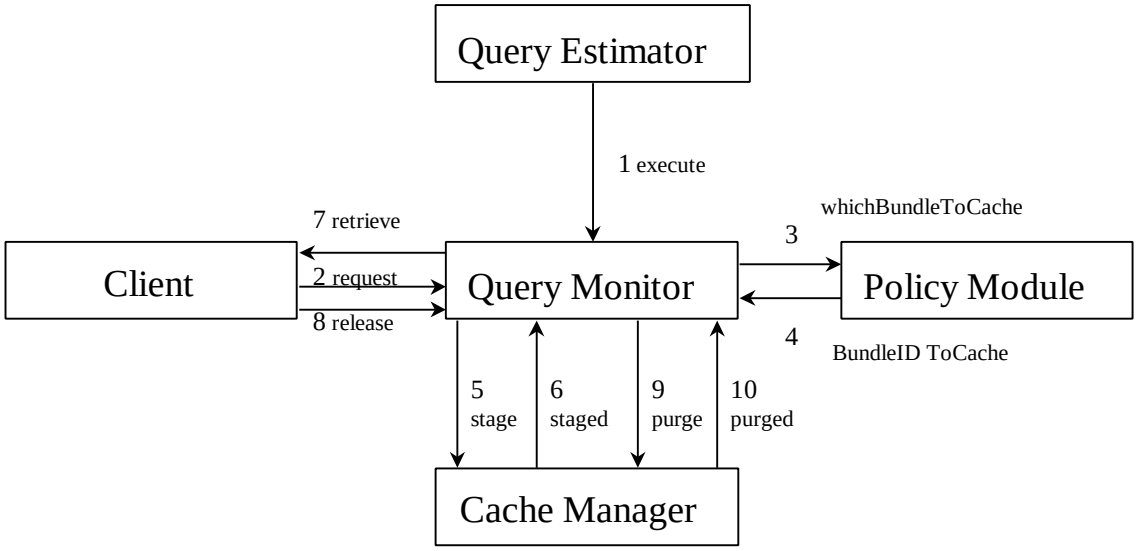


Figure 4. The sequence of operations that follow an execute command

Finally, we note that the design of the Query Monitor and the bit-sliced index make no assumptions on the number of files per bundle, or whether the event components are stored in a separate or the same file. Therefore, the system can be used for any simultaneous file caching requirements, for HEP data or other applications.

**5.3 Monitoring the dynamic system operation**

The graph in Figure 5 was drawn from the actual logging of a test run. The method of displaying the runs is based on a visualization tool developed at LBNL (called NetLogger [Netlogger Web]) that was applied to show the progression of actions over time. The graph represents the occurrence of logged actions (stretched out in the y-axis) over time (the x-axis). There are five logged actions shown from bottom to top: a) the file_request arrived (to HPSS), b) file_staged from HPSS-tape to HPSS-cache, c) File_in_cache (i.e. all the files of a bundle were moved over the network to the Client's local disk), d) file_retrieved by the Client, and e) file_released by the Client. Thus, a vertically connected (crooked) line represents the history of a single file from the time of request to the time of its release by the Client component.

Figure 5 illustrates the coordination of file caching according to bundles. In this test, we ran 2 queries, each with 3 components, 15 minutes apart. The number of bundles required for each query is large, but the figure shows only a short time window where only a few bundles per query are shown. Each bundle has 3 files corresponding to the 3 components requested in the query. As can be seen, only after the 3 files of a bundle were cached, was the bundle passed on to the Client. Further examination of the graph shows that files were shared by the 2 queries when they were in common in these queries' bundles, and that files were left in cache when they were in common with other bundles.

We have used this method of graphing the dynamic behavior of the system to verify that it performed correctly. We can tell in this graph if a file was brought in from the robotic tape or passed to the application directly from cache. For example, the third bundle shown in the graph from left to right belongs to the second query. As can be seen the line starts from "file_in_cache", which indicates that all the files for this bundle were in cache because they overlapped the first query.

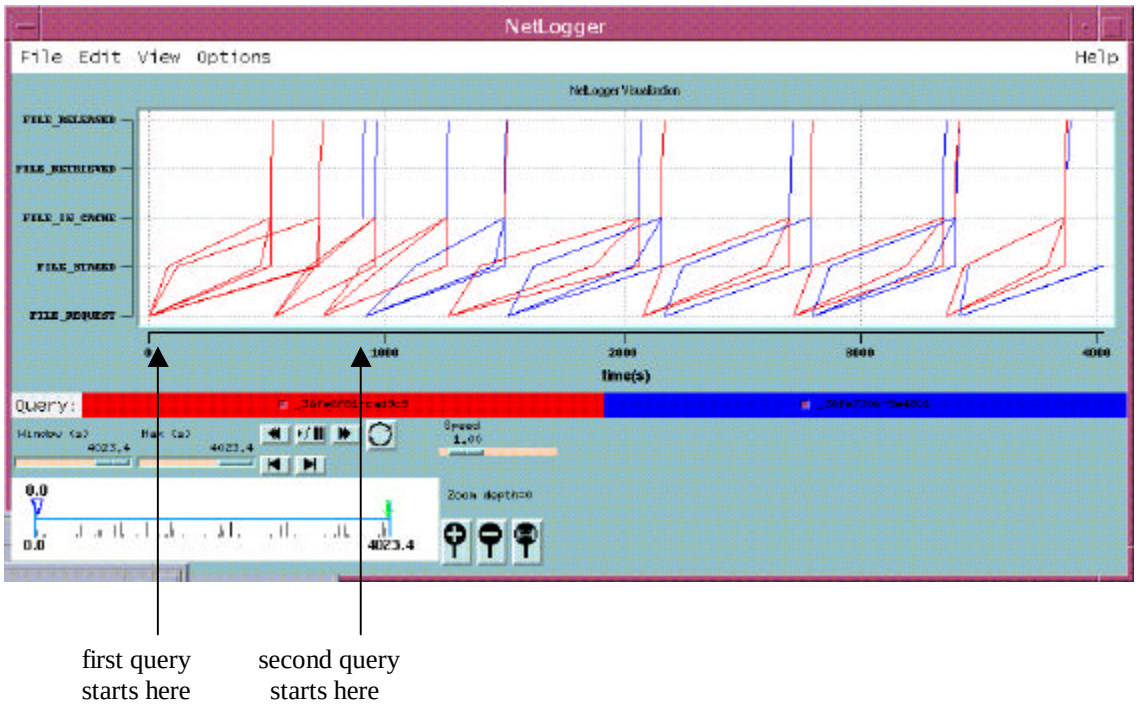


Figure 5. Monitoring the caching of file bundles

## 6. Summary and future work

The technique of assigning file weights and bundle weights permitted us to use different criteria for caching of bundles and purging of files. File caching for bundles is handled on a bundle basis, ensuring that the bundle that has the most files shared by other bundles is cached first. This way we can bring about file sharing whenever possible. File purging occurs on a file weight basis, keeping in cache as long as possible files that are shared between bundles and are still needed by active queries.

As mentioned several times in this paper, one can consider other file weight policies. We chose to give files weights that represent file sharing. Using these weights tends to maximize file sharing between queries. At a later stage, we look at the queue of the files that were requested, and reorder the caching of files according to tape clustering.

Proving which policy works best is a very difficult task. One can try different policies on the real system, but for such tests one has to have full control of the mass storage system and the content of the cache. Such systems (e.g. HPSS) are usually shared by other users, so any measurements will be effected by the load of the system at the time it is measured. Still, it is important to compare how the weight policy we use performs relative to simple policies such as "least recently used" for removing files from the cache.

Another possibility is to model the behavior with a system of queues, and assume certain distributions to query arrivals, and bundle requirements per query. One can then simulate the behavior of the system under various policies. We plan to attempt this form of analysis in the future, first for queries with a single component (i.e. only one file per bundle), and then to queries with more than one file per bundle.

**References**


[Shoshani et al 99] Multidimensional Indexing and Query Coordination for Tertiary Storage Management, A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim, Eleventh International Conference on Scientific and Statistical Database Management (SSDBM'99). (This paper can be downloaded from http://gizmo.lbl.gov/~arie/papers/download.papers.html).

[Bernardo et al 2000] Access Coordination of Tertiary Storage for High Energy Physics Application, L. M. Bernardo, A. Shoshani, A. Sim, H. Nordberg, Eight IEEE Symposium on Mass Storage Systems (MSS 2000). (This paper can be downloaded from http://gizmo.lbl.gov/~arie/papers/download.papers.html).

[NASA Web] Destination Earth, http://www.earth.nasa.gov

[Netlogger Web] NetLogger: A Methodology for Monitoring and Analysis of Distributed Systems, http://www-itg.lbl.gov/DPSS/logging/.

[Sim et al 99] Storage Access Coordination Using CORBA, A. Sim, H. Nordberg, L. M. Bernardo, A.Shoshani, D. Rotem, 1999 International Symposium on Distributed Objects and Applications (DOA'99). (This paper can be downloaded from http://gizmo.lbl.gov/~arie/papers/download.papers.html).

[STAR Web] The STAR Collaboration, http://www.rhic.bnl.gov/STAR/ . See also, STAR Computing Software, http://www.rhic.bnl.gov/STAR/html/star_computing.html