Multidimensional Indexing and Query Coordination for Tertiary Storage Management

A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim
National Energy Research Scientific Computing Division
Lawrence Berkeley National Laboratory

Abstract

In many scientific domains, experimental devices or simulation programs generate large volumes of data. 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. In this paper we describe the architecture and implementation of a Storage Access Coordination System (STACS) designed to optimize the use of a disk cache, and thus minimize the number of files read from tape. We achieve this by using a specialized index to locate the relevant data on tapes, and by coordinating file caching over multiple queries. We focus on a specific application area, a high energy physics data management and analysis environment. STACS was implemented and is being incorporated in an operational system, scheduled to go on-line in the end of 1999. We also include the results of various tests that demonstrate the benefits and efficiency gained of using the STACS.

1. Introduction

Today, the term "a large dataset" refers to hundreds of terabytes or even petabytes of data. While datasets of hundreds of gigabytes can be managed by large disk caches, it is too costly (and will continue to be so for the foreseeable future) to store petabytes of data on disk caches. Many scientific and business domains generate very large volumes of data that are stored on tertiary storage, typically robotic tape systems. Some examples are large scale simulations for climate modeling, combustion modeling, high energy and nuclear physics (HENP) experiments and satellite data. In such applications, 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. In this paper we describe the architecture of a Storage Access Coordination System (STACS). The system uses a specialized bit-sliced index to locate the data that need to be read from tape files. This information is used to coordinate file caching for multiple queries, thus minimizing the number of files read from tape.

To explain the details 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 additional 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 [1], 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 the particles it produced and summary properties for each event are generated (such as the total energy of the event, momentum and number of particles of each type). The number of summary elements extracted per event is typically quite large (100-200). 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 (http://www.sdsc.edu/hpss/) – 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 [2] 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.

Figure 1 shows a real example of data values for one such event for 54 properties.
Note that some of the properties are integers (preceded by "I"), and some are real numbers (preceded by "R"). Thus, for 100 properties and 10^8 event this property space is about 20-40 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 the number (N) of anti-protons (pbar) and 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. One of the challenges we faced was to develop an efficient index over the large number of properties (100-200) and the large number of objects (10^8 events).

2. Optimization opportunities

The architecture of the Storage Access Coordination System (STACS) was designed to take advantage of 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). For a given query, only a subset of the files needs to be accessed, and only a fraction of the events in each file is needed 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.

```
I N(1) 9965
I N(2) 1192
I N(3) 1704
I Npip(1) 2443
I Npip(2) 551
I Npip(3) 426
I Npim(1) 2480
I Npim(2) 541
I Npim(3) 382
I Nkp(1) 229
I Nkp(2) 30
I Nkp(3) 50
I Nkm(1) 209
I Nkm(2) 23
I Nkm(3) 32
I Np(1) 255
I Np(2) 34

I Np(3) 24
I Npbar(1) 94
I Npbar(2) 12
I Npbar(3) 24
I NSEC(1) 15607
I NSEC(2) 1342
I NSECpip(1) 638
I NSECpip(2) 191
I NSECpim(1) 728
I NSECpim(2) 206
I NSECkp(1) 3
I NSECkp(2) 0
I NSECkm(1) 0
I NSECkm(2) 0
I NSECp(1) 524
I NSECp(2) 244
I NSECpbar(1) 41
I NSECpbar(2) 8

R AVpT(1) 0.325951
R AVpT(2) 0.402098
R AVpTpip(1) 0.300771
R AVpTpip(2) 0.379093
R AVpTpim(1) 0.298997
R AVpTpim(2) 0.375859
R AVpTkp(1) 0.421875
R AVpTkp(2) 0.564385
R AVpTkm(1) 0.435554
R AVpTkm(2) 0.663398
R AVpTp(1) 0.651253
R AVpTp(2) 0.777526
R AVpTpbar(1) 0.399824
R AVpTpbar(2) 0.690237
I NHIGHpT(1) 205
I NHIGHpT(2) 7
I NHIGHpT(3) 1
I NHIGHpT(4) 0
I NHIGHpT(5) 0
```

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

2.1 File caching order

Given a query that has objects (events) in multiple files, it is necessary to determine what order to cache the files 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 any 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, the user may abort a query after he/she analyzes some fraction of the data. 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 needing them, or files that are smaller in size. In the current implementation, we have chosen to cache files that are needed by the largest number of queries, so that these queries can proceed as soon as the file is cached.

2.2 Multi-query overlap

Another opportunity for access time improvement is to share files in cache to whenever possible. Given a query, we can use an index over 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, we can determine the files that are needed for each query in advance. 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, which is the case if several investigators are working on similar phenomena. Indeed, it is the current practice in the HENP community to extract small pre-selected subsets (called micro-DSTs) based on some features to be used by several collaborators. These subsets are stored for the long term on disk cache for their joint investigation. The extraction of these subsets is performed by full scans over the data. However, this practice has its limitations in that only a relatively small number of such subsets can be cached, and it is impractical to wait for a full scan of the data when other ad-hoc subsets of the data are needed. In order to extract ad-hoc subsets, it is essential to have an index over the properties of all events.

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 requested 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” using the same index that evaluates a query.

2.4 Minimize tape mounting

Another opportunity for optimization is to read files from the same tape for the set of queries currently in the system whenever possible. This strategy is partially performed by the mass storage system we are using – HPSS. When multiple file transfers are requested, HPSS chooses to transfer files from tapes that are already mounted regardless of the order they were requested. However, a storage coordination system can take advantage of knowledge of file location for all the files needed by queries, and schedule files to be cached based on their co-location on tapes.

3. The STACS Architecture

The STACS is responsible for determining, for each query request, which events and files need to be accessed, and for scheduling the caching of files from tape to disk cache. A specialized index (a compressed bit-sliced index, that will be described in the next section) is used for quick (real-time) estimation of the number of events that qualify given a query. STACS has 3 main components that represent its 3 functions. 1) The Query Estimator, which determines what files and what events are needed to satisfy a given partial range query. It uses the index to perform a quick estimation, as well as generating a precise list of events and files for query execution. 2) The Query Monitor, which 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, which interacts with the mass storage system (HPSS) to perform staging of files, and purges files from the disk cache when space is needed.

STACS interfaces to other components of the system. While we only concentrate here on the storage coordination function, 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 STACS. 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).

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 take many minutes to hours to download from tape. Query estimation can also be used by the system to prevent the users from proceeding with large queries if they do not have proper user permissions to perform such large queries.
2) If the user decides to proceed, he/she issues an "execute" command to the QE. 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.

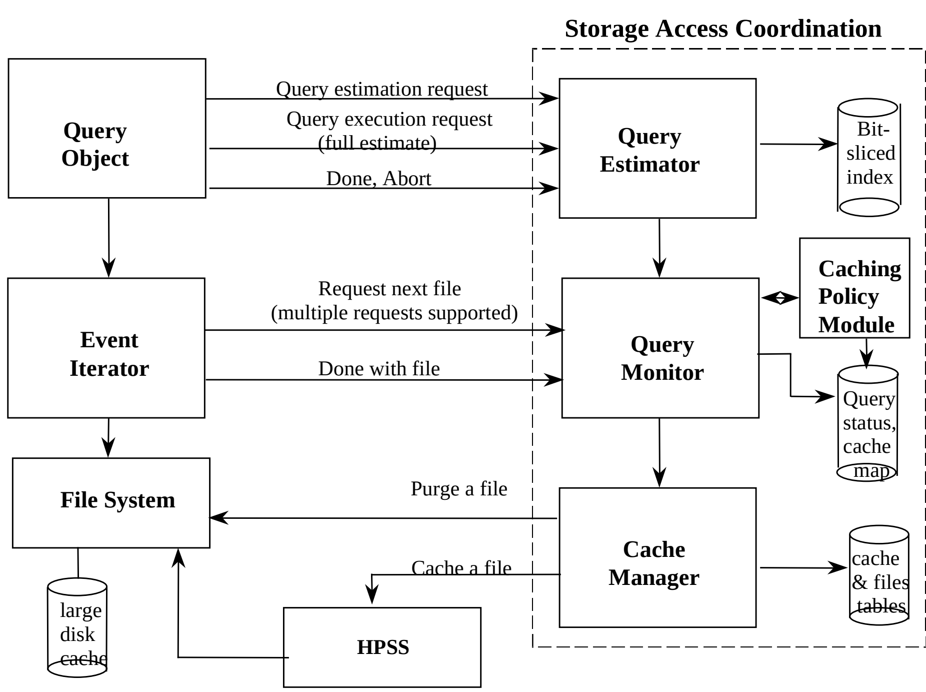

Figure 2: The architecture of the storage system and its interaction with other system components

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 needs to determine which file to give the query next.
6) The QM checks what is 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 for events in this file that qualify for the query.
9) When the Event Iterator is finished processing all the events 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.
10) When all the files 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.

4. The caching policies

Cache management is a complex problem. Although it has been studied in the context of memory management for files cached from disk, the problem of caching files from tape to disk is different in several respects: 1) files have to be read in their entirety, not one "page" at a time; 2) the cost of reading a file from tape is so high (can be up to 2-3 minutes, depending if the tape is mounted) that the most important thing to optimize is the sharing of files by multiple queries whenever possible.

We have addressed this problem by separating the "policy module" from the mechanism that runs the queries (the QM), and broke the policy decisions into 5 aspects as follows.

1. File weight policy

File weights can be assigned on the basis of various criteria, such as: a) the number of events per file that qualify for a query, b) the number of queries in the query queue that need the file, or combinations of these. Initially, we have chosen the criteria a) because it helps minimize the number of files cached for aborted queries. However, we are now favoring b) since the file weight can be used to keep files in cache that are needed by other queries.

2. Query service policy

The QM keeps a queue of all queries according to their arrival time. The order of servicing queries can use various policies, such as Round Robin (RR) or Shortest Query First (SQF). Care also needs to be taken that queries are not starved perpetually. For example, a query may be skipped if not enough space is found in the cache for any of its files. In principle, query service policies can be tuned to types of users or types of queries based on priority assignment. The default policy we are using is RR. Service for a query is skipped if the query has all the files it requested in cache (subject to pre-fetching limits – see below) and it is still processing them.

3. File caching policy

The file caching policy determines which file to cache when it is a query's turn to be serviced. If a file is found in cache, it is selected. Otherwise, the selection is based on the "file weight". The policy we are using is to pick first the file with the highest weight. If no space is found for that file, we select the file with the next highest weight, etc. We note here that the file caching policy can also prefer the caching of files that are on the same tape as recently requested files. This will tend to minimize the number of tape mounts. Note that this policy may conflict with caching files that are shared by a large number of queries. We have not attempted so far to combine these two policies.

4. File purging policy

The file purging policy is the policy that determines what to release from cache when a file is requested and cache space is needed. No file purging occurs until space is needed by some query. The policy is based on the "file weight" of the files in cache. The default policy is simply to purge the file with the smallest weight.

5. Pre-fetching policy

This policy states how many files will be pre-fetched for a query. For example, if this number is set to 2, then each query can have only 2 files requested at any one time. For these 2 requests, the system will either schedule a file caching from tape, or if the file is already in the cache, it is locked till the query processed it. If a high level of parallel processing on multiple files is anticipated, this parameter should be high. Incidentally, a pre-fetching number can be associated with a user's priority or even with each query. The current policy is to limit this parameter to 2, so that by the time a query finishes processing one file (can take from seconds to many minutes) another file will already be in the disk cache.

By separating the policy decisions into these 5 components, we can more easily control the policies and compare their effect on actual processing of data. We have begun simulation analysis as well as formal analysis of the problem, but have no results to report yet. We also instrumented the various modules to keep track of the behavior of the system. This will be discussed further in the section on tests performed.

5. The bit-sliced index

In this application, we are faced with indexing a very high dimensional space (100-200) over 100-500 million objects. Let's assume that we have 100 properties and 10^8 objects. This can be viewed as indexing a table 10^8 long and 100 wide. Each row represents an object with 100 values for its properties (columns). The index must respond to any range query over any subset of the properties.

It is well known that multi-dimensional indexes, such as Quad-trees and R-trees, do not scale to a high number of dimensions [3]. This is referred to as the "curse" of high-dimensionality. At best they scale up to 7-8 dimensions. In addition, these indexes work best with a full specification on all dimensions. Performing partial range search (say on 2 dimensions out of 7) results in having to read most of the index, which can be as bad as a sequential scan. One can choose to represent the 100 dimensional space in subsets of domains, say 7 at a time. However, searching any subset of the dimensions that do not fall in the same subset results in having to take intersections of the partial searches. Again, for partial range queries this adds to the inefficiency of these methods.

A recent paper [4] describes a technique, called the Pyramid-Technique, that partitions the d-dimensional space into 2d pyramids, and each pyramid is further partitioned into slices parallel to the base of the pyramid. This technique is shown to work best for data that is not skewed (random is best), and for the full hypercube queries (i.e. range conditions in all dimensions). Their experimental data shows that when the number of dimensions is small (i.e. 1-5 out of 100) the performance is close to sequential scan. Our requirements are on the opposite end of the spectrum: the data is skewed (that is the dimensions are correlated – such as the energy of an event collision and the number of particles generated), and the queries specify range conditions over a small number of dimensions.

Our approach was to take advantage of three aspects of this application.

1) The database, and, therefore the index are "read-only". Therefore we do not need an index that can be modified or updated. In reality, this database is "append-only", and for our application the index needs to be extended with periodic appends.
2) Since the number of properties specified in each query is relatively small (1-5 out of 100), a "vertical partitioning" method over the index can be used, so that only the partitions for the properties in the query are accessed.
3) The queries are predominantly range queries. As we will see, the index method is particularly suited for such queries.

We note that the above conditions are not that unusual in that they apply to other multidimensional database analysis environments, such as OLAP.

We describe next the principles guiding the index design.

We assume that the objects are stored in a certain order in the index, and this order does not change. We first generate vertical partitions for each of the 100 properties. The partitions are stored on disk. Now, the bit-sliced index is designed to be a concise representation of these partitions, so that it is much smaller and can be stored in memory. The details are explained next.

Since the properties we deal with are numeric, we can partition each dimension into *bins*. For example, we can partition the "energy" dimension into 1-2 GEV, 2-3 GEV, and so on. (For categorical data, one or more categories can be assigned to bins). We then assign to each bin a bit vector, where a "0" means that the value for that object does not fall in that bin, and "1" means it does. Figure 3 shows an example where Property 1 was partitioned into 7 bins, Property 2 into 5 bins, etc. Note that only a single "1" can exist for each row of each property, since the value only falls into a single bin.

This in itself does not bring to a significant concise representation. In fact, the total size may be larger than the partitions if the number of bins is large. For example, if the property values are integers (i.e. represented with 16 bits), and we partition the property into 20 bins, then the space for the 20 bit slices will be larger than the vertical partition of the integer values. However, we achieve a more concise representation by compressing the vertical bit-vectors. There is a large number of compression methods one can choose form, but as was pointed out in [5], it is advantageous to select a method that would permit boolean operations between the compressed vectors. That is, we wish to apply boolean operations on the bit-slices without decompressing them. We chose a modified version of run length encoding. A run-length encoded sequence simply stores the count of sub-sequences of 0's or 1's. The longer the "0" (or "1") runs the better the compression. For data that is highly skewed, as is the case for High Energy Physics data, the compression factor is high. Obviously, it is wasteful to store counts for short sequences. There are several variations of run-length encoding [6] that avoid counts for short sequences. The version we chose, avoids encoding short sequences into counts, by simply representing them as-is. We use 4 bytes segments for the counts, or as-is sequences. The high order bit in each 4 byte segment determines what each segment represents, a count or the actual bit sub-sequence. An additional bit is used to indicate whether the count is for a 0's or 1's.

Tests performed with real physics data show a compression factor of about 10. In one such test, we had 10 million rows and 70 properties (columns) where most of the values were real numbers. The total space required was about 2.5 GB. We partitioned each property into 20 bins. The total space for the compressed bit-sliced index was 280 MBs, or 4 MBs per property. For comparison purposes, we stored the 10 million x 70 table as a relation in Oracle, and indexed all 70 columns. The total space required by Oracle for the indexes was 9 GBs or about 120 MB per property.

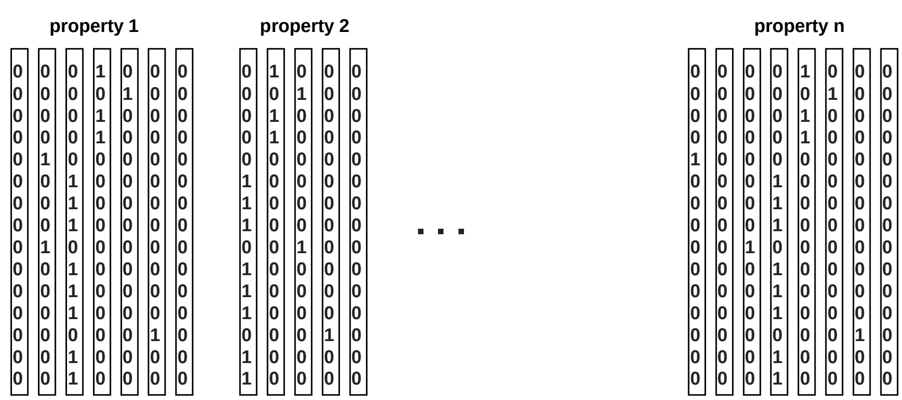

Figure 3: bit-slice representation of the vertical partitions

The choice of the number of bins and their boundaries is an important issue, but the main advantage to our method is that it is not necessary to keep all the bit slices in memory. We can bring from disk only the bit slices needed for the range conditions and keep in memory only the most popular “working set”.

5.1 Logical operations on bit slices

The compression scheme described above, permits logical operations on the compressed bit-slices (bitmap columns). This is an important feature of the compression algorithm used, since it makes it possible to do the operations in memory. These operations take as input two compressed slices and produce one compressed slice (the input for “negation” is only one bit-slice).

All logical operations are implemented the same way:

- The state ["0" or "1"] at the current position and the number of bits of the current run (number of consecutive bits of that same state), '*num*', from each bit-slice, are extracted (decoded).
- The result is created (encoded) by performing the required logical operation (AND, OR, XOR) on the state bits from each bit-slice and subtracting the smaller '*num*' from the larger and appending the result to the resulting bit-slice. The resulting bit-slice is encoded as we go along, using the most efficient method for the size of its run lengths.

An example: pseudo code for logical or.

```
function bmp_or( bitslice left, bitslice right )
{
      while( there_are_more_bits(left) and
      there_are_more_bits(right) )
      {
            lbit = decode( left, lnum );
            rbit = decode( right, rnum );
            result_bit = lbit | rbit;
            result_num = min( lnum, rnum );

            lnum = lnum - result_num, rnum =
            rnum - result_num;
            encode( result, result_bit, result_num );
      }
      return result;
}
```

This compressed bitmap index is used by the Query Estimator in two important cases. First, it is used when we need to quickly get an estimate of the number of hits for a query. We can give a quick answer, by giving a maximum and minimum number of hits. The minimum is given by the number of events that fall in bins that are covered entirely by the range query. In Figure 4, the dark rectangle shows the region covered by the range query. Range(x) shows 3 bins that are fully covered, and 2 “edge bins” that are partially covered. Range(y) falls on bin boundaries and therefore all its bins are fully covered. The maximum is determined by adding the minimum number of events plus those in the two edge bins.

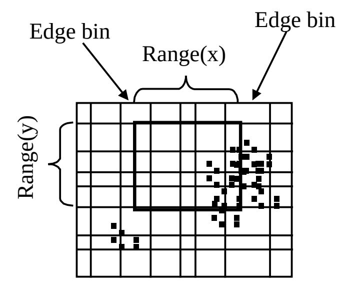

Figure 4: Edge bins are bins that are partially covered by the range condition

The second usage of the bitmap index is for a precise lookup. Such a lookup is needed when we execute a query. In this case we first find the events we know will satisfy the query. Again, those are the events in bins fully covered by the range of the query. For the events in the edge bins, we need to do a lookup in the full, disk resident, index. But, we only need to access the disk for the events in the edge bins. Note that there are at most 2 edge bins per dimension to lookup, in case that the range condition does not fall on a bin boundary. The disk access will require only one seek per event (in edge bins); we don't need to read more data than exactly for those events.

A range condition is performed with an "or" operations on all the bins involved. For queries involving several attributes, we perform bitmap index lookups for each of them and then do the final boolean logic (most often “and”) on the resulting, compressed, bit-slices.

Figure 5 illustrates the index operation. As shown on the left, a conjunctive range query can be viewed as a (multidimensional) rectangle on

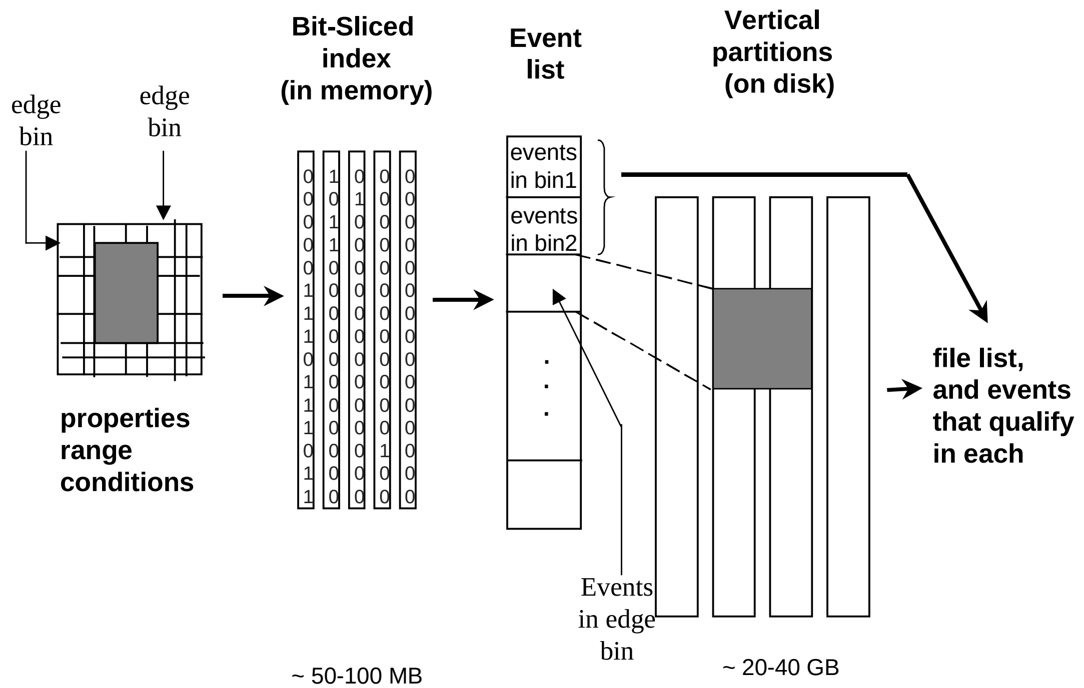

Figure 5: the use of the bit-sliced index for partial range queries

top of the grid. For the bins that are fully covered by the query the bit-sliced index is used directly for the result. For the edge bins the vertical partitions need to be checked for each of the dimensions. This technique minimizes the access to the full vertical partitions, resulting in a large gain in performance.

To summarize, we gain efficiency by using the bit-sliced index because:

1) we search only the dimensions in the query;
2) for each dimension we use only the bit-slices that the range specification spans;
3) for these dimensions we get all the qualified objects of fully covered bins directly from the bit-sliced index;
4) we need to check the disk resident vertical partitions values only for events in edge bins for the dimensions in the query.

6. Real time tests

The system has been tested in a real environment. While running the test, we logged various events, such as when a request was made for caching, when a file was cached, and when a file was passed to the analysis code. By plotting the behavior of different tests we could not only confirm the correct behavior of STACS, but also see the value of using an index to determine the entire set of files that queries need ahead of time. By doing so, we could share files in cache, and prevent files from being removed from cache if they are needed by other queries at a later time.

The graph shown in Figure 6 illustrates the value of file sharing in cache. We started with an empty cache. We ran 3 queries: the first needing 8 files that were cached from tape to disk; the second also needed 8 files, but 4 of them are in common to the first query; the third also needed 8 files, but 6 of them are in common to the first and second. We then ran all three queries concurrently. The entire test ran for about one hour.

The graph in Figure 6 was drawn from the actual logging of the test run. The method of displaying the runs is explained below. It is based on a visualization tool developed at LBNL (called NetLogger [7]) that was applied to show the progression of events over time.

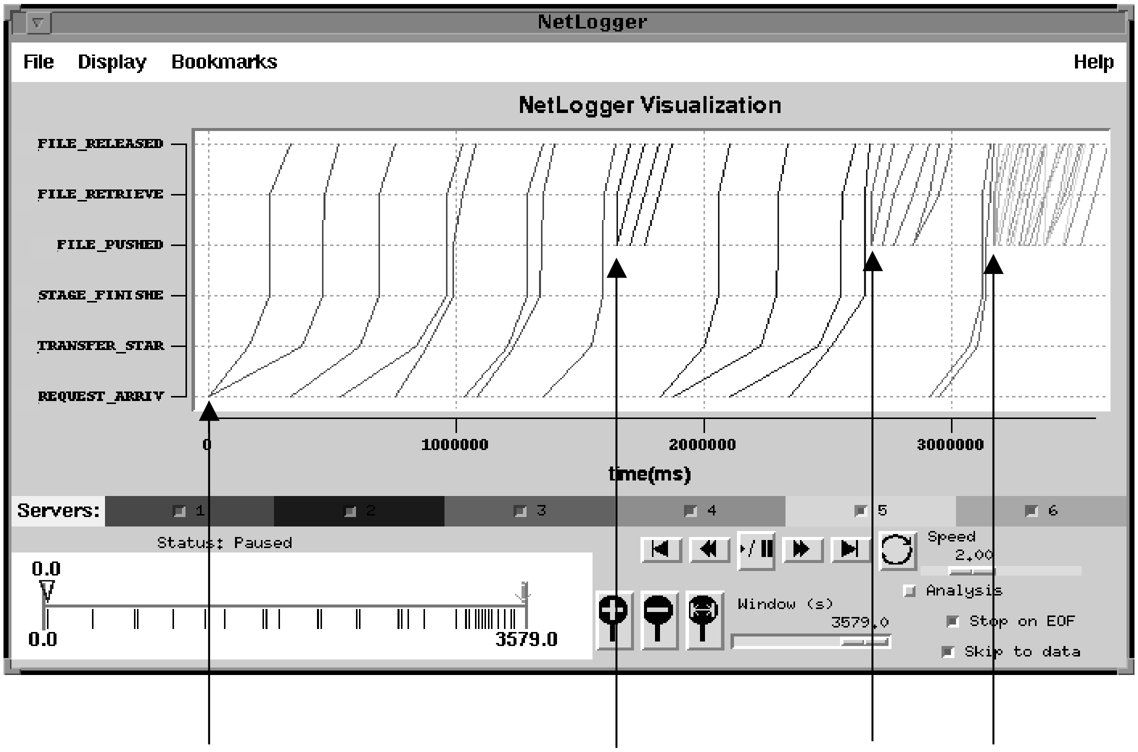

Figure 6: Display of a test run showing file sharing in cache

The graph represents the occurrence of logged events over time (the x-axis), but spreads out various logged events in the y-axis. In this graph, six logged events are shown from bottom to top: a) request_arrived (to HPSS), b) transfer_start (from HPSS-cache to local cache), c) stage_finished, d) file_pushed (i.e. file is available for the Event Iterator), e) file_retrieved (by the Event_Iterator), and f) file_released.

We note that:
-- The time between a) and b) is the time to get the file from Tape to HPSS-cache.
-- The time between b) and c) is the time to transfer the file from HPSS-cache to local cache.
-- The time between c) and d) is the time needed for the file to be passed to the EI after it was cached. This should normally be practically immediate.
-- The time between d) and e) is the time between making a file available to an EI and the time it actually reads it.
-- The time between e) and f) is the time that it takes the analysis code to process all the events from that file.

Thus, a jagged vertically connected line represents a single file and chronicles how it was processed. The graph shows that when Query 1 started, two file requests were made. The first file took several minutes to cache, and was passed to the query for processing. The file was processed for several minutes, and only when the query finished and released the file, a new request was made, reflecting the two file pre-fetching policy. The first query cached 8 files and processed them. When Query 2 was launched, the query was handed first the 4 files that were in cache. This can be seen from the short vertically connected lines starting at "FILE_PUSHED" line (reflecting that they did not have to be cached again). The remaining 4 files had to be cached. Thus, the second query ran in almost half the time. Similarly, Query 3 shared 6 files from cache and cached only two. Finally when all three queries ran simultaneously, all the files were shared from cache, and the total processing time for all queries was quite short. This test validated the correctness of the performance of the software, as well as the benefits of file sharing.

7. Implementation Issues

The system is implemented in C++ on a UNIX platform. One of the more challenging implementation issues we encountered was module interconnections. Since it may be advantageous to have each component of STACS reside on a different machine (e.g., the Cache Manager may be on the same machine that support the Mass Storage System), we chose to use CORBA for interfacing between the modules. A critical issue was that each of these components needs to deal with multiple requests concurrently, so we needed to use an ORB that supports multi-threading. This limited us with the choice of such products. Another aspect that we have not checked thoroughly is passing very large objects sets with CORBA. If this is not handled properly we may be forced to cut large objects into smaller chunks.

8. Lessons learned

1. The ability to use an index over the 10^8 objects is extremely valuable in being able to provide file sharing. File sharing is especially useful to scientists that are investigating the same phenomena, since they are getting files that cover the same region.

2. The index is also useful for estimating the cost (MBs retrieved, time, etc.) of running the query. This may provide information to users to tighten their query (smaller ranges), or to prevent naïve users from launching overly large queries and clogging the system.

3. Conventional multidimensional indexes do not scale to very high-dimensional spaces. If in addition, for partial range queries (i.e. if the number of dimensions in the query is small, e.g. 3-5 out of 100), we found the ideas of vertically partitioned files and bit-sliced indexing very useful.

4. Organizing the architectural design into separate functional modules was essential for large projects of this type. It is also useful to have well defined interfaces by using the CORBA IDL.

5. Having a storage access coordination system between the application software and the mass storage system (MSS) was not only valuable in terms of efficient caching. It also insulated the application from transient failures of the MSS. In case of a transient failure, STACS simply waits and re-issues caching requests periodically until the MSS recovers. We saw that in several tests. This is extremely important for long running queries (sometimes many hours or even days), where it is prohibitive to restart a query run because of transient failures of the MSS or the network.

Acknowledgement

This project is funded by the Grand Challenge program at the Department of Energy, in the Office of Energy Research, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

We thank all the collaborators of this project who have developed other components of the system (the Query Object, the Event Iterator, etc.), and who provided the realistic requirements for the storage access coordination component of this project. Special thanks are due to Dave Malon at Argonne National Laboratory, Doug Olson and David Zimmerman at Lawrence Berkeley National Laboratory, and Jeff Porter and Torre Wenaus at Brookhaven National Laboratory. We are also grateful for the very useful comments made by the referees.

References

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

[2] L. Bernardo, H. Nordberg, D. Rotem, and A. Shoshani, Determining the Optimal File Size on Tertiary Storage Systems Based on the Distribution of Query Sizes, Tenth International Conference on Scientific and Statistical Database Management, 1998. (http://www.lbl.gov/~arie/papers/file.size.ssdbm.ps).

[3] Roger Weber, Hans-Jörg Schek, Stephen Blott: A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. VLDB 1998: 194-205.

[4] S. Berchtold, C. Bohm, H. Kriegel, The Pyramid-Technique: Towards Breaking the Curse of Dimensionality, SIGMOD 1998: 142-153.

[5] P. O'Neil, "Informix Indexing Support for Data Warehouses," Database Programming and Design, v. 10, no. 2, February, 1997, pp. 38-43.

[6] Theodore Johnson: Coarse Indices for a Tape-Based Data Warehouse. ICDE 1998: 231-240.

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