Scientific Data Repositories - Designing for a Moving Target

Etzard Stolte, Christoph von Praun, Gustavo Alonso, and Thomas Gross

Department of Computer Science
ETH Zürich
CH 8092 Zürich, Switzerland

ABSTRACT

Managing scientific data warehouses requires constant adaptations to cope with changes in processing algorithms, computing environments, database schemas, and usage patterns. We have faced this challenge in the RHESSI Experimental Data Center (HEDC), a datacenter for the RHESSI NASA spacecraft. In this paper we describe our experience in developing HEDC and discuss in detail the design choices made. To successfully accommodate typical adaptations encountered in scientific data management systems, HEDC (i) clearly separates generic from domain specific code in all tiers, (ii) uses a file system for the actual data in combination with a DBMS to manage the corresponding meta data, and (iii) revolves around a middle tier designed to scale if more browsing or processing power is required. These design choices are valuable contributions as they address common concerns in a wide range of scientific data management systems.

1. INTRODUCTION

Scientific databases in general and astrophysical repositories in particular have been repeatedly identified as one of the open challenges facing the database community [13]. For instance, in a recent panel at VLDB 2002, it was hotly debated whether scientific databases should contain the actual data or only meta data pointing to files where the data resides [11]. Such repositories pose a wide range of problems. First, some of them have already passed the petabyte line [24, 23]. Second, and this is not often fully recognized, there are many critical design issues that go beyond fast query processing. Some of the most relevant are unpredictable usage patterns, the need to incorporate external processing tools to the data repository (SQL can select the data but it cannot process it), the need to accommodate constant change (of the raw data, of the usage patterns, of the analysis routines, etc.), and the fact that performance and efficiency is not measured by query response time but by the time a scientist needs to extract information from the data. Most existing systems nowadays tend to ignore these issues. In astrophysics, for instance, many data centers provide primarily data access via FTP. Advanced searches are done by downloading raw data and filtering the data using ad hoc scripts [13].

In this paper we describe our experience addressing these and other limitations of scientific repositories as part of the design and implementation of the RHESSI Experimental Data Center (*HEDC*)[1] [17]. HEDC is a scientific data warehouse that manages the high-energy solar observations generated by the NASA's Reuven Ramaty High Energy Solar Spectroscopic Imager (*RHESSI*)[2] satellite. The design philosophy behind HEDC has been that a successful data management environment should provide seamless access to arbitrary subsets and combinations of both local and remote data, as well as being compatible with the rich set of existing data analysis environments. These analysis environments need to be an intrinsic part of the system to allow users to both access the data and process it to obtain information.

In what follows we describe how these requirements have been met in HEDC. Although we use HEDC as motivation and context for the implementation, the techniques we discuss are not particular to solar astrophysics data. They can be used in a wide range of scientific applications and, hence, are an important contribution towards more efficient handling of scientific data. Some of these techniques include:

- Clear separation between data (raw and derived data) and meta data. The meta data is managed using a database. The data resides in file archives. There is also a clear separation between database schemas for meta data and actual data. This arrangement allows the system to accommodate constant change to the data without affecting the rest of the system.

- Extensive support for meta data management and data processing. For searching the meta data, users can use either visual tools to graphically render the search space, predefined queries, or their own SQL queries. For working with the data, users can use the standard

This research was sponsored, in part, by the Institute of Astronomy and the Department of Computer Science at ETH Zürich, and by grant TH-WI/99-2 from the VP for Research at ETH Zürich.

SIGMOD 2003, June 9-12, 2003, San Diego, CA.

[1] http://www.hedc.ethz.ch
[2] http://hesperia.gsfc.nasa.gov/hessi/

analysis routines provided with the RHESSI data or download the data and use their own routines. New analysis results thus produced may be uploaded and imported into the system.

- A comprehensive set of user interfaces to support the spectrum from the casual, non-specialist user (who can access the system through a Web page) to advanced users (who can create a local mirror copy of the entire HEDC server, including data and functionality).

- Support for approximated search, analysis, and visualization of both large data volumes and individual events. This turns HEDC into an interactive tool that significantly increases the capacity of scientists to explore and analyze data.

- A flexible architecture that can be adapted to multiple scenarios. In HEDC, where the current user community is small, we use a single server for the core of the system and a number of workstations and PCs for data processing. However, the entire architecture can be transparently extended to a cluster-based system with multiple Web servers, processing servers, and a distributed database.

In what follows, we describe the background for the project in more detail (Section 2). In Section 3 we discuss the requirements imposed on the system. Then we present the design tier by tier (Sections 4 – 6). In Sections 7 and 8 we empirically validate some of the ideas discussed. We conclude the paper after discussing related (Section 9).

2. PROJECT BACKGROUND

2.1 NASA's RHESSI

RHESSI was launched on February 5, 2002. Its primary scientific objective is to understand particle acceleration and explosive energy release in the magnetized plasma at the Sun. RHESSI combines an imaging system consisting of 9 rotating modulation *collimators* (each made up of a front and a rear segment), each with a high-spectral resolution germanium detector (GeD) covering energies from soft X-rays (3 keV) to high-energy gamma-rays (20 MeV). RHESSI's hard X-ray imaging spectroscopy provides spectral resolution of 1 keV, spatial resolution down to 2 arcsec, and temporal resolution as short as tens of milliseconds.

Since launch, RHESSI has been generating 2.0 GB of *raw data* per day. The data produced is buffered and forwarded to a ground station at pre-established intervals. This raw data stream is analyzed for possibly relevant events, segmented along the time axis, packaged into units of roughly 40 MB, formatted as Flexible Image Transport System (FITS) files[3] and compressed using gnu-zip. If relevant events are detected, some summary data and a number of *derived data products* (mostly images) are generated which are then attached to the raw data unit. This *standard catalog* of events is meant as a starting point for analysis. Within about 24 hours, the raw data is sent to HEDC and two other repositories at the Space Science Lab (UC Berkeley) and at NASA's Goddard Space and Flight Center (GSFC).

[3]http://fits.gsfc.nasa.gov/

A basic software package is provided together with the data (the Solar Software Tree, *SSW*[4]) to allow scientists to process the data. These routines are written in IDL (Interactive Data Language[5]), a proprietary interpreted language and run-time environment commonly used in solar astrophysics. Most scientists working on RHESSI data are expected to develop their own algorithms using this basic package.

2.2 HEDC

HEDC has been built to optimize the scientific return of the RHESSI mission. Its main goals are to facilitate access to the data, to support data processing and imaging, and to serve as a platform for sharing scientific results. A system like HEDC is needed in addition to the significant efforts associated with RHESSI spacecraft since the astrophysical data is typically prepared with a view towards optimizing ease of production, storage and delivery rather than the *processing* of the data. The main concern in HEDC is therefore to speed up the process whereby scientists extract useful information from the data delivered by the satellite. As a result, most of HEDC's functionality is geared towards the efficient creation, management and utilization of *derived data* or *data products* rather than the management of the raw data produced by the spacecraft. When the raw data units reach HEDC, they are once more searched for interesting events, using programs that detect a wider range of events such as solar flares, gamma ray bursts, or quiet periods. Whenever these programs locate a potential event, several dozen analysis algorithms process the associated data. The analysis algorithms most frequently used in HEDC are imaging, lightcurves and spectroscopy, all of which generate pictoral content. Together with extensive meta data (algorithm parameters, log files) these pictures are cataloged and stored in HEDC to become part of the *extended catalog*.

HEDC has been in development since 1998. It is online since May 2000, when it started to manage data from ground-based observatories. With the launch of RHESSI, HEDC became available to solar astrophysicists, professional astronomers, and the interested public worldwide. Towards the end of the planned RHESSI mission (two years), the raw data volume will rise to a minimum of 1.5 TB. It is expected, nevertheless, that RHESSI will remain operational for three additional years. In this case, about 5 TB of raw data are produced during the spacecraft's life. The pre-processed version of this raw data will amount to several hundred GB. The extended catalog and data products generated by users will amount to an additional 3 TB. Currently, more than 320 GB of calibrated RHESSI data containing 174000 catalog entries and around 25 GB of measurements taken by the Phoenix-2 Broadband Spectrometer in Bleien, Switzerland are available at HEDC. The Phoenix catalog contains spectrograms for around 3000 identified solar events and is part of the extended catalog.

2.3 HEDC Architecture

HEDC follows a 3-tier architecture (Figure 1) to provide as much flexibility as possible in terms of scalability (most scientific repositories use a 2-tier approach [5]). The core of HEDC currently runs on a SUN Enterprise Server with 2 GB RAM, dual 450 MHz processors, a SUN A1000 1.0 TB

[4]http://www.lmsal.com/solarsoft/index_old.html
[5]http://www.rsinc.com/idl/

RAID and 2.5 TB of additional hard-disk drives (storage space will grow with time).

The resource management layer includes the data storage and the external processing units in charge of executing analysis and imaging algorithms. The processing units are IDL servers (version 5.4) running on the SUN server. The system supports running additional IDL servers on nodes other than the server, a feature that has been tested but not yet used in practice. Data storage includes a database and a number of storage devices. For the database we use Oracle 8.1.7 Enterprise Edition. Critical data, such as the database redo logs and configuration information, is stored on the A1000 with tape backup. Secondarily generated data is stored on no-backup RAID5. Raw data files are stored on hard-disks with no back-up and are archived on CDs. Remote archives are linked by NFS. A tape archive connected to the conventional backup facilities is also used for storing data files that are not needed on-line.

The application logic tier includes two components that communicate through RMI and HTTP. The Data Management component (DM) controls and optimizes access to the data. It hides specific details like file formats and the specific data type required by analysis programs behind interfaces. Thus high level DM functions are generic and do not require adaptations if, e.g., the underlying file format changes. The Processing Logic component (PL) manages the IDL analysis servers, which run the SSW routines and provide the standard RHESSI analysis algorithms. It also coordinates necessary data transformations as the IDL server provide only rudimentary job control, data management, and error recovery functionality. Both the DM and PL are implemented in Java (Java 1.3/1.4) and run either as stand-alone programs or as servlets in the Web server (Apache 2.0.39 and Tomcat 4.0.4). For performance reasons some database specific code was implemented using PL/SQL and C++.

At the presentation layer, HEDC can be accessed through either a Web based client using a conventional browser[6] or a Java-based client, the StreamCorder[7]. The former is a thin client solution for users who are only interested in browsing existing data or have the necessary access rights to perform data processing at the server. The latter is a fat client solution that we use to optimize client side processing and to cache data on the client side.

3. DESIGN CONSTRAINTS

3.1 Dealing with Change

A data repository that is used for any length of time must face obvious changes as part of the usual evolution of computing environments (in our case new versions of Java and the DBMS; new operating system releases and data encoding libraries; etc.). A scientific data repository must be prepared for a wide range of additional changes. For example, the actual raw data may change, the analysis routines will certainly evolve during the entire life time of the system, and the usage patterns of the system will change as users evolve the focus of their research. At HEDC we already had to accommodate new raw data formats and new data sources (different RHESSI instruments and other sensors all together), some of which require a new database schema.

[6]http://hercules.ethz.ch:8081/hedc/index.html
[7]http://www.hedc.ethz.ch/release/

Another important requirement is version control. A typical assumption about the raw data stored in a scientific repository is that it does not change beyond perhaps a few corrections here and there. Yet with RHESSI, as in many similar instruments, it is to be expected that the raw data will be recalibrated several times. Accordingly, the raw data and all the derived data based on it must be versioned. In addition, data and analysis algorithms need support for lineage tracking. Depending on user requests and capacity, a significant number of the analyses performed for previous versions of the data may have to be recomputed.

Systems that limit themselves to SQL queries as the basis for data search and processing do not face the problem that the analysis routines used will constantly change. Some may change as a result of data recalibration, others will change simply because designers optimize existing routines, many more will appear as scientists gain experience with the data and devise new ways to extract information from it. It was this form of constant change, experienced since we started to design the system, that led to the inclusion of the PL component as part of the architecture. The PL abstracts the analysis routines and permits changes to them without affecting the rest of the system.

HEDC is capable of dealing with change thanks to a number of programming and user interfaces that abstract from the domain specific data model and processing aspects. Some of these interfaces are most obvious in the organization of the database schema and the implementation of the application logic. The database schema has been divided into two parts: one related to the management of meta data and another with the actual data model for RHESSI. The two parts are independent of each other and it is straightforward to change the RHESSI specific part of the schema. Similarily, the DM abstracts from the actual database schema and the intricacies of optimized SQL, while the PL encapsulates IDL and the SSW routines. Thus the application logic provides clean and generic interfaces that allow changing the underlying infrastructure without affecting the rest of the system. With these solutions, we minimize downtime and the number of places that must be adapted over time, while at the same time maximizing the autonomy of the various system components with respect to each other. Since it is operational, HEDC has successfully undergone several recalibrations, accommodated changes in the format of the raw data, introduced new forms of data processing (e.g., producing video animation rather than just still images), and extended its support for other applications (mainly non-solar events).

3.2 Dealing With the Unexpected

The biggest challenge when building a scientific data repository is that, more often than not, it is not entirely clear how the data will be used. RHESSI data is but one example of this. RHESSI is primarily intended for solar observation. One could, therefore, construct a very successful (and much more efficient) system by focusing on a single use of the data: the study of solar flares. By doing this, it is possible to reduce the amount of data in the system significantly (e.g., dropping all periods when the satellite does not observe the sun, quiet periods, or transits through the South Atlantic Anomaly). It is also possible to optimize querying, processing, and data representations by considering only flare data.

The problem with this approach is that such a system has

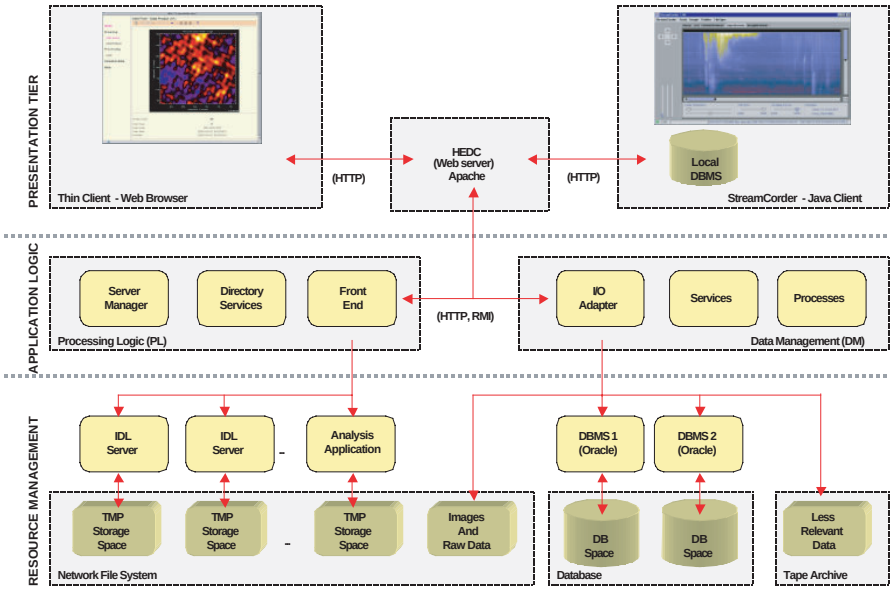

Figure 1: 3-Tier system architecture of HEDC.

the potential of significantly reducing the scientific return of the mission. RHESSI is also a good example of what could be lost. Given the characteristics of the detectors, some of the measurements taken by RHESSI can be used in research related to gamma ray bursts (which are non solar events). In a "solar flare only" system, such research would be impossible and would require direct manipulation of the raw data completely outside the data repository. As pointed out by Gray and Szalay [13], if posing a question in a scientific data repository is difficult and takes time to get an answer, the result is that fewer questions are posed. If a system does not allow at all to pose certain questions, then it limits outright the potential outcome of the data gathering effort.

With HEDC we have tried very hard to provide a completely open system, that is, one in which no question is ruled out from the beginning. This has direct and significant implications on the architecture of the system. First, we cannot solely rely on querying as the way to find relevant data. Second, the time to respond to a query has only a minimal influence in the overall response time observed by a scientist.

3.3 Processing Instead of Querying

SQL is a more efficient way to find out relevant information than ad hoc methods such as scripts that comb through the data files [13]. Unfortunately, this holds only for those cases in which the information the scientist is looking for is properly represented in the database schema. A "solar flare only" repository cannot answer questions about gamma ray bursts and vice versa. The problem is that it is hopeless to attempt to create a super-schema that contemplates all possible uses of the data. This is especially true in repositories that contain continuous observations.

As a result, a scientific data repository that needs to support unexpected analyses needs to use a very flexible data organization, one that does not cast in stone what scientists can and cannot do with the data. Note that the problem is not one of schema evolution. Users of the system may (and will) come up with their own way of processing the data. They will use generic tools, if available, for coarse searches, but afterwards they will perform their own analysis. Such a variety of data views cannot be efficiently supported by any scientific data repository.

The solution we propose as part of HEDC involves two steps, one related to data organization, the other to data processing. In terms of data organization, HEDC does not provide predefined "types" for the data (e.g., solar flare, quiet period, etc). In HEDC there are only *events*. An event is an observation period that has some meaning to a particular user. The notion of event allows users to build their own catalogs of relevant data using any information available in the raw data. By maintaining a basic and an extended catalog, HEDC provides lists of events that are generally accepted as being of a particular type. These lists are generated when the data is loaded. In terms of data processing, HEDC incorporates external programs as part of its repertoire of operations. There is also the possibility for users to submit analysis routines that can be included into the system and made available to other users.

3.4 A Meaningful Notion of Response Time

An interesting requirement imposed on HEDC is that it should significantly reduce the time to obtain meaningful results from the available data. RHESSI data, like in many similar repositories, is at a very low level. Simply stated, it is a list of photon impacts on the detectors, with an energy and a time tag attached to each record. A long chain of processing steps that, in most cases, cannot be automated is taken before a scientist can decide that some records contain meaningful data. This chain typically involves identifying an observation period, producing a particular image for that time period and a given energy range, and deciding whether the data is relevant. The catch is that the analysis and imaging routines can be used in many different ways and with many different parameters. Before a sensible decision can be made, dozens of analyses might be necessary. Thus the amount of data a scientist can explore depends on the overall time it takes until the analysis results may be viewed.

This notion of response time is important because it includes processing time beyond the time it takes the system to retrieve the data necessary for the analysis. This is not a database design problem but an overall system design problem, since the time span to get to the data is significantly shorter than the time needed for actual processing. In HEDC we have implemented a novel solution that shortens this *holistic* response time by at least an order of magnitude (in fact, allowing interactive work with the system which would otherwise be impossible). The approach is based on using approximated data for analysis and visualization and it involves preprocessing the data when it is loaded into the system to construct wavelet compressed range partitioned views over the raw data.

3.5 Avoiding Redundant Work

HEDC is primarily a platform to facilitate the collaboration among scientists. Users can log on into the system to browse and down load data, apply transformations to the data and have the results stored in HEDC. When the user decides, these results are made available to other users as part of the HEDC data collection. This facility is currently being used to disseminate the results of particularly relevant analyses. Since these analyses are already precomputed, users do not need to repeat the analyses themselves, thereby reducing the system load and speeding up the search for relevant data. When a user requests to analyze a given data set, HEDC can check whether this has already been done and, if that is the case, offer the available results as an alternative.

4. DATA MANAGEMENT

4.1 Data Organization

From a system's perspective the data is organized into two categories: meta data and data. By data, which resides in a file system, we refer to the raw data and most derived data products (mainly images). By meta data, stored in a database, we refer to the data that describes, classifies, and summarizes the actual data. The meta data also includes information necessary to manage and control the system. Within HEDC, the data in the file system is only accessible through the meta data available in the database. In this way we can maintain referential integrity and implement proper backup/recovery procedures for the file system.

All file data is read only. Raw data is introduced in the system by uploading routines and stored as FITS files. Some parts of the raw data are wavelet encoded for later use during progressive processing and visualization. The images of the basic and extended catalogs are also stored as files when new RHESSI data is uploaded into the system. Additional derived data is produced either by users directly, by automatic search routines that comb the raw data as a background process, or by users who upload derived data produced with the StreamCorder.

Importing an analysis involves (i) storing and referencing multiple files (algorithm parameters, process log, resulting images, and sometimes the actual algorithm) and (ii) creating multiple meta data tuples. From the user's point of view, this meta data is organized into results of *analyses* (ANA), *high level events* (HLE), and *catalogs*. A HLE roughly corresponds to a period of time and range of energy that has been determined to be relevant by a specific user. For each HLE, there can be many analyses performed. HLE tuples are not only generated during data loading, but also automatically created during local and remote data processing. HLEs can be grouped into catalogs. We use the notion of catalog to implement private user workspaces and to group HLEs according to different criteria (solar flares, gamma ray bursts, flares of certain characteristics, etc.). Examples are the standard and the extended catalog, which are maintained for the benefit of HEDC users as reference points to start their own exploration.

Each HLE and ANA is represented in HEDC by a tuple. These tuples contain enough information to describe events as well as analyses (around 25 and 45 attributes each). They also contain a reference to the files associated with that event or with that analysis. The meta data needed to manage external these file references is stable over time, whereas the meta data describing analyses and HLEs changes as new analysis algorithms are developed. The database schema is therefore divided into two parts, a generic and a domain specific (RHESSI related) part.

The generic part contains three sections. The *administrative* section (three tables) includes all configuration parameters necessary for setting up and using the repository: schema description (used to track the lineage of attributes); available services (type, location, prerequisites); connected clients (type, IP, status); predefined queries and reports; current database instances and data partitions; data refresh and purging rules; user and user group profiles (access rights, sessions, status); etc. The *operational* section (four tables) includes data collected during the operation of the repository: logs and messages; lineage of migrated or transformed data; status of archives (online, capacity left, type); monitoring information such as usage statistics or audit trails. The *location* section (four tables) stores external file references. These tables keep track of the physical location of a file (file path) and provide the information used to access the file directly (e.g., download URL) or after some transformation (e.g., download of a compressed file). Subsection 4.3 explains this name mapping in detail.

The domain specific part (seven tables) contains the analysis, HLE and catalog data. All tables contain references (i) to the location tables, so that files may be associated with any of the tuples, and (ii) to the user table (administration section), so that access rights are enforceable.

4.2 LOBs versus File System

An alternative solution to the one followed in HEDC is to use the "Large Object" (LOB) data type supported by many commercial databases. After extensive testing, we decided against this option for several reasons. First, accessing a LOB is significantly slower than accessing a file. For the LOBs to be manageable, they must be reasonably small. But then requests for, e.g., a long range spectrogram, would have incurred unacceptable delays while retrieving the necessary raw data (which, on top of that, needs to be processed to produce an image). Second, existing implementations of LOBs tend to lack support for the hierarchical storage management systems needed to provide vendor independent, scalable, and robust data access, migration and backup across different file systems and platforms, all of them key features in any realistic data repository.

Another alternative would be to use a file system exten-

sion of a DBMS (e.g., DB2 DataLinks[8]). The problem is that the processing done at HEDC is external. Any form of analysis would require extracting the data through an SQL interface, merging it outside the database, passing it on to the IDL servers and putting the result back into the system through the SQL interface. With our approach, we skip most of these steps so that components and clients simply copy files to the appropriate location.

4.3 Name Mapping

HEDC uses a dynamic mapping scheme to locate and retrieve data items. Information is located by constructing a *name* that refers to the data. This is done through the location tables. Each name has the form: [type] [root] [path] [item id], each one of these elements being determined dynamically for every request. There are three types of names: *filenames*, *tuple identifiers*, and URLs. Filenames describe the local storage location. Tuple identifiers are used to locate tuples (independent from DBMS location or type). URLs are used to download data.

For any tuple in the specific part of the schema (be it an HLE, a catalog, or an ANA), a particular field contains an item identifier ([item id]). The [root] information is obtained from the system configuration files. Querying the location tables with the item identifier returns a number of entries associated with that data tuple. From each entry, the system extracts the name type ([type]) and the archive id. The archive id is used to retrieve the current archive type and [path]. By combining all these elements, the system can therefore obtain the location of the files associated with a given tuple. The same applies for tuple identifiers and URLs.

The cost of this dynamic name construction is two extra database queries on an indexed field. The advantages of the approach are many. On one hand, the name construction process amounts to using *dynamic binding*. This setup provides a great deal of flexibility in managing the file system and allows system administrators to change the location of files arbitrarily without having to modify all tuples in the specific part of the schema (it is enough to modify the location tables). In addition, it can all be done at run time without having to halt the system. System administrators can install or repair disks, reorganize the data, or move data from disk to tapes by simply changing tuples in the location table.

4.4 Consistency

An obvious problem when dividing the system into a database and a file system is how to maintain consistency between the two. In our experience, the problem is significantly simplified by the fact that it is generally possible to run the system on dedicated servers with restricted access. This prevents users or applications from making changes in one part of the system (e.g., deleting a file) without updating the other part (e.g., removing the corresponding tuples).

In HEDC we ensure consistency through the data management part of the application logic layer (see next section). The DM access control component filters unwanted calls and offers transactional properties around *entities* (such as an HLE and its related analysis tuples) and their references to data files during data loading, updates, and migration. It also acts as a coordinator for external programs called during data staging, backup/restore and data processing.

[8]http://www-3.ibm.com/software/data/db2/datalinks/

5. APPLICATION MANAGEMENT

A key feature in HEDC is the integration of data management with external data analysis tools. The goal is to create the illusion of a seamless dataspace that scientists may manipulate with their favorite tools [10]. The role of the middle tier in HEDC is to incorporate external processing and data management tools so that changes in these environments require only localized adaptations in the middleware and are transparent to the user.

5.1 Structure of the PL Component

The goal of the processing logic (PL) is to hide external processing environments behind an interface that the rest of the system can use to request external processing. The PL component is organized around a software framework that implements the following services:

- *Frontend*: Primary controller of sessions and requests, dispatch and scheduling of requests to processing subsystems. There is one instance of this service.
- *IDL server manager*: Multiple native IDL interpreters are managed (start, stop, restart). It provides the possibility to invoke IDL routines synchronously and asynchronously and implements error handling (timeout, resource drain). Every processing client executes one instance of this service.
- *Global directory*: Provides a directory of all services related to the processing logic. There is one instance of this service.

Interactions between these services are self-recovering and tolerate failure and restart. IDL server managers can be dynamically added and removed as needed without halting the system.

To maintain generality, the PL does not incorporate specific information about a processing environment into the interface of a service and the foundation classes of the software framework that the services are built upon. Such information is exchanged in dynamic structures, their interpretation being delegated to framework extensions, mostly *strategy* classes [12]. As before, this represents a layer of indirection that could potentially slow down the system but provides much needed flexibility to incorporate a variety of processing environments.

The PL accepts requests through various interfaces, e.g., the command line, HTTP, or RMI. Regardless of the interface, an analysis follows an abstract model that describes the workflow of an individual request along 4 phases:

- *Estimation*: This is an optional phase that determines the feasibility and availability of resources for a request. We use a simple predictor to inform the user about the duration of the subsequent execution phase. The result of this phase is an execution plan. This phase returns immediately.
- *Execution*: Carries out the actual processing. This phase can be executed synchronously or asynchronously.
- *Delivery*: Results are made available.
- *Commit*: Results are written back into HEDC (through the DM component).

Phases must be executed in order, and not all phases are mandatory. Requests can be canceled at any time and induce the cleanup for the current phase.

Specific request types feature analyses that are implemented as a set of strategies, i.e., one for each phase. The strategy concept supports our demand for flexibility, and the common structure of requests simplifies the implementation of the front end. Incorporating new processing environments into HEDC involves defining the strategy that extends the existing framework as needed. In this model, the front end takes the role of an *interpreter of abstract requests*, and the execution of requests (execution and subsequent phases) is launched according to a priority scheduling.

5.2 Structure of the DM Component

The DM has a layered architecture with two internal interfaces, both of which are also public to client programs. The purpose of this architecture is mainly to improve the adaptability of the system to the constant changes in all aspects of data management and processing, while at the same time offering client programs a stable API to operations, services and processes.

The *I/O layer* abstracts from the actual storage type and location. All data accesses happen through this layer. It manages database access, file system manipulation, database connections and performs general resource management. *Operations* like dynamic name construction are also done at this layer. Adapters encapsulate the formats of the various storage types. The database adapter, e.g., translates incoming data requests into proper SQL queries. The layer supports dynamic partitioning of the load so that, e.g., data requests for certain parts of a database schema are routed to a different DBMS. We use this feature to separate processing from browsing clients.

The intermediate *semantic layer* is used to implement *services* and interact with the Application Integration component. It enforces access rules, ensures referential consistency, and determines data dependencies. For instance, for a given analysis request, reading and writing files in a FITS format is implemented by the I/O layer. The semantic layer, however, is in charge of determining which type of processing is performed and how the results of the analysis are grouped together. This layer ensures that all images produced during an analysis are properly referenced in the system. The semantic layer also implements services such as data insertion and deletion. The analysis service is called, e.g., to initiate a spectrogram analysis for a given set of raw data files, or to call the algorithm that extracts meta data from an RHESSI file.

The *process layer* combines the operations of the I/O layer with the services of the semantic layer to provide *processes*. One such process defines, e.g., the workflow during physical archive relocation. First, tuples referenced or referencing an entity are queried and altered, then the corresponding files are copied, compensating actions are taken if failures occur, and finally logs are generated. Other processes implement raw data preparation, event filtering, entity association, and catalog generation.

5.3 Sessions and Constraints

Each request to the DM contains user authentication to retrieve the associated user profile (user rights, configuration, constraints), identify the use, and build a temporary view (to speed up subsequent data access). Profile, status information and view are stored in *sessions*.

Depending on user authentication, user rights and system setup, processes might also apply constraints. *Privacy constraints* guarantee that only public data may be read or processed by other users. *Access constraints* may enable DBMS queries but disable edits for specific data and specific user groups. *Integrity constraints* ensure that the application specific rules are followed, e.g., referential integrity, so that tuples belonging to an entity may not be deleted if data dependencies exist.

Creating database connections and user sessions are the two most expensive parts of request processing. To improve performance, we have implemented pools for both in the DM. The database connection pool is split into separate pools for query processing, updates, and user authentication. Connections are immediately released by sessions after the result set has been copied. The DM caches up to three sessions per user (one for analysis, HLEs, and catalogues each). The cache lookup algorithm uses the network IP and cookies to match clients with their sessions.

5.4 DM Call Redirection

The system has been designed to run either on a single node, or distributed across a cluster. In the same way that the I/O layer of the DM component supports data partition, there is the possibility of redirecting calls from one DM component to another. We use this feature to increase capacity in HEDC by adding more nodes to the system (see the experiments).

The DM API is accessible locally through direct instantiation and pool managers. Local configuration files determine if calls execute locally or remote. In general, the calling methods do not know where the code is actually executed, but can use overwrites to, e.g., force local execution. Furthermore, a DM might decide to place a request in an execution queue, send the request to a pool of worker threads for asynchronous execution or execute the call directly.

The DM API has no provisions for regular SQL calls. It uses Java collection objects instead. During query processing these objects are parsed, analyzed, verified and transformed into regular SQL queries suitable for the target database and schema. As a result, queries may be adapted and optimized without system downtime. Furthermore, no adaptations to the API are needed to accommodate configuration changes, such as newly materialized views or changed tables.

5.5 Access Control

By default all derived data belongs to the user who creates it and is considered private. Only the owner may change or delete private data. For data to be visible to other users, the owner must flag that data as public. HEDC's catalogs, e.g., contain tuples created by an *import* user, and are later made public. All tuples in the specific part of the database contain the key of the tuple owner. The system typically appends the user id to all queries so that only public tuples or tuples owned by that user are returned.

HEDC requires an account to access its more advanced features. Non authorized users may only browse public data. Depending on their user profile, authorized users may in addition download, analyse and upload data. Every system component (such as Web server servlets or the PL) must identify itself to the DM, which also must send a valid

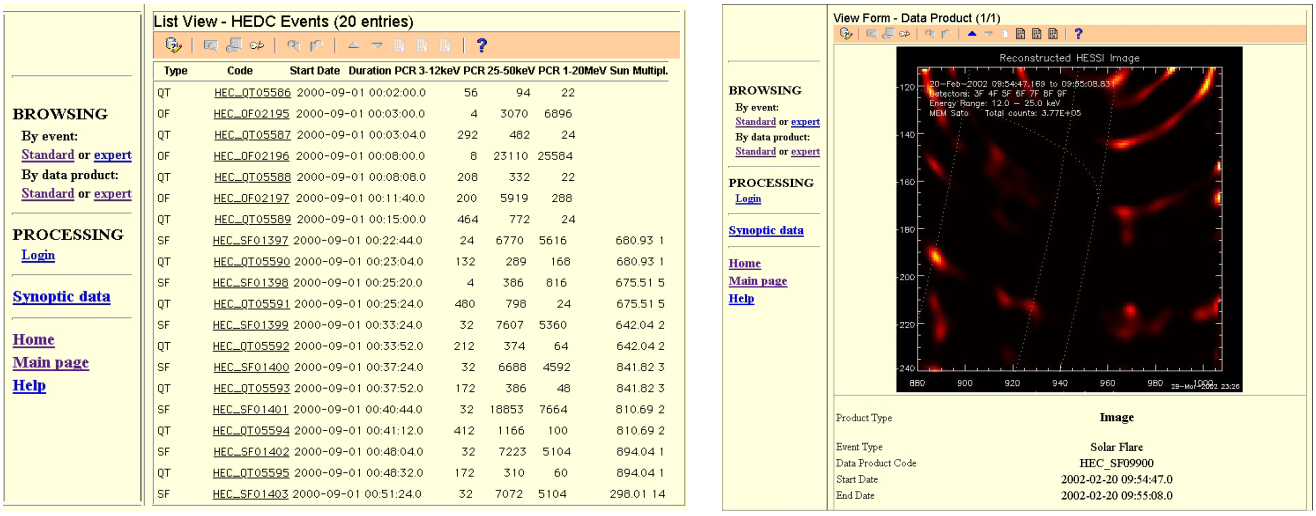

Figure 2: The Web interface supports browsing of catalogs (left), HLEs and analyses (right).

user/password combination to every database it connects to. As DM and database accounts are not related and use different access methods, users have no possibility to connect directly to the database.

6. WORKING WITH HEDC

6.1 Web Interface

The basic interface to HEDC is through a Web browser. The pages needed for this purpose are generated by the DM component. Note that a response may involve a combination of multiple HTML template files, which are populated during query processing. Each template contains dynamic and static images, Java Script, CSS style sheets and plain text. A request to display an HLE, e.g., involves loading and filling in HLE header/footer templates and an analysis template for each ANA tuple associated with that HLE. For such a request, the DM issues on average seven database queries, building the response HTML page from the tuples returned by the database.

When a request arrives, system status and user context are considered in deciding where to execute it (if remotely, the execution takes place using the redirection feature of the DM component). System status includes, e.g., the current vertical partition that determines what DBMS a query is routed to. The user context consists out of access rules, which determine if users are allowed to see and/or edit all committed data (similar to a super-user) or are restricted to their private tuples (normal user).

The Web interface supports browsing of catalogs, HLEs and analysis through links. Thus, the user may query for certain HLEs contained in a catalog, jump to all analysis contained in one HLE, then query for similar analyses and jump to display all associated HLEs. Furthermore, links are available for data download, analysis and upload. On the analysis pages users may enter the parameters and execute routines on the server to produce, e.g., ligthcurves, images and spectrograms.

6.2 StreamCorder

The StreamCorder is a fat Java client (Figure 3) offering the same functionality as the HEDC Web-interface, plus additional features that would have been very difficult to implement using an HTML/applet approach. The StreamCorder architecture is similar to the one of the HEDC. The functionality is divided between basic services and dynamically loadable modules (or *cordlets*). Core services include job- and resource-management, request queues and interfaces to local analysis programs. To increase performance, some libraries were implemented in C++. Modules are data-type sensitive, in the sense that the StreamCorder offers different modules to the user depending on the context. The context is determined by the data type of the view or analysis in question and kept across all modules.

The Web-based HEDC client relies on the caching built into the Web-browser, which is limited to static HTML-page elements, such as analysis images, that are stored in the local file system. The StreamCorder offers two caching strategies. The first version caches not only images downloaded during browsing but all large data-objects, including data-segments used in local processing. Cache access is accomplished through a local DM component, which calculates a unique but *static* file system path for each data-object. As this path is based on fixed object attributes, such as type and creation date, the cache structure is predetermined.

The second version adds a local DBMS installation for *dynamic* object references and meta data caching. With both a DM and a DBMS present locally, cache object-retrieval and -placement is identical to the way the server DM handles the server-side data archives during data-loading and query-processing. As the schema used locally is the same as the one on the server, every installation of the StreamCorder is, in fact, a clone of the HEDC server extended with a GUI and extra services.

6.3 Approximated Analysis and Visualization

Many queries require summary data and use aggregates. Hence, in addition to indices, we use materialized views to improve response time. Our goal is to *minimize the response time* between the submission of a request and the presentation of the analysis results to the user (see [22, 21] for details on these issues).

Our basic approach to make HEDC interactive is to pre-process the raw data. The pre-processing step involves partitioning the data and encoding the data using wavelets.

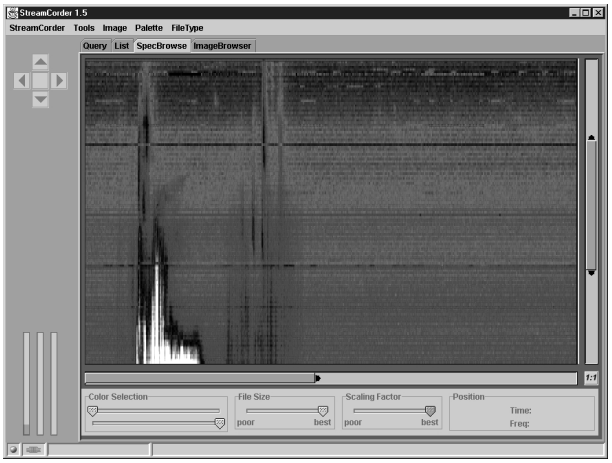

Figure 3: The Streamcorder offers progressive data analysis and visualization.

The wavelet transformation is done in such a way as to allow the data processing routines to work on a fraction of the original data. At the client side, these coefficients are used to reconstruct an approximated view of the original data set and this view is fed to the analysis routines. Since the time complexity of many important analysis routines is directly related to the size of the input data (linear for short analyses and exponential for complex ones), the time it takes to perform such analyses is significantly improved.

Approximated materialized views are also used to allow interactive database visualization. The basic idea is to reorganize the catalogs as a number of multi-dimensional arrays and allow users to specify ranges in any of the dimensions. Based on these ranges the information is then presented in a compact and efficient manner using *density* (number of tuples per bin) and *extent* (location and extent of each tuple or cluster of tuples) plots. To achieve the necessary performance for interactive use, we also implemented several important optimizations. First, the arrays are pre-processed and sorted according to the most relevant attributes. Then they are partitioned across the dimensions to form the equivalent of materialized views. Since the partitioned views tend to be large, we encode them using a wavelet transformation. Decoding takes place at the Java client side to minimize the load at the server (otherwise interactive exploration would require a very powerful server). To further speed up the generation of the plots, the client works on approximated and aggregated versions of the original data. In both cases the StreamCorder coordinates the download, caching, decoding and analysis of the data. Users access and manipulate an analysis or visualization result without any knowledge about how and where the data is stored and processed.

6.4 Integration: Synoptic search

The synoptic search subsystem serves to locate synoptic data in remote repositories. The query mechanism is context dependent so that users can find information correlated to what they are seeing in HEDC. The approach followed resembles a Web-crawler. First, online requests are issued to several remote archives in parallel. Then the results are collected, grouped and displayed to the user. Currently, the only search criterion is the observation time.

This service operates independently from other subsystems of HEDC. The service is best effort (if a query to a remote archive times out, no results are available); query results are not cached, and there is no data synchronization between HEDC and the remote archives. This light-weight approach of rendering synoptic data accessible through HEDC has proved to be practical and robust, avoiding the issues of consistency and data synchronization with remote archives. In its current configuration, six popular remote archives are searched, including the SOHO synoptic data archive[26].

7. EVALUATION: WEB BROWSING

The current version of HEDC supports a limited amount of simultaneous Web users and is meant to run on low-budget hardware. Nevertheless, the system is designed to scale. The Web browsing performance depends on several variables: server hardware and operating system, server software, network speed, and workload. The purpose of this evaluation is to (i) determine the number of simultaneous Web clients sending continuous requests until the request rate matches the capacity of the server (maximum load), (ii) observe the system behavior once that threshold has been passed and more clients are being added (degradation behavior), and (iii) measure the increasing maximum load as more DM components running on separate processing servers are added (scalability). For an evaluation of the Java client see [21].

7.1 Test Environment

Tests are performed on a development testbed consisting of a SUN Enterprise 3000 database server (SUN OS 5.6, two 450 MHz CPUs, 512 MB RAM, 3.5 TB RAID), one to five Web servers (RedHat 7.1, dual Pentium III 1GHz CPUs, 1 GB RAM), and up to 96 dedicated client workstations (RedHat 7.1, Pentium III 1GHz CPU, 256 MB RAM). During the tests, an increasing number of Web clients send requests to one of the five Web servers, which in turn send requests to the DM that generates the response pages. Database, Web server, and client computers are connected by a switched 100 Mb/s Ethernet.

The responses are queries to an Oracle 8.1.7 Enterprise server installation holding more than 100,000 tuples for each queried table. All database queries are performed on indexed fields. More than 32,000 images are available during the tests. To avoid contention during test buildup, the Web servers are configured with a large number of initial child processes. The database connection pool sizes are large enough to guarantee free connections throughout the tests.

7.2 Measurements

The chosen requests are representative of how users browse the HEDC. A typical sequence of requests by a single user first sends a query to select an HLE, then sends another query to retrieve all its related analyses, and finally sends requests for all images related to these analyses. For everyone of these requests the Web server servlets generate a response page. Every response page entails multiple database queries, parsing the query results, adding a number of static images (e.g., for browser navigation), and wrapping everything in HTML.

During testing the clients randomly read requests from a local list, try to establish a connection to a Web server (which in turn passes the connection request on to the appropriate servlet), send their request, read the servlet re-

sponse, render the response page, and then start all over again. On average, a request generates seven DM queries and requires parsing of 80 tuples. Two of these queries warrant a full index scan and two are count queries. The average response size is 12 KB for the response HTML page and 35 KB for the embedded dynamic images. Static images are downloaded only once and then cached on the client side. We use persistent connections (as defined by HTTP 1.1) in the experiments. The number of "Keep Alive" requests per connection is set to unlimited on the Web server.

During each test, clients send their requests to a single Web server. If multiple servers are used, the client requests are spread evenly. All clients choose requests randomly from the same list. Measured are the response time of the system, from the moment the client issues the request until the moment the page has been rendered. Session objects are cached on the server side. Thus, every client must authenticate itself only once (authentication requires one DBMS query and one update).

The purpose of these experiments is to determine maximum server throughput for defined server setups. Therefore, the individual client workload is chosen so that local bottlenecks could not distort server performance. To minimize swapping, the client workload is chosen to fit into RAM (together with the relevant processing routines and operating system modules). All log information is stored on local disk to avoid networked I/O operations. Only requests with low HTML response page complexity are included in the tests, so that rendering on the client is much faster than generating the dynamic requests on the Web server side. The delay between requests is set to zero so that these experiments represent the worst case possible (as in real life, users first view the request response before a new request is issued).

7.3 System Throughput

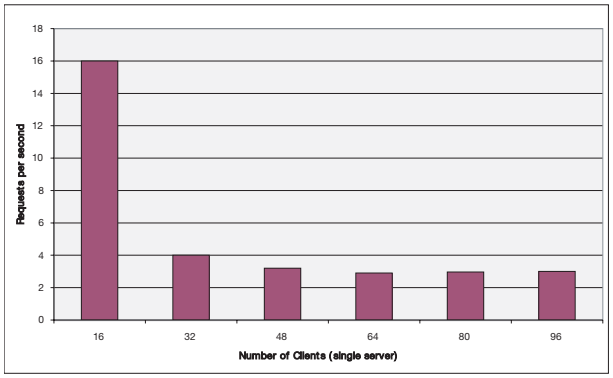

Figure 4: Browse throughput versus number of clients (single middle tier server).

The client workloads are carefully chosen so that a small number of clients (around 16) are sufficient to drive our system with a single Web server at full capacity (see Figure 4). This is due to the underlying database, which supports a maximum throughut of around 120 HEDC request per second. In the tests, the clients skip the visualization step. With normal clients (involving a human who looks at the results of the analyses) the peak performance of the system is reached with about 60 simultaneous clients (which, given the existing user community, is in fact larger that the typical number of concurrent clients we expect at any one time in HEDC). Hence, by using clients that only submit requests without looking at the results, the experiments are a comprehensive stress-test of the key areas that impact browsing performance at HEDC: the DM, DBMS, and file system. The areas covered by these tests include (i) integration: request forwarding to DM, name mapping, archive access, intra middle tier communication, DBMS queries; (ii) security: authorization, constraints, consistency checks; (iii) sessions: transactions (update session attributes), caching, database connections.

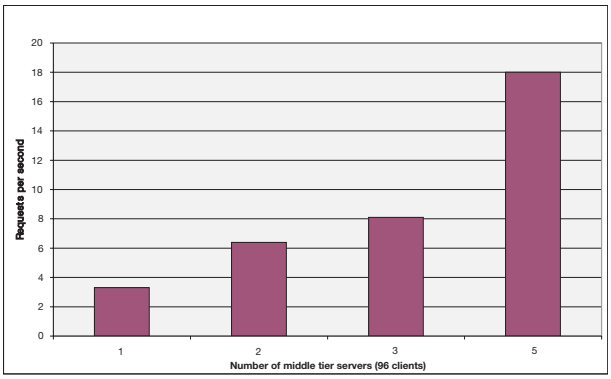

Figure 5: Browse throughput versus number of middle tier servers.

At 16 test clients, the database is running close to its maximum performance, with roughly one complex Web request per second per client (which translates to about 100 database queries per second). If more simultaneous Web clients are added, the overall throughput drops to around 3 requests per second at 96 clients (see Figure 4). As the number of simultaneously used database connections (and thus the number of issued queries) also decreases, the database is not the reason for the slowdown. Instead, the drop in performance is caused by the increased processing load of the application logic. To verify this hypothesis, we conduct another experiment where the application logic is spread across multiple computer nodes. Figure 5 shows the overall throughput versus the number of middle tier nodes for 96 simultaneous test clients. As more nodes process the Web requests arriving at the application logic, the throughput rises from 3 requests for one node to 18 requests for five nodes. These 18 requests result in around 120 HEDC database queries, the peak performance of the database setup for this type of range query.

This experiment shows that HEDC can scale to handle more concurrent clients by simply running several copies of the application logic in different nodes. Further scalability can be achieved by replicating the database using standard techniques. Or we could add another DBMS on a different node and apply the partition facilities provided by the DM component. Then performance would be determined by the CPU capacity, a scalability problem that can be solved by replicating the application logic as shown in the experiment.

8. EVALUATION: PROCESSING

8.1 Test environment

We conducted two series of tests that approximate the application and system load that we expect for HEDC. A 2×177MHz SUN SPARC machine is used as server for data

	Image				Histogram				
Processing on	S	S	C	S+C	S	S	C	C/Cached	S+C
Concurrent analyses	1	2	1	2+1	1	2	1	1	2+1
Overall duration [s]	6027	3117	2059	1380	960	655	841	821	438
Turnover [GB/day]	0.8	1.5	2.3	3.5	4.6	6.8	5.3	5.4	10.0
Avg. sojourn time [s]	109	56	37	24	115	74	98	90	40
Avg. sys CPU server [%]	2	3	2	4	9	13	5	6	17
Avg. usr CPU server [%]	50	96	3	95	49	76	10	11	70
Avg. sys CPU client [%]	-	-	5	4	-	-	5	6	5
Avg. user CPU client [%]	-	-	90	85	-	-	29	30	28

Table 1: Performance of the imaging test (Legend: S=Server, C=Client).

and processing, one 400MHz Linux PC acts as a client for processing that accesses data remotely from the server. The HTTP bandwidth between client and server is 2 MB/s. The input for all tests are 50 MB of raw data partitioned into 50 files. Each test consists of a series of requests, each of which computes an analysis from an individual file. The requests are submitted successively such that no more than 20 requests are in the system at any given time. Each test is executed on various system configurations to study the effects of combining or separating data management and processing tasks. The configurations differ in the number of analysis tasks that execute concurrently on the server and processing client.

8.2 Imaging Tests

The first test series comprises 100 requests for images (see Table 2). Imaging is a typical CPU-intensive analysis, the computation of an image takes about 20 s on an input data set of 800 KB on the processing client, and 60 s on the server. Only a fraction of the input data is actually accessed. The query/edit figures show the total number of interactions with the data management subsystem during the test. Each analysis involves 3 queries and 2 edit operations. Table 1 (left) summarizes the results.

Requests	100
Input [MB]	50 (50 files total, 2-3 per analysis)
Output [MB]	5.5 (100 GIFs)
Queries	300
Edits	200

Table 2: Characteristics of the imaging test.

8.3 Histogram Tests

The second test series comprises 150 requests for histograms. This type of analysis is more I/O-intensive than imaging. The net computation of a histogram takes about 2–3 s per 300 KB input data on the processing client and 5–7 s on the server. The characteristics of the histogram test are summarized in Table 3.

Requests	150
Input [MB]	50 (50 files total, 1/3 per analysis)
Output [MB]	1.2 (150 GIFs)
Queries	450
Edits	300

Table 3: Characteristics of the histogram test.

Table 1 (right) summarizes the results. For this test, we measure an additional configuration called 'client/cached', where input data resides already on the scratch space of the processing client. Compared to non-cached operation, the figures demonstrate that even for the data intensive histogram test, the cost of data movement are relatively small.

8.4 Integrating new Processing Environments

The duration of query and edit operations is almost constant and equal in all scenarios; the net processing time (i.e., sojourn time - waiting time) is equal for individual requests and increases slightly with higher CPU load.

The results indicate that the primary system load stems from the analysis routines and the corresponding coordination; the cost of data movement and management are comparably low. The scenarios and applications we consider tend to be CPU-bound. In a configuration with a central data repository, however, I/O could become a limiting factor with more concurrency and distribution in processing.

The overhead of the abstraction layer represented by the application logic is generally low compared to the overall resource requirements of our target applications. For analyses with computations longer than 5 s, the interaction frequency between data management, processing logic and processing subsystems is low; the overhead per request is negligible. In scenarios with parallel computations of analyses shorter than 5 s, the central scheduling in combination with the fault tolerant protocol among the services becomes critical: jobs are not scheduled timely to available resources (Table 1, right: the client CPU is not saturated).

The integration of new applications and proprietary subsystems is feasible but requires enhancements across all layers of our architecture. During the development of HEDC, we were confronted with minor daily updates and several major updates of the SSW software. We experienced that our design absorbed such changes well and that updates to the software interface could be handled more smoothly than changes in data formats.

9. RELATED WORK

An important point to keep in mind when considering the functionality of HEDC is that it provides access to the raw data obtained by the observation instruments. Typically, scientific data repositories provide access only to derived data products (e.g., images). This makes a big difference in terms of requirements and the challenges faced when building the system. For instance, repositories with only derived data do not need that much flexibility in terms of processing.

With this difference in mind, there are a number of astrophysical data repositories available on line, such as AstroWeb[9] which currently holds data from 143 other repositories. However, the complexities due to changing formats,

[9]http://cdsweb.u-strasbg.fr/astroweb.html

different user expectations, and evolving software imply that numerous existing repositories of scientific data focus on a stable core functionality and provide access via FTP with Perl scripts and an HTML user interface [8, 7, 14, 20]. Some interfaces are more complex, such as [15] or Aladin [1], the latter offering a Java applet to query multiple XML repositories and to visualize, refine and superimpose images. A small number of archives actually use a DBMS as a query processing tool and apply database optimization techniques [6, 9, 15, 19, 18].

Most modern sites offer browsing and download of their raw data and (some) of their data products. Many sites store very large amounts of raw data [23], yet in contrast to HEDC, usually not on disk and not on line. User management, access rights, etc., tend to be managed on an ad hoc, manual way, e.g., by email. Many sites offer access to a mixture of related remote data, data references or data products [1, 2, 3, 4, 7]. In contrast, HEDC not only gives access to the raw data but also allows users to create data products and to store analysis data products in the system.

There are also several projects for sharing scientific experiments (e.g., [3, 16, 25]). Few of these projects automate the detection of overlapping requests and it is the user's responsibility to find out whether a particular type of search or analysis has already been performed. A novel aspect of HEDC is that it provides a platform to automate the steps and that it provides users with tools for offline work.

10. CONCLUSIONS

This paper outlines the challenges and requirements encountered during the design and implementation of HEDC. Our design choices will be valuable in many other scientific repositories as they address common difficult problems that plague such systems. The contribution of HEDC lies, therefore, both in the set of features and services provided to the (application) scientist and in the specific system organization that makes these features practical. Due to the separation of meta data extraction from the generic data handling and repository management, our methodology may be used for any of the complex data formats encountered in science. As the experiments demonstrate, the design of HEDC allows scientists to construct a powerful repository using minimal hardware. If needed for performance or scalability, the very same system can be transparently extended to run in a cluster configuration where different components run in different nodes. HEDC can be made to grow with the data and the user needs.

Another important characteristic of HEDC is that it incorporates external processing into the system transparently to the user. It does so without imposing constraints on the nature of the processing routines. HEDC tolerates software evolution and changes to the routines and the processing environment. Similarly, HEDC has been designed to cope with constant change in the underlying data (format, calibration), usage patterns (queries, analysis routines, data collections), and derived data products stored in the system (usually the result of new analysis routines).

Supporting external processing and data sharing opens the door to novel interaction schemes. A scientific data warehouse, even if hosting a huge data collection, can be organized as a set of collaborating systems. As every StreamCorder is in reality a fully functional server, requests may also be sent to peer clients to allow peer to peer interaction.

Acknowledgments

We thank A. Benz, P. Saint-Hilaire, and A. Csillaghy for their help in dealing with the scientific aspects of the RHESSI data and for their contributions to the design of HEDC. We are also grateful to the entire RHESSI team at UC Berkeley and the NASA Goddard Space Flight Center.

11. REFERENCES

[1] Aladin. http://aladin.u-strasbg.fr/java/.

[2] Astrobrowse. http://heasarc.gsfc.nasa.gov/ab/.

[3] Virtual Obs. http://www.eso.org/projects/avo/.

[4] ADS. http://adswww.harvard.edu/.

[5] Banerjee, S. A DBS Platform for Bioinformatics. In *VLDB, Cairo, Egypt* (Sept. 2000), pp. 705–710.

[6] Barclay, T., Gray, J., and Slutz, D. Microsoft TerraServer: a spatial data warehouse. In *SIGMOD, Dallas, USA* (2000).

[7] CIO. http://ircatalog.gsfc.nasa.gov/.

[8] CDA. http://cdsarc.u-strasbg.fr/.

[9] DSP. http://www.cacr.caltech.edu/digital_sky.html.

[10] Frew, J. Data management for earth science systems. *Sigmod Record 26*, 1 (1997), 27–31.

[11] Freytag, J. (Panel Chair) The future home of data. In *VLDB, Hong-Kong, China* (Aug. 2002).

[12] Gamma, E. et al. *Design Patterns*. Addison Wesley, 1995.

[13] Gray, J., and Szalay, A. The world wide telescope. In *CACM. Vol. 45, No. 11* (Nov. 2002), pp. 50–54.

[14] Hubble Space Telescope. http://hubble.nasa.gov/.

[15] Infrared Space Obs. http://www.iso.vilspa.esa.es/.

[16] Kaestle, G., Shek, E., and Dao, S. Sharing experiences from scientific experiments. In *SSDBM, Cleveland, USA* (1999).

[17] Saint-Hilaire, P., et al. The RHESSI Experimental Data Center. In *Solar Physics* 210(1-2), pp. 143–164 (Dec. 2002).

[18] Sheikholeslami, G., et al. WaveCluster: A wavelet based clustering approach for spatial data. *VLDB Journal 8*, 3-4 (2000), 289–304.

[19] Sloan Digital Sky Survey. http://www.sdss.org/.

[20] Stoesser, G., et al. The EMBL nucleotide sequence database. *Nuclear Acids Research 27*, 1 (1999), 18–24.

[21] Stolte, E., and Alonso, G. Efficient exploration of large scientific databases. In *VLDB, Hong Kong, China* (Aug 2002), pp. 622–633.

[22] Stolte, E., and Alonso, G. Optimizing scientific databases for Client-Side proccessing. In *EDBT, Prague, Czech Republic* (Mar 2002), pp. 390–408.

[23] Szalay, A. S., et al. Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. In *SIGMOD, Dallas, USA* (2000).

[24] Tsur, S. Data Mining in the Bioinformatics Domain. In *VLDB, Cairo, Egypt* (Sept. 2000), pp. 711–714.

[25] Wang, J. T.-L., et al. Pattern matching and pattern discovery in scientific, program, and document databases. In *SIGMOD, San Jose, USA* (1995), p. 487.

[26] Zarro, D. SOHO synoptic database. http://sohowww.nascom.nasa.gov.