LBNL-45551

New Capabilities in the HENP Grand Challenge Storage Access System and its Application at RHIC

L. Bernardo[1], *B. Gibbard*[2], *D. Malon*[3], *H. Nordberg*[1], *D. Olson*[1], *R. Porter*[2], *A. Shoshani*[1], *A. Sim*[1], *A. Vaniachine*[1], *T. Wenaus*[2], *K. Wu*[1], *D. Zimmerman*[1]

[1]Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[2]Brookhaven National Laboratory, Upton, NY, USA
[3]Argonne National Laboratory, Argonne, IL, USA

Submitted to Computer Physics Communications, Proceedings of
Computing in High Energy and Nuclear Physics, Padova, Italy, February, 2000.

Abstract

The High Energy and Nuclear Physics Data Access Grand Challenge project has developed an optimizing storage access software system that was prototyped at RHIC. It is currently undergoing integration with the STAR experiment in preparation for data taking that starts in mid-2000. The behavior and lessons learned in the RHIC Mock Data Challenge exercises are described as well as the observed performance under conditions designed to characterize scalability. Up to 250 simultaneous queries were tested and up to 10 million events across 7 event components were involved in these queries. The system coordinates the staging of "bundles" of files from the HPSS tape system, so that all the needed components of each event are in disk cache when accessed by the application software. The caching policy algorithm for the coordinated bundle staging is described in the paper. The initial prototype implementation interfaced to the Objectivity/DB. In this latest version, it evolved to work with arbitrary files and use CORBA interfaces to the tag database and file catalog services. The interface to the tag database and the MySQL-based file catalog services used by STAR are described along with the planned usage scenarios.

keywords tag,tape,MySQL,query,analysis,HPSS,STAR,RHIC,CORBA

Introduction

The High Energy and Nuclear Physics Data Access Grand Challenge project[1] has developed an optimizing storage access software system in collaboration with RHIC[2]. This system is targeted at optimizing the access to data in tertiary storage in the data analysis environment of large-scale high-energy and nuclear physics experiments. The RHIC facility began accelerator-commissioning runs in June 1999 and is planning to begin a physics data-taking program starting during the summer of 2000. The STAR experiment data handling capabilities are currently being developed and integrated with this data access software. The initial implementation and architecture were reported at CHEP'98[3]. In this paper we report on the developments following the RHIC MDC1 in Sept. 1998.

Figure 1 shows the basic Grand Challenge architecture (GCA). Client processes contain the data analysis algorithms. They establish a CORBA connection to the servers in STACS (storage access coordination system). The index contains event attributes as well as references to the file in which each event component resides. This index is built from the experiment's tag

[1] http://www-rnc.lbl.gov/GC/
[2] http://www.rhic.bnl.gov/
[3] http://www.hep.net/chep98/

database and file catalog. Clients request events by specifying queries containing predicate conditions on the event attributes, usually in the form of range conditions (such as 500 < No_Pions <1000). These queries are submitted by the clients to STACS which then moves any necessary files from the tape storage system (HPSS) to the disk cache and returns sub lists of event identifiers (IDs) to the clients as the files become available on disk. Details of the multi-component event model, optimization features, file catalog and tag database are given in the following sections.

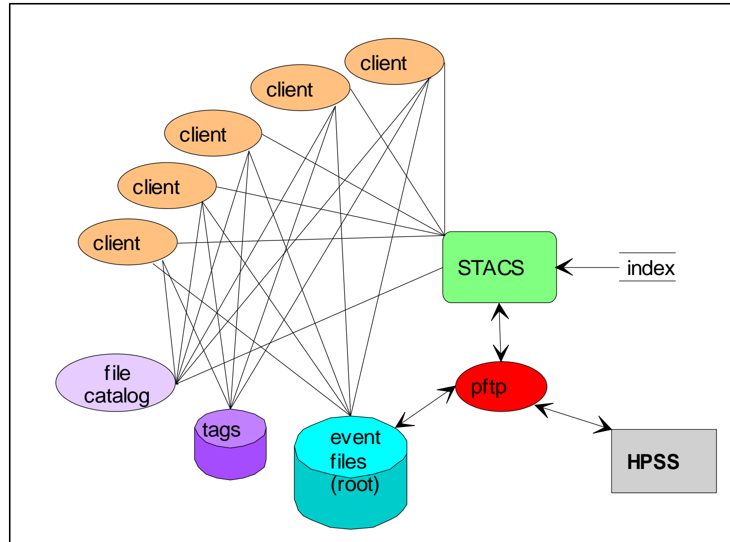

Figure 1. Illustration of basic software architecture.

The assumptions about the event data model are quite minimal. Each event is considered to be composed of named components. A single component of a single event is contained within a file. Separate components for the same event may reside in different files, or in the same file.

STACS implementation

STACS is the part of the system responsible for determining, for each query request, which events and files need to be accessed, determining the order of files to be cached dynamically so as to maximize their sharing by queries, requesting the caching of files from HPSS in tape optimized order, and determining dynamically which files to keep in the disk cache to maximize file usage. It uses a specialized index, called a bit-sliced index (which was described in detail in [Shoshani et al 99]) that is used for quick (real-time) estimation of the number of events that qualify for given a query. This index is also used to determine the set of files that have to be cached for each query, and the set of event IDs that these files contain for that query.

As shown in Figure 2, STACS has 3 main components that represent its 3 functions: 1) The Query Estimator (QE), that uses the index to determine what files and what events are needed to satisfy a given 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 Query Monitor consults an additional module, called the Caching Policy module, which determines what file to cache next according to the policies selected by the system administrator. 3) The Cache Manager, that is responsible for interfacing to the mass storage system (HPSS) to perform all the actions of staging files to and purging files from the disk cache. The Cache Manager controls the rate of PFTP submission to HPSS, so as not to flood it. The number of active PFTP 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 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 via CORBA interfaces. The Client modules (described in a later section) communicate with STACS by issuing query requests, asking for estimates of the numbers of events and the time to execute the query, issuing an execute request for the query submitted, and getting the information about files when they are cached. Our experience with using CORBA ORBs is described in [Sim et al 99].

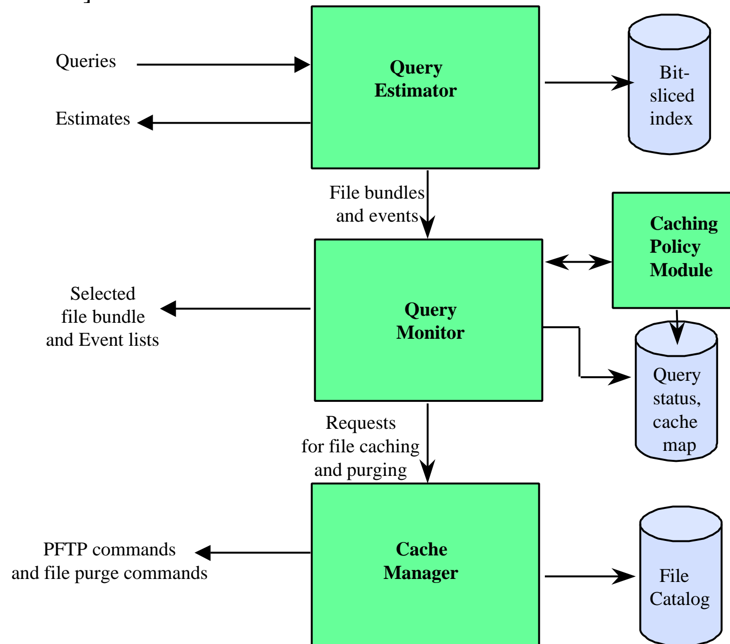

Figure 2. The Storage Access Coordination System (STACS)

The graph in Figure 3 was drawn from actual logging of a test run. The method of displaying the runs is based on a visualization tool developed at LBNL (called NetLogger [7]) that was applied to show the progression of logged actions. The graph represents the occurrence of logged actions over time (the x-axis), where the actions are stretched out in the y-axis. There are six logged actions shown from bottom to top: a) request_arrived (to HPSS), b) transfer_start (from HPSS-disk cache to local disk cache), c) stage_finished, d) file_pushed (i.e. file is available to the Client), e) file_retrieved (by the Client), and f) file_released (by the client). Thus, a vertically connected (crooked) line represents the history of a single file from the time of its request to be cached to the time of its release by the Client component.

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, after the first 8 files were cached by one query, a second query was issued that requested 4 of these 8 files. The short lines (9-12) show that these files were passed directly from cache. We will use the same kind of graph in the next section to show that caching of file bundles also worked correctly.

Handling multiple event components and file bundles

The system described above was operational during the first phase of the project and was tested during the Mock Data Challenge 1. In the second phase of the project, we set out to support multiple components per event. Given that each event is partitioned into several components, such as "tracks", "hits", and "raw", it was necessary to find a way of caching files in a

coordinated fashion according to the components requested in the query. Files are typically organized by component type, where a file of a certain type (e.g. tracks) contains only data for that type (tracks for one or more events).

We introduced the term "file bundle" to refer to the ordered set of files, one for each component, that need to be in cache *at the same time* to process events whose components are in these files.

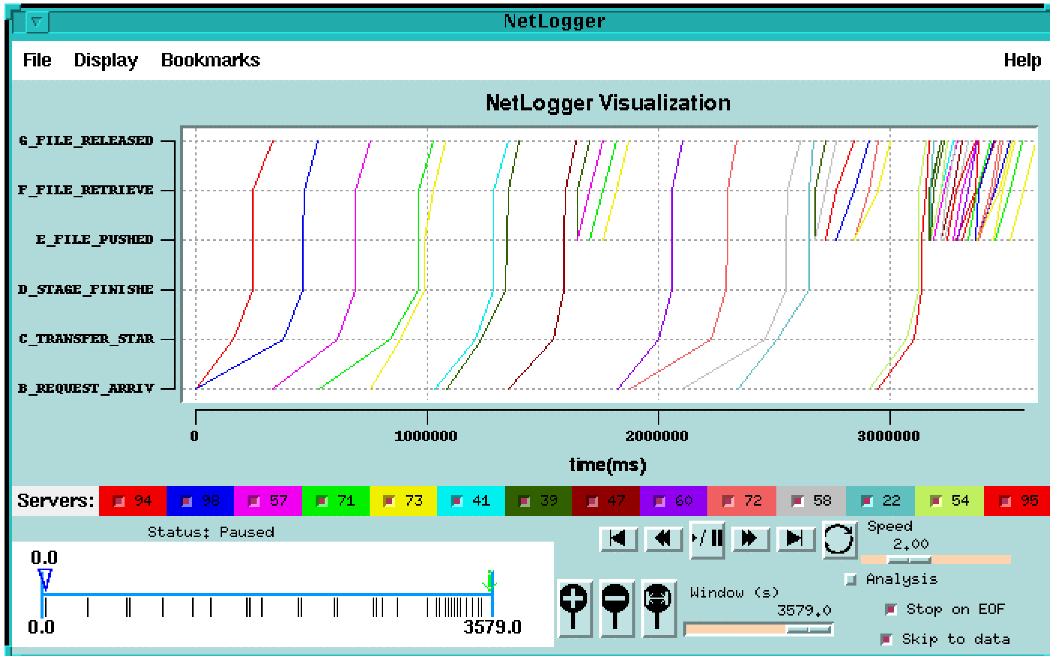

Figure 3. Monitoring file caching

For example, suppose that a query requested components c1 and c2. Suppose that files F13, F206 are 2 (of many) files for components c1 and c2, respectively. Assume that F13 and F206 contain event components in common for events {E7, E36, E102} for a particular query. Then, we refer to the ordered set <F13, F206> as the “file bundle” and to the set {E7, E36, E102} as the set of events associated with that file bundle for that query. It is often the case that the same file can appear in a another file bundle for a given query. Thus, <F13, F301> could be another file bundle for the same query. In general, a query can have a large number of file bundles, where some of the files can be in more than one file bundle.

The ability to select a subset of the components in the query required an extension to the query language. The following is an example pseudo language for such a query:

```
SELECT tracks, hits
WHERE 0<glb_trk_tot<10 & n_vert_total<3
```

We describe next the changes made in STACS to support multiple components. Naturally, the index had to be extended to support multiple components, and the Query Estimator (QE) search component modified. The part of the index that find the events that qualify for the query was unchanged. However, once the events are selected, the system now generates the file bundle for each event, and then combines duplicate bundles using a hash mapping. A much more significant effect was on the component that scheduled files to be cached, the Query Monitor (QM). Now, the QM has to have all the files of a bundle in cache before it can return to the Client. It needs to check if any of the files of the bundle are already in cache and request caching the remaining files of the bundle from tape. This is achieved by selecting policies on which files should be in cache at any one time. We have developed a methodology that is

based on policy components for assigning bundle weights, and on determining what should be left or removed from disk cache when servicing queries. We explain these policies next.

a) Assigning file weights and bundle weights

Given a query, files are initially assigned "file weights" according to how many bundles they appear in. As additional queries come into the system, the weight of a file is summed over all the queries in the system. Thus, we have:

File weight = SUM (all bundles for each query) over all pending queries.

For example, if there are 2 queries in the system, and file Fi appears in 5 bundles for query 1 and in 3 bundles for query 2, then Weight (Fi) = 8. The file weights are dynamically decremented after each file bundle is processed (i.e. released by the Client). The file weights are dynamically incremented for each new query request arrives to STACS. Now, the "weight of a file bundle" is simply the sum of the weights of all the files in the bundle.

b) Servicing queries

The Query Monitor 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 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. 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 – see below) and it is still processing them. Next we describe the policies for determining which files to cache, which to keep in cache, and which to remove from cache.

- Bundle caching policy

  This policy determines which bundle to cache when it is a query's turn to be serviced. This is based on the "bundle weight". Currently, 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 in the cache is selected. In addition, the default policy will pass a bundle to any query (out of RR order) that has all the files for any of its bundles in cache. This is also subject to the pre-fetching limit.

- File purging policy

  File purging policy is the policy that determines which files to purge (remove) from disk cache when cache space is needed. No file purging occurs until space is needed by some query. Rather than considering purging all files in a bundle, this policy is based on one file at a time. This is to insure that if a file is needed by more than one bundle, its purging is deferred as long as possible. The policy is based on the "file weight" of the files in cache. The current policy is simply to purge the file with the smallest weight provided that it is not currently in use by some query. In case of a tie, the largest file will be purged to gain the most amount of space (an alternative policy for ties is to choose the file longest in the cache since it was released).

- Pre-fetching policy

  This policy states how many bundles will be pre-fetched for a query. For example, if this is set to 2, then each query can have only 2 bundles requested at any one time. If a high level of parallel processing is anticipated, this parameter should be high. In principle, a pre-fetching number can be associated with each type of query or even with each query. For example, users can be assigned a priority level, and their pre-fetching parameter can be set accordingly. In the current version, query priorities were not implemented, and the default pre-fetching level is applied to all queries. The current pre-fetching default is 2, allowing for one bundle to be pre-fetched while another is processing. The pre-fetching level can be set dynamically.

Figure 4 illustrates the coordination of file caching according to bundles. In this test, we ran 2 queries, each with 3 components, 15 minutes apart. As can be seen, only after the 3 files of a bundle were cached the bundle was 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. A more detailed description of the file bundle coordination methodology can be found in [Shoshani et al 2000].

Results from scalability testing

The scalability testing was done for the purpose of finding areas where the system can potentially break as the number of events, files, and queries increases. A test dataset was set up for about 10 million events, each partitioned into 5 components, organized into some 4700 files, totaling about 1.6 TB. The tests were performed by launching a large number of queries (about 100) 5 minutes apart. The queries were for 2 components each, and ranged from 10s to 100s of bundles each. The total amount of time to run such a test to completion was about 18 hours. The tests were run on an HPSS system shared by other users. The total amount of disk cache available to STACS was 100GB. This was sufficient to hold 200-300 files at a time. The queries had a small overlap of the files, so that the cache was continuously purged to make space for new files. Initially, the limit for concurrent PFTP requests was set at 7, but when the HPSS system administrator complained, this was reduced to 4. At one test, the same 100 queries were launched two more times, concurrent with the first run, so as to have about 250 queries active at the same time. All tests were successful, and the system has been running for up to a week without any failure in our test.

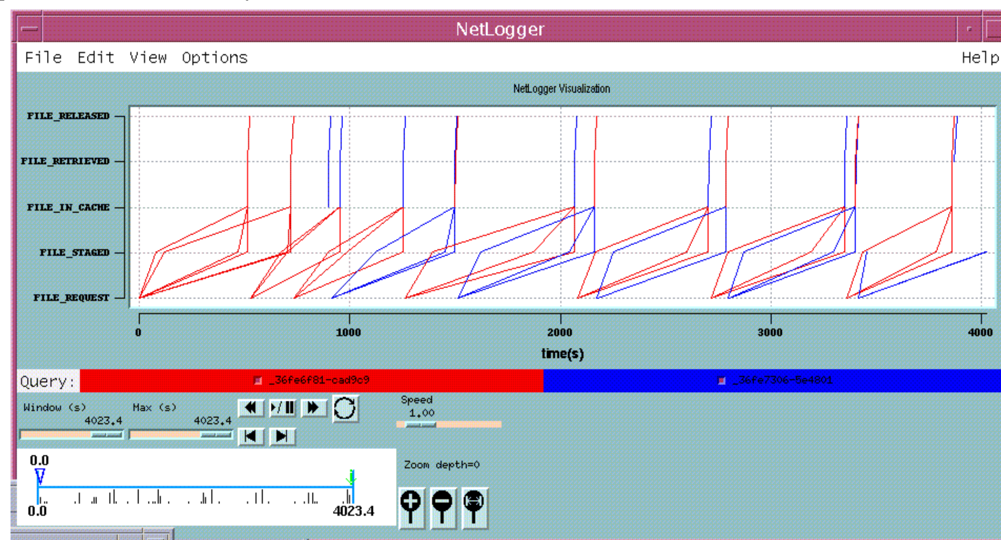

Figure 4. coordinating the caching of file bundles

There were several potential problem areas to be concerned about. (1) The Query Estimator and its index. It handled up to 100 concurrent queries OK. (2) The query queues and the bundle and file queues in the Query Monitor. It handled up to 250 concurrent queries in the query queue, and queues up to 1000s of bundles and files. (3) The queue in the Cache Manager also supported 100s of files.

An additional problem we were concerned about is the number of pending CORBA connections that the system can handle. This can be a complex issue, because the number of possible open CORBA connections depends on the operating system set up, and the amount of memory on the system. In our design, we made sure that CORBA connections are closed as

soon as possible. If a long delay was expected for a response, a "call back" was set up. This requires a CORBA client and a server on both ends of each connection. One exception was between the Client and STACS when it gets bundles. We wanted to avoid from making the Client a CORBA server, so we allowed bundle requests to "hang" till the files of the bundle were cached. In our tests, 2 such bundles were requested at a time, so we had 100s of connections open. Fortunately, this did not present a problem, and the system worked well. However, this may still be a problem, if we allow each query to request a large number of bundles at the same time (for distribution to parallel processors). 100s of queries requesting 10-100 bundles each, will cause 1000s of CORBA connections to be open. If this becomes an issue, we'll have no choice but make the Client be a CORBA server as well.

Client interface

The Grand Challenge (GC) software presents a client interface that consists of three primary components: a GCA_Resources service, a query object, and an order-optimized iterator. These components are often wrapped, in turn, by an experiment-specific layer of software to provide a view of data access that matches the experiment's analysis or reconstruction framework model. In this section we describe the GCA-provided application-independent client interface.

The GCA_Resources service initializes all client code connections to STACS components. Communication with STACS components is CORBA-based, but the GCA_Resources service insulates the client from the need to know about or deal with CORBA-specific constructs. The GCA_Resources service also handles configuration initialization, provides methods for clients to read and set configuration options, and serves as a factory for query objects.

The query object is the means by which clients make event selections and specify which components of qualifying events should be made available. Queries may be constructed from selection strings and predicates:

query = queryFactory->newQuery(select_string, predicate_string);

An event collection may also serve as a query, with a sequence of event IDs taking the place of the predicate string. The query object provides access to the functionality of the STACS Query Estimator, and may be used, therefore, to examine quantities like the number of qualifying events, the number of files that will be delivered, the total number of bytes to be transferred, and so on, allowing the client to understand the scope of her query prior to execution. Clients must explicitly invoke the query object's execute() method to set in motion the machinery of data delivery.

Every query has, upon construction, a unique token assigned to it; this is how the STACS components distinguish clients in a multi-user environment. The token is also used to initialize the order-optimized iterators--the means by which events are ultimately delivered, one at a time, to client applications.

The GC order-optimized iterator supports both ODMG-style iteration (ODMG is the Object Data Management Group [Cattell et al 2000]) and the interface of an STL forward iterator. The idea is that iteration over events should look to a client exactly the same as it would without the GC; only the order of event delivery is different. The following block illustrates one supported style of iteration:

```
// iterator is initialized with this query's token, and a
// pointer to GCA_Resources for access to remote STACS components
// and configuration parameters:
    OrderOptIter iter(query->token(), &GCA_Resources);
    while (iter.not_done() {
// inside the while block, *iter points to the current event
        usercode(*iter);   // process an event
        ++iter;
    }
```

Internally, the Grand Challenge components know nothing of "events"; they transmit and receive CORBA sequences of opaque structs, whose shapes and definitions are provided by the experiment. An object id as returned by the order-optimized iterator may be {run_number, event_number} for one experiment, an Objectivity/DB ooRef for another. (A cast or conversion operation may therefore be useful, e.g., usercode(make_d_Ref(*iter)) or usercode(make_experiment-specific-Ref(*iter)); alternatively, the iterator may be adapted (or templated) to perform the cast/conversion internally.

Parallel iteration is possible by initiating multiple iterators in different processes (possibly on different compute nodes) with a single query token.

STAR file catalog and tag database

The STAR experiment has adopted the MySQL database to keep records of data files and their production history. To simplify user queries for any particular file, records for all files are kept in one database table – fileCatalog. To enable storage of different file types in the same table, for every file the concept of file "producer" is used. Examples of file "producers" are: data acquisition run, geant simulation and event reconstruction (production) jobs. Since every file is stored in the primary repository – HPSS, records in the fileCatalog table also keep information about this primary file instance. Records of other possible instances (copies) of the same files are placed in a separate table.

The information specific to each producer is kept in other tables. Every producer has only one record in one of these tables. Since each of producers typically creates many output files, reference to that record in the fileCatalog table is used to resolve this one-to-many relationship. Since the production job can, in principle, have multiple files on input, an extra table is used to resolve the many-to-many relationship.

Every production job is created by a perl script that connects to the database server to create records for output files and relationships for input files. Another perl script monitors the production job status and updates records in the database accordingly.

During the event reconstruction a set of structures containing selected event information is saved as the tag component of the event. This information (event tags) is used to pre-select events for the time-consuming end-user analysis. The STAR event tags consist of overall event summary tags, tags from the data acquisition system, and a set of useful physics tags. The total number of tags (dominated by the tags for physics analysis) is about 500.

The file cataloguing scripts, which build the fileCatalog database, process this tag component for each file and store them in a separate tag table in the database. Although MySQL is a fast database, queries will be more efficient if there is enough space to keep all the table data in the database server memory. Since tags from one million events will result in a 2 GB table, the memory-resident bit-sliced index technology provided by the GC is a more efficient way to achieve high performance. Thus the tag database is mainly used to provide the tag information to the GC IndexBuilder.

Use of CORBA interfaces in the GC minimized the number of software components that have to be implemented in an experiment-specific way. To interface GC software for use in STAR only two components have to be modified: IndexFeeder and FileCatalog servers.

At the GC initialization stage the IndexFeeder component reads STAR tags from the MySQL database and forwards them to the IndexBuilder. During the run-time the multi-threaded fcFileCatalog server takes requests for file information from other GC software components, delegates them to the STAR MySQL database server and provides the requested results back.

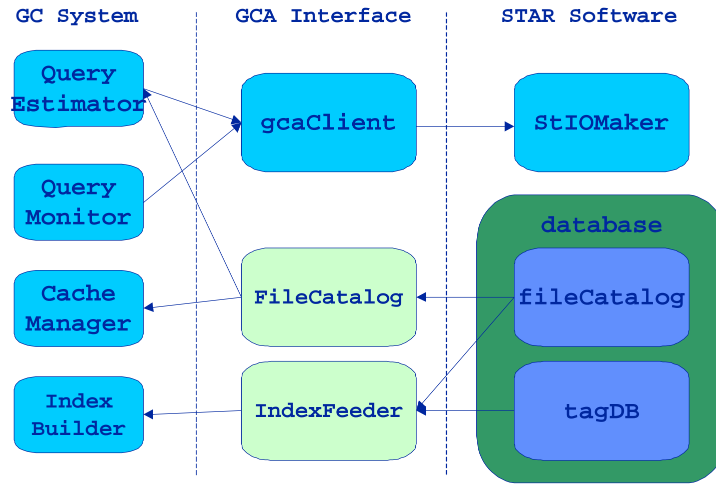

Figure 5. Interfacing Grand Challenge software to STAR experiment. Arrows show the direction of the data flow between software components. Colors indicate potential dependencies.

Figure 5 shows the simplified data flow diagram of the STAR-GCA interface. The data flow scheme is useful to indicate potential dependencies establishing between the software components. In particular it is beneficial to avoid creating the dependency between the two large software systems: STAR software and the Grand Challenge software. To eliminate this dependency we have implemented the modified Adapter pattern. Figure 6 shows the diagram of the software components involved.

During the STAR MDC3 in March/April 2000 the HENP Grand Challenge system has been integrated with STAR data analysis software. The completion of the integration of the GC server software (STACS) and the GC client code (GCAClient) with the STAR data analysis system as well as the implementation of the STAR file catalog and tag database enabled several new capabilities for STAR data analysis. The catalog of 120K events each with 170 tag attributes and 6 event components across 8K files has been loaded into the index for STACS. In

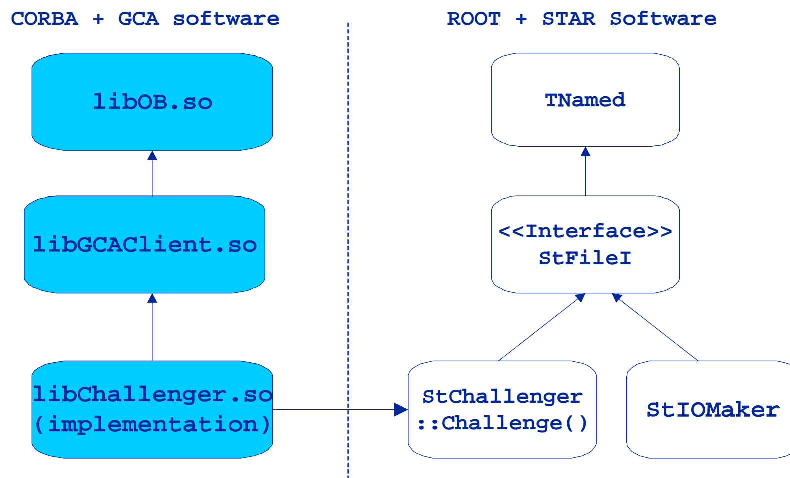

Figure 6. Eliminating dependencies between two software systems.

that way a 3.6 TB of STAR MDC3 data has been indexed and used for physics analysis with STAR software.

A typical usage scenario would be to respond to a request: "I want to run an analysis on 5000 central trigger taken between day 4 and day 8" from an end-user. The user analysis program formulates the corresponding GC query. This query is handled by the StIOMaker – a STAR software library component that uses GC services. In a simplified scenario the StIOMaker connects to the queryEstimator server and gets back a token for this query. The token is used in further calls: to get the time estimate for the query, to monitor query status, etc. The same token is used to communicate with the qmEventIterator component to execute the query and to retrieve selected events collected in a file bundle. When the event processing in a particular bundle is done, the StIOMaker calls the qmEventIterator to release the files.

Conclusion

The optimizing storage access software developed by the High Energy and Nuclear Physics Grand Challenge project has been updated to support a multi-component event data model that is particularly well suited to the large datasets of nuclear and particle physics experiments. This system has undergone scalability tests to a level of 10M events, 7 event components and 250 simultaneous queries. It has been integrated with the data management and analysis software of the STAR experiment at RHIC. It is currently being used to access simulation data stored in the HPSS system at the RHIC Computing Facility for purposes of testing and developing the analysis software. Heavy use of this data access mechanism is expected in the summer of 2000 when the experiment begins acquiring and processing the actual colliding beam data from the accelerator.

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).

[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).

[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).

[Cattell et al 2000] Cattell, R.G.G., et al, The Object Data Standard: ODMG 3.0, Morgan Kaufmann Publishers, 2000.

[Shoshani et al 2000] Coordinating Simultaneous Caching of File Bundles from Tertiary Storage, A. Shoshani, A. Sim, L. M. Bernardo, H. Nordberg, , to be published in the twelfth International Conference on Scientific and Statistical Database Management (SSDBM'00). (This paper can be downloaded from http://gizmo.lbl.gov/~arie/papers/download.papers.html).