# Tertiary Storage Management
# For HENP Analysis


## Summary

In many scientific domains, experimental devices or simulation programs generate large volumes of data. Examples are atmospheric data transmitted by satellites, climate modeling simulations, combustion modeling and high energy physics experiments. The volumes of data may reach hundreds of terabytes and therefore it is impractical to store them on disk systems. Rather they are stored on robotic tape systems that are managed by some mass storage system (MSS). A major bottleneck in analyzing the simulated/collected data is the retrieval of subsets from the tertiary storage system. This bottleneck results from the fact that the requested subsets are spread over many tape volumes, because the data are stored as files on tapes according to a predetermined order, usually according to the order they are generated. In this project, we focus on a specific application area, a high energy physics data management and analysis environment, that was recently developed and is being incorporated in an operational system, scheduled to go on-line in the summer of 1999 [See STAR Computing]. We developed the architecture and implementation of a Storage Manager designed to optimize the use of a disk cache, and thus minimize the number of files read from tape. We achieve this by taking advantage of information in queries to determine the desired subset of the dataset, and by using a specialized index to locate the relevant data on tapes. The details and structure of the Storage Manager components are described in section 3 below. The results of various tests that demonstrate the benefits and efficiency gained of using the Storage Manager are described in the document Summary of Mock Data Challenge 1 Test Results.


## 1. Introduction

High energy and nuclear physics (HENP) experiments generate large volumes of data that need to be stored on tertiary storage (typically robotic tape systems) because it is prohibitively expensive to store them on disk storage. One of the major bottlenecks in analyzing the simulated/collected data is the retrieval of subsets from the tertiary storage system. This bottleneck results from the fact that the requested subsets are spread over many tape volumes, because the data are stored as files on tapes according to a predetermined order, usually according to the order they are generated. We have developed a Storage Manager designed to optimize the use of a disk cache, and thus minimize the number of files read from tape. Although we focus in this work on HENP applications, similar principles of the architecture and the use of indexes apply to other areas.

To explain the detail of the system we developed, it is necessary to provide some background on this application area. HENP experiments consist of accelerating sub-atomic particles to nearly the speed of light and forcing their collision. A small part of the particles collide and produce a large number of sub-particles. Each such collision (called an "event") generates in the order of 1-10 MBs of raw data collected by a detector. The rate of data collection for the HENP experiment we are addressing (called the STAR collaboration) is a few such event collisions per second, or about 10 MB/s on the average. This corresponds to $10^7$-$10^8$ events/year, and the total data volume amounts to about 300 TBs per year. This corresponds to 10,000 30 GBs tapes, which is the reason for the use of a robotic tape system. A typical experiment may run for 3 years.

After the raw data are collected, they undergo a "reconstruction phase". Each event is analyzed to determine its sub-particles and to extract summary properties for each event (such as the total energy of the event, momentum and number of particles of each type). The amount of data generated after the reconstruction phase ranges from a tenth of the raw data to double that, which amounts to about 30-600 TBs per year. Most of the time only the reconstructed data are needed for analysis, but the raw data must still be available.

Events 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 too much unneeded data may be read. In a recent paper we have analyzed the optimal file size, and determined that a 1GB size is a reasonable compromise. We note that for our purposes, we use the term "event" as equivalent to the "chunk of data" stored for that event.

The number of summary elements extracted per event is quite large (100-200). Figure 1 shows a real example of data values for one such event for 54 properties.

| | | |
|---|---|---|
| I N(1) 9965 | I Np(3) 24 | R AVpT(1) 0.325951 |
| I N(2) 1192 | I Npbar(1) 94 | R AVpT(2) 0.402098 |
| I N(3) 1704 | I Npbar(2) 12 | R AVpTpip(1) 0.300771 |
| I Npip(1) 2443 | I Npbar(3) 24 | R AVpTpip(2) 0.379093 |
| I Npip(2) 551 | I NSEC(1) 15607 | R AVpTpim(1) 0.298997 |
| I Npip(3) 426 | I NSEC(2) 1342 | R AVpTpim(2) 0.375859 |
| I Npim(1) 2480 | I NSECpip(1) 638 | R AVpTkp(1) 0.421875 |
| I Npim(2) 541 | I NSECpip(2) 191 | R AVpTkp(2) 0.564385 |
| I Npim(3) 382 | I NSECpim(1) 728 | R AVpTkm(1) 0.435554 |
| I Nkp(1) 229 | I NSECpim(2) 206 | R AVpTkm(2) 0.663398 |
| I Nkp(2) 30 | I NSECkp(1) 3 | R AVpTp(1) 0.651253 |
| I Nkp(3) 50 | I NSECkp(2) 0 | R AVpTp(2) 0.777526 |
| I Nkm(1) 209 | I NSECkm(1) 0 | R AVpTpbar(1) 0.399824 |
| I Nkm(2) 23 | I NSECkm(2) 0 | R AVpTpbar(2) 0.690237 |
| I Nkm(3) 32 | I NSECp(1) 524 | I NHIGHpT(1) 205 |
| I Np(1) 255 | I NSECp(2) 244 | I NHIGHpT(2) 7 |
| I Np(2) 34 | I NSECpbar(1) 41 | I NHIGHpT(3) 1 |
| | I NSECpbar(2) 8 | I NHIGHpT(4) 0 |
| | | I NHIGHpT(5) 0 |

Figure 1: 54 Properties for one event. (There are $10^8$ events per year.)

Note that some of the properties are integers (preceded by "I"), and some real numbers (preceded by "R"). Thus, for 100 properties and $10^8$ event this property space is about

40-80 GBs. Searching this property space is a major challenge. As we'll discuss in Section 5, we had to build a specialized index to be able to search this space efficiently.

Incidentally, the names of the properties are meaningful to physicists. For instance, Npip(3) stands for number (N) of pions (pi) positively charged (p = plus) and the 3 refers to the 3rd component (z-component). Similarly, NPbar(1) means number (N) of anti-protons (pbar) and again the 1 refers to the 1st component.

A typical analysis that physicists wish to perform on the reconstructed data involves the selection of some subset of the events according to some conditions over the summary information. An example of such a query is given below:

Query predicate: ((0.1 < AvpT(1) < 0.2) AND (100 < Npip(3) < 300)) OR (N(1) > 6000)

As can be seen, this is a "partial range query" in that the conditions on the properties are range conditions, and the query only specifies conditions on part of the properties (3 out of 54 in this example). The events that qualify for this query may be spread over many files and over many tapes. It is therefore essential to be able to determine ahead of time where these events are on tapes. The challenge we faced was to developing an efficient index over the large number of properties (100-200) and the large number of objects (the events).

We describe in this paper a system built for a high energy physics experiment scheduled to go on-line by the end of 1999, called STAR. Details about this experiment can be found in [1].

## 2. Optimization opportunities

The architecture of the Storage Manager was designed to take advantage of three aspects of the physics analysis that can improve the system performance. We refer to them as "optimization opportunities". Before we proceed we note that each file stored on tape usually contains many events (about 200-300 events per 1 GB file). The events that qualify for a given query will normally reside on multiple files, where only a fraction of each file will contain the event that qualify for the query. If no attention is paid to what goes into a file, which is the case when the events are stored in the order they are generated, then that fraction is small, typically less than 10% of each file is used by a query. Thus, it is important that when a file is in the cache, as many queries as possible access that file.

### 2.1 File caching order

Given a query that has objects (events) in multiple files, the question arises in what order to cache the files that contain the events that qualified for that query. In the case of physics data, each event is independent of another, and therefore analysis can proceed as soon as one file is cached. (This is similar to extracting data for an OLAP database to perform some statistical operation). The order of processing the files is irrelevant to the application. Having the freedom to choose which file to cache gives us an advantage. For example, it is often possible to determine if the query is of interest from the analysis of some fraction of the data. If it is determined that this query is of no interest, the query can be aborted. This implies that for such cases, it is better to cache first the files that have the largest number of events that qualify for the query, since the partial analysis will access fewer files. Another consideration for file caching order is to prefer files that have the larger number of queries potentially needing for them, or files that are smaller in size. In the first implementation, we have chosen to cache files according to the number of events that qualify in it for the query in question.

### 2.2 Multi-query overlap

Another opportunity for access time improvement is to have users share files that were cached to disk whenever possible. Given a query, we use an index according to the properties associated with each object (event), in order to determine which events qualify for the query and which files they are stored in. Thus, given a query we can determine the files that we need to cache. If any of these files are in cache because another query is using them, we can immediately make these files available to the new query. This strategy pays off well if the set of queries overlap the multi-dimensional regions, which is the case if several investigators are working on similar phenomena. Indeed, it is the practice in the HENP community of that pre-selected subsets are extracted, and if they are small enough, they are stored on disk cache for the long term. However, this practice has its limitations in that only a relatively small number of such subsets can be cached, and thus the physicists cannot ask for ad-hoc subsets to be retrieved.

### 2.3 Query estimation

Retrieving large subsets of data from tape-based datasets is very expensive both in terms of computer and system resources, and in the length of time for a query to be satisfied. Therefore, it is quite useful to provide an estimate of the amount of data and the number of files that need to be retrieved for a potential query. Often, users start a query, get frustrated with the length of time to perform the analysis, and abort the query. An important optimization strategy is to prevent such non-productive activities by providing a quick estimate of the size of the returned data, and a time estimate of how long it will take to retrieve the data. As is discussed below we designed and developed such a "query estimator" by taking advantage of the index used to evaluate a query.

## 3. The Storage Manager Architecture

The Storage Manager is responsible for determining, for each query request, which events and files need to be accessed. A specialized index (called a "compressed bitmap" index, that will be described in the next section) is constructed and is used for quick (real-time) estimation of the number of events that qualify given a query. The SM has 3 main components that represent its 3 functions. 1) The <u>Query Estimator</u> (QE), that uses the index to determine what files and what events are needed to satisfy a given partial range query.  2) The Query Monitor (QM), that keeps track of what queries are executing at any time, what files are cached on behalf of each query, what files are not in use but are still in cache, and what files still need to be cached. The QM consults an additional module, called the caching policy module, that determines what file to cache next according to the policies selected by the system administrator.  3) The Cache Manager (CM), that is responsible for interfacing to the mass storage system (HPSS) to perform all the actions of staging and purging files from the disk cache.

The Storage Manager (SM) interfaces to other components of the system.  While we only concentrate here on the Storage Manager, it is important to understand how it interfaces to the rest of the system.  Figure 2 shows the system components and the interaction between them.  On the right are the three components of the Storage Manager.  On the left there are two components: 1) the Query Object, which is the component that initiate the query, and passes the events one by one to the analysis program, and 2) the Event Iterator, which is the component that reads the events one by one from the file system. In addition, the diagram shows the mass storage system (HPSS) and the file system (UNIX).

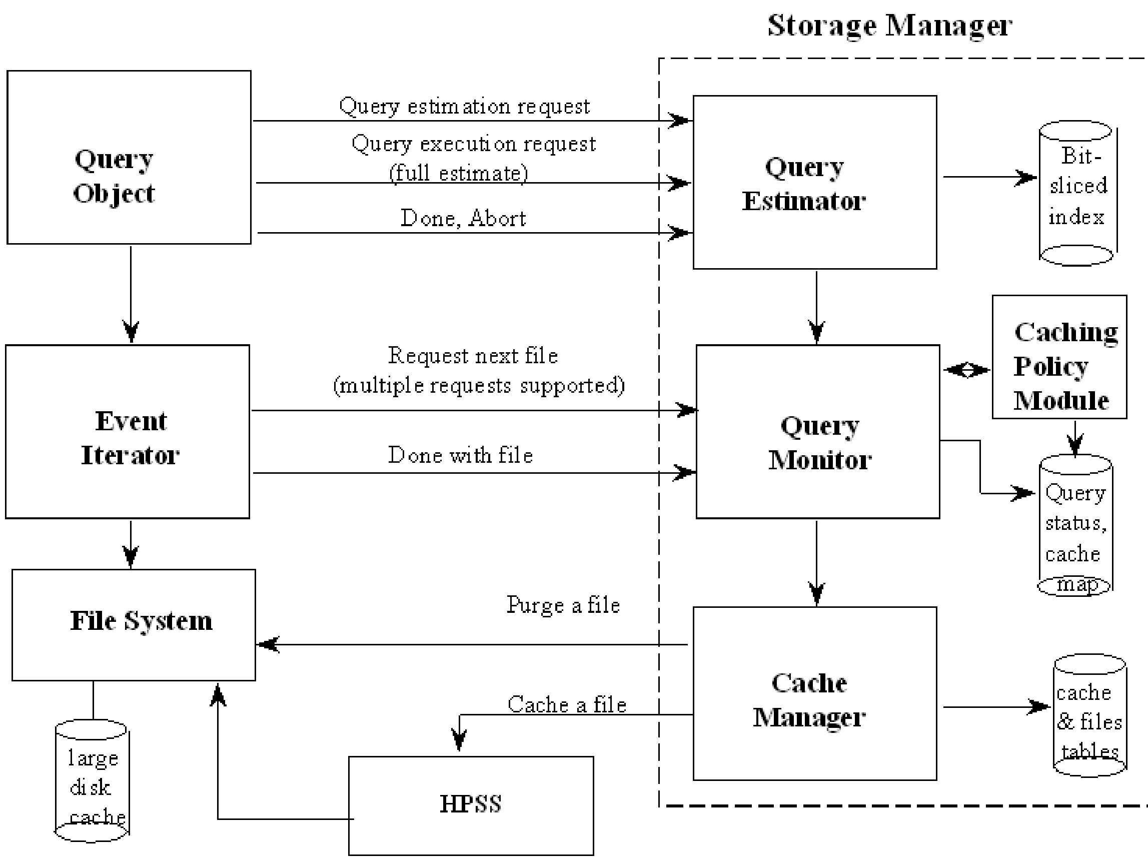


Figure 2: The architecture of the storage system and its interaction with other system components

The labels on the arcs are intended to show the type of messages sent between the components (via CORBA interfaces). In order to appreciate the value of a modular design that makes such a complex system manageable, we discuss below the steps taken by the system in order to process a query.

1) A "query estimation request" is sent from the Query Object to the Query Estimator (QE). The QE uses its index to estimate the total number of events that will result from such a query, the number of files involved, and how long it will take to process this query. Note that even small queries involving 10s of files make take hours to download from tape and process.
2) If the user decides to proceed, he issued an "execute" command to the QE. (Note that query estimation can also be used by the system to prevent user from proceeding with large queries if they do not have proper user permissions.) The QE uses its index to generate a set of files that the query needs to access, as well as a set of object_IDs for the events that qualify for the query.
3) The Query Object then starts an Event Iterator module that issues one or more "get next file" requests to the Query Monitor.
4) The QE passes the (file, object_IDs) request to the Query Monitor (QM).
5) The QM adds the query to its "query queue". When it is the query's turn to be serviced, the QM consults the Policy Module which file to give the query next.
6) The QM checks what in the cache. If a file is found for the query, it is selected, and "locked" into cache. A counter is used to mark how many queries have a lock on the file. If no file is found in cache, the Policy Module selects a file to be cached from tape according to its policy. If necessary, it also selects which unused files to remove from cache to make room for the new file.
7) If a file has to be read from tape, the QM requests the Cache Manager (CM) to cache that file. The CM requests the caching of the file from the (HPSS) mass storage system (currently via parallel FTP), and monitors its progress. When the file is cached, the QM is notified.
8) The QM passes the Event Iterator the file_ID along with the set of Object_IDs that qualified in this file for the query.
9) When the Event Iterator is finished processing all the objects in a file, it issues a "done with file" message to the QM. The QM then makes the query "active" and services it in its turn, for the next file.
10) When all requested by a query are processed, the QM notified the Event Iterator that there are no more files, and this causes the Query Object to terminate the query, by issuing the "done" message to the QE. The whole process can also be stopped with an "abort" message from the Query Object.

The details of the three main components (QE, QM, and CM), as well as the Caching Policy (CP) module refer back to the main <u>Storage Manager</u> page.