

MIGRATING A MULTITERABYTE ARCHIVE FROM OBJECT TO RELATIONAL DATABASES

A commercial, object-oriented database engine with custom tools for data-mining the multiterabyte Sloan Digital Sky Survey archive did not meet its performance objectives. We describe the problems, technical issues, and process of migrating this large data set project to relational database technology.

The advent of digital archives—enabled by quantum leaps in the technology to publish, distribute, and mine data over the Internet—has created a science and engineering data avalanche. The Human Genome Project (www.ornl.gov/TechResources/Human_Genome/home.html) and the CERN Large Hadron Collider (<http://lhc-new-homepage.web.cern.ch/lhc-new-homepage>) are two examples of very large scientific data sets coming online in the biological and particle physics communities, respectively. Astronomy is no exception, and soon might be deluged with more new data from current and proposed sky surveys than any other science. In this age of multiterabyte scientific archives, scientists need to share common lessons from different disciplines to make the most of available opportunities and to

avoid being overwhelmed by the flood of data.

The Sloan Digital Sky Survey (SDSS) is a multi-institution project to map about half of the northern sky in five ultraviolet-to-infrared wavelength bands (www.sdss.org). Projected for completion in 2005, the survey should image over 200 million objects and collect spectra (redshifts) for the brightest one million galaxies among the objects. A detailed description of the SDSS project appeared in a previous issue of *CiSE*.¹ To learn more about the SDSS science archive (SA) discussed in detail here, go to www.sdss.jhu.edu.

SDSS will revolutionize astronomy in many ways, but most significantly, it will dwarf current astronomical databases. We expect raw data to exceed 40 Tbytes, with the resulting archive available for scientific research (object catalog) being 2 to 3 Tbytes. The survey's information content will be larger than all the text in the US Library of Congress.

In 1995, we chose a commercial object-oriented database management system (OODBMS, or OODB, for short) for our data repository. The leading OODBs seemed to offer significant advantages for the anticipated SDSS SA data model and application. OODBs have a larger set of available data types and the ability to use object associations to traverse references (links) between objects instead of the expensive table joins used in relational database management system (RDBMS, hereafter, RDB) models.

1521-9615/03/\$17.00 © 2003 IEEE
Published by the IEEE CS and AIP

ANI THAKAR AND ALEX SZALAY

Johns Hopkins University

PETER KUNSZT

CERN

JIM GRAY

Microsoft Research

Glossary

CSV—Comma-separated value
DDL—Data-Definition Language
EDR—Early data release
FITS—Flexible Image Transport System, a binary data exchange format used extensively by astronomers
OID—Object identifier
OODB—Short for OODBMS, object-oriented database management system
OpDB—Operational database in which we deposit raw SDSS data.
OQL—Object Query Language
QET—Query execution tree
QSO—Quasi-stellar object, or quasar
RDB—Short for RDBMS, relational database management system
SA—SDSS science archive
SAQA—Science archive query agent
SDSS—Sloan Digital Sky Survey
SXQL—Science Archive Extended Query Language

We based our OODB product selection primarily on its performance, transparent and configurable data organization, and binary compatibility across a range of platforms. But, in spite of these anticipated advantages, we began to have problems with the OODB's performance, vendor support, and missing features as data volume increased. After we could no longer meet our user community's demands, we decided to migrate our data to an RDB. This article details that process.

SA Overview

The SDSS data is collected at Apache Point Observatory in New Mexico. This raw data is first calibrated and fed to a pipeline code at Fermi National Laboratory that performs several operations on it before it is stored in the OpDB (OA in Figure 1). After further processing, "survey-quality" data is exported to the master science archive (MSA). The total data is expected to be a few terabytes in size. It is accessible to the entire astronomical community through specialized tools, and to the public at large through the Internet. The SA is replicated at SDSS member institutions, with the master archive residing at FermiLab. Details of the data-acquisition process, the software pipeline, and URLs for the latest public data release can be found at the main SDSS Web site www.sdss.org.

Data Products

The SA contains several distinct data products, including photometric and spectral catalogs, a redshift catalog, images, spectra, and maps. We optimized the SA's data model for fast catalog access; users can access other data products indirectly.

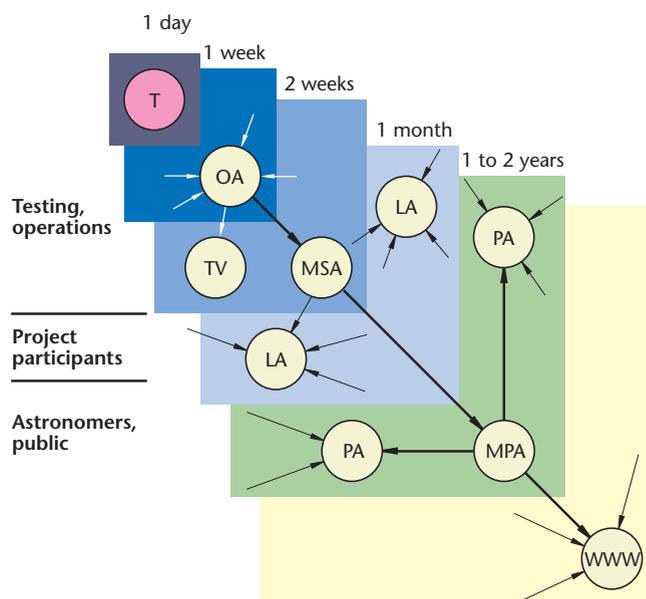


Figure 1. Sloan Digital Sky Survey (SDSS) dataflow diagram. After initial testing, the telescope (T) data first will reside in an operational archive (OA), and be stored in a tape vault (TV). Then, it will be calibrated and stored in a master science archive (MSA) and local archives (LAs), and ultimately replicated as public archives (PAs), via a master public archive (MPA).

Table 1 shows the complete list of data products available through the SA. Figure 1 illustrates the conceptual SDSS dataflow.

User Community

The SA must serve three types of users:

- *Power users.* Sophisticated astronomical com-

Table 1. Data products available through the SDSS science archive (SA).

Data product (as of Data Release 1)	Size	Comments
Photometric Object catalog	800 Gbytes	Parameters of more than 10^8 objects
Spectroscopic Object catalog	150 Gbytes	Parameters of more than 10^6 objects
Atlas images	2 Tbytes	Five-color cutouts of more than 10^8 objects
Spectra	100 Gbytes	In a one-dimensional form
Derived catalogs	100 Gbytes	Clusters, quasi-stellar object absorption lines
4×4 pixel all-sky map	100 Gbytes	Heavily compressed

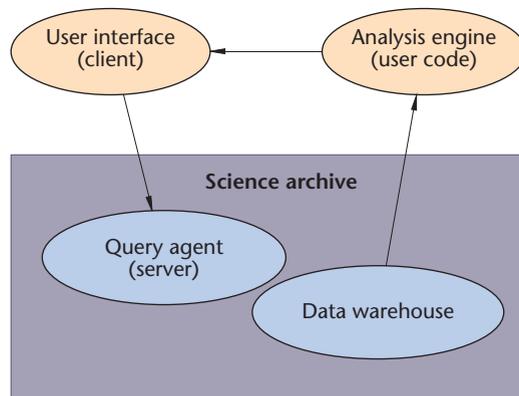


Figure 2. Science archive (SA) client-server architecture. Users can submit queries to the SA query agent (server) with a GUI client Sloan Digital Sky Survey query tool (sdssQT) and direct output from queries to preexisting analysis engines.

munity users, with lots of resources. Their research is centered on archive data, likely with a moderate number of very intensive queries, but mostly statistical queries with large output sizes.

- *General astronomy public.* Frequent but casual lookup of objects and regions. The archives will help their research, but probably won't be central to it. There will be a large number of small queries and cross-identification requests.
- *Wide public.* Browsing a “virtual telescope” and using the data in education (the SkyServer education projects received over one million Web hits in the past year as students used the site to do 150 hours of online training). This access has surprisingly large appeal, which produced a large number of requests. The SkyServer is the most popular Web site hosted at the FermiLab.

The SA OODB Implementation

In this section, we describe the implementation of the SA on a commercial OODB, and the software

and tools we developed to convert it into an advanced data-mining facility. We also outline the challenges we faced once our implementation was completed.

SA User Access

Figure 2 shows the SA's primary functional units from a user's perspective. The astronomical community can access the SA's full data and query facilities via a user account (username and password) and a client-server interface to the actual database. Users begin by starting up the SA GUI—the SDSS query tool, *sdssQT*, which is a portable Tool Command Language and GUI toolkit (Tcl/Tk) client application that communicates with the SA query agent (SAQA) over a TCP/IP connection.

The *sdssQT* lets users open multiple, simultaneous sessions with different SA servers and submit multiple parallel queries to each server. *sdssQT* queries can direct their output back to the GUI (default), to a file, or to an analysis tool through a different socket connection. A proprietary binary output option provides compact output.

Query Processing

An intelligent SAQA query server manages user sessions, processes (parses, optimizes, analyzes, and executes) user queries, extracts user-requested individual object attributes, and routes query output to the user-specified target.^{3,4} The query agent uses several modules to perform its tasks. We describe the agent architecture and the modules in detail next.

Query Analysis

The SAQA analyzes each query and provides a query “cost” estimate in terms of the database subset (number of database files and containers) that will be searched and a time approximation required to complete the search. The user decides whether the query is worth running based on its scope and the time required to search that scope.

We compute the projected query cost by building a *query tree* and intersecting it with a preconstructed *multidimensional spatial index*—the Hier-

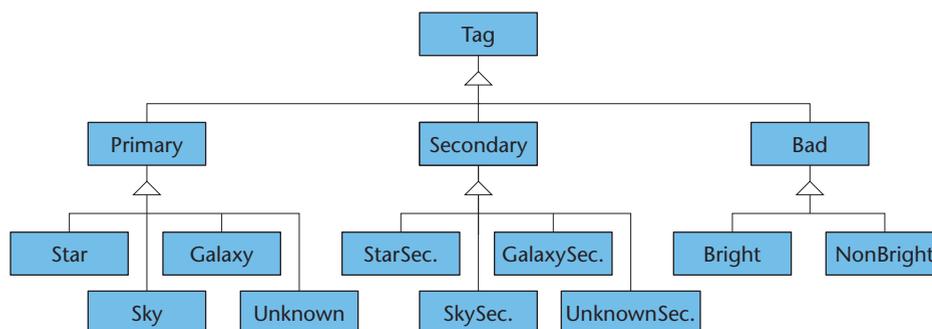


Figure 3. A hierarchy of tag classes narrows searches to the chosen object's subclass. Data organization is such that a subclass search is much faster because only the database container corresponding to that subclass is searched.

archical Triangular Mesh (HTM).^{5,6} The intersection yields the query's scope in terms of the number of database files and containers the query will touch. An entire (generally distributed) database is referred to as a *federation* in OODB terminology, and each federation consists of many individual database files. Each database is further subdivided hierarchically into *containers*, *pages*, and *slots*. The query tree, along with the scope obtained from the intersection, yields a *query execution tree* (QET), which is the user's parsed query in tree form with scope information included in it. The query engine executes the execution tree to return query-selected objects.

The SA Query Language

The SA Query Language—SDSS Extended Query Language (SXQL)—is an SQL-like language that implements the clauses and functions necessary to formulate queries for an astronomy object database. We did not attempt to be Object Query Language (OQL) compliant, although we borrowed some concepts from OQL. SXQL recognizes the standard SQL `SELECT-FROM-WHERE`, including nested `SELECT` statements. It further allows users to specify *association* links in the `SELECT`, `FROM`, and `WHERE` subclauses. Associations are links to other objects, and can be to-one (unary) or to-many (*n*-ary) links. SXQL also recognizes a proximity query syntax, which lets users search for all objects close to a given object in space. Finally, SXQL contains several astronomy-specific macros (for example, `RA()`, `DEC()`, `GL()`, and `GB()`), and complete mathematical function support (for example, trig functions, `SQRT`, `LOG`, `EXP`, and so on).

Tag Objects

We defined object classes to speed up query

searches. The *tag* class encapsulates all the data attributes indexed and requested most often. Included are spatial coordinates, fluxes (the recorded measures of object brightness in each wavelength band), and photometry flags status. The tag objects improve database search speed in the following ways:

- *Indexed lookup.* Encapsulating the most popular data attributes means an indexed lookup can speed up the majority of searches. We worked hard to develop multidimensional spatial and flux indexes^{5,6} specifically for this purpose.
- *Caching.* Tag objects' small size ensures that many can load into the cache, which speeds up I/O.
- *Specialization.* We further divided tag into several subclasses, as Figure 3 shows. Each tag subclass contains exactly the same data members, but the class hierarchy narrows searches on tag objects because all the tags reside in separate containers in each database, and different subclasses of tags reside in different containers. Hence, when a user searches galaxy tag objects, only the galaxy container in each database file will be searched.

SAQA Multithreading

The SAQA's software incorporates multithreading³ at several levels (see Figure 4). At the top, a continuously running main thread monitors a socket to accept new user connections. Each new user session spawns two I/O threads, one each from—to the GUI. Searches on remote partitions in a distributed federation execute in parallel remotely via multithreaded remote slave servers (Slave). Figure 5 illustrates the query agent's distributed operation.

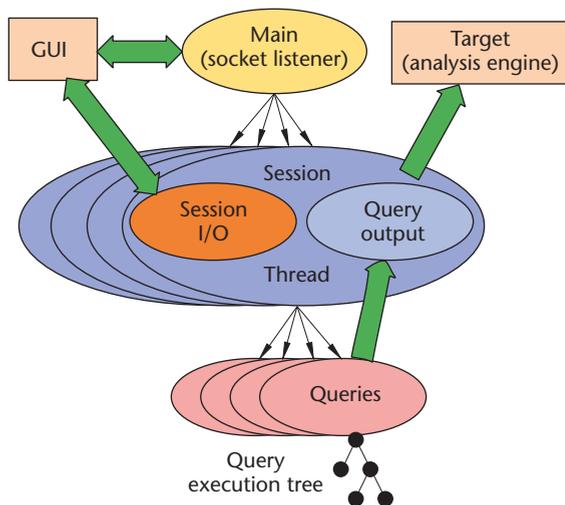


Figure 4. Server multithreading in the query agent. Each user session is in a separate thread, with a separate thread handling each query.

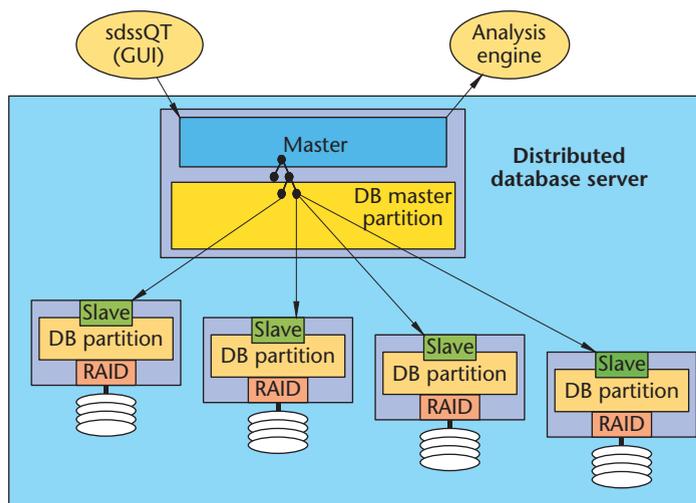


Figure 5. Distributed implementation of the science archive (SA) query agent. Remote slave servers execute parallel searches on remote database partitions.

Query Engine

The engine library module implements a multi-threaded query engine, which executes the SXQL query and returns a bag of pointers to the selected objects as output. A bag is a generalization of a set in that duplicate members are allowed. An extractor module then chooses selected members from each object in the bag.

The query engine's input is the QET. Each QET node is either a query or a set operation. Operations are `union`, `intersection`, `difference`, and `distinct` (which converts a bag

to a set). The `union` operation is n -ary, `intersection` and `difference` are binary, and `distinct` is unary.

We used the OODB's predicate query facility to perform the SA query primitives, although we could not use the OODB's predicate-matching and had to develop our own predicate facility, as described later. Each QET node executes in a separate thread, and an ASAP data-push strategy (data is pushed up the tree using stacks as soon as it becomes available from child nodes) ensures rapid response even for long queries.

QET

Figure 6 shows an example of a QET. The QET's nodes are the SA query primitives and set operations, which map onto the OODB query primitives. Each node returns a bag of object pointers to its parent. All nodes have bags as inputs and outputs, except the leaves, which only have an output bag because they operate directly on a database scope. The different types of query nodes (query primitives) are

- *scoped*, which uses the OODB's scoped query functionality. A subset of the federation (scope) is specified for the search in the form of a node list. The predicate is applied only to the selected type of objects. The scoped query is the only type of query primitive that creates a bag of objects from scratch, hence, it is a leaf node in the QET. All other query types operate on at least one bag, and they form the QET's interior nodes.
- *bag*, which applies the given predicate to each object in the bag that it obtains from its single child node.
- *association*, which selects all objects linked through an association-link to each object in the bag obtained from its single child node. The given predicate is then applied to the selected objects.
- *proximity*, which searches for all objects that are nearby in space to a given object. Such a query is very useful and common in astronomy.

Parser, Intersector, and Partition Map

Several other modules play important roles in the SA machinery. Figure 7 shows the SA software's detailed architecture, which includes client (GUI), query support (server), and data warehouse layers. The parser module parses an SXQL query and converts it to a query tree, which is then passed to the intersector. The intersector intersects the query tree with the spatial and flux indexes.^{5,6} The parti-

tion map tells the query agent how the data is distributed in the federation by identifying partitions on local and remote partitions. This partition knowledge enables parallel data searches on different partitions.

Abstract Library

The abstract library module provides a runtime abstraction of the SA data model. It allows manipulation of database objects without knowledge of schema, retrieval of data values, or invocation of methods (if applicable). In effect, it behaves as a runtime metadata server, or type manager. The abstract can

- specify an object's class using a name or alias,
- translate a name or alias to an actual C++ class, and
- identify a particular member of a persistent class and retrieve the following information about it: the kind of member (basic data, array, function, association link), type of value returned by member (`int`, `float`, `string`, and so on), size of array or object member, and input and output formats, and then return a pointer to the actual object in memory.

The parser, intersector, query engine, and extractor modules use the abstract library. In addition, any application or module that accesses the data model can use the abstract library.

Extractor

The extractor library module extracts individual or groups of attributes (members) of a given object; it executes the `SELECT ...` part of an SXQL statement. The members that it can extract include data, function, and association-link members. The extractor retrieves data values based on information provided by the abstract and includes the required mechanisms for invocation of object methods.

Port Daemon

The port daemon ensures that the GUI and analysis engine communicate with the server on the correct port. It also performs process-level authentication, because firewalls should open only for the port daemon's own port and the predetermined ports configured for the server process.

Data Loader

The data loader is part of the data warehouse layer, and is the application that loads new data into the OODB federation. The server opens the federation

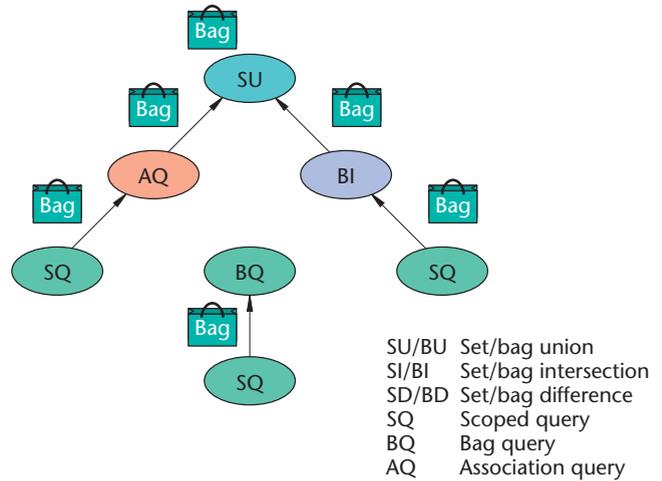


Figure 6. A query execution tree (QET) example. Processing starts at the leaf nodes (scoped queries) and filters up the branches to the root. The internal nodes are association queries (AQs), bag queries (BQs), or set operation nodes (union, intersection, difference). The operand at each level above the leaf is a bag of object identifiers (OIDs).

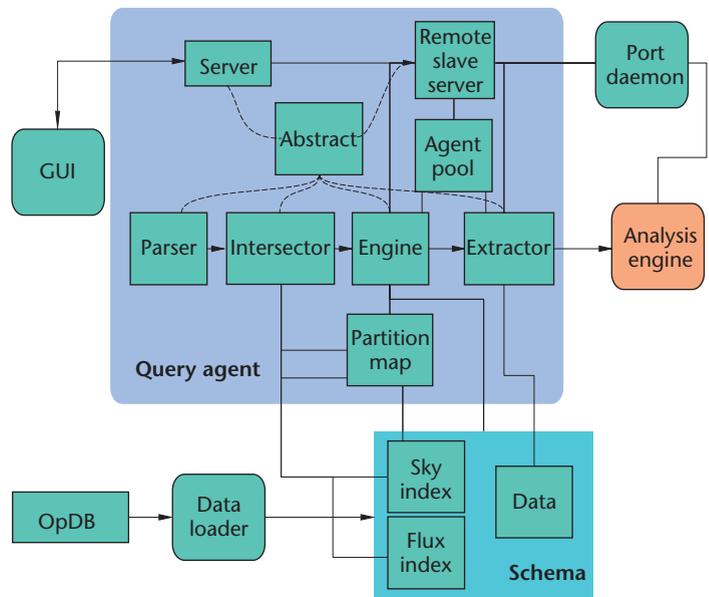


Figure 7. Science archive (SA) detailed architecture. The components in the large box comprise the query agent, which does all the query processing and manages user sessions. The GUI is the user client where queries are submitted, the port daemon routes users to the proper server address, and the analysis engines are user-developed postprocessing applications. The sky and flux indices are the multidimensional indices that speed up spatial and flux queries. We developed all the software components shown here with the exception of the analysis engines.

in read-only mode, and the OODB addresses concurrency issues. The incoming data format is hard-coded into the data loader module to prevent data mismatches. The latest version of the data loader also incorporates the ability to store different calibrations (versions) of the same data that need to be accessible through the archive.

OODB Evaluation

Now that we have described SA deployment on the OODB, it is time to evaluate the OODB's features and performance and how those factors influenced our decision to migrate to an RDB.

Advantages

Early in the planning of the SA, we decided that an OODB offered significant advantages over a relational, or table-based, system. In addition to the conceptual attractiveness of a DBMS that could store the data in the same way astronomers think about it, several other considerations made the choice of an OODB a beneficial one:

- *The ability to link objects to each other.* An OODB allows one-to-one, one-to-many, and many-to-many links between objects.
- *Transparent and configurable organization of data on disk.* The OODB has a hierarchical data organization with the data federation at the top, followed by database, container, page, and slot. Each level in the hierarchy can contain up to approximately 65,000 (2^{16}) objects of the next level. Thus, each database can contain approximately 65,000 containers, and so on. More importantly, each database is a disk file, and a database administrator can configure the placement of objects into containers.
- *Binary data compatibility across platforms.* OODB database and federation files can move across heterogeneous platforms without compatibility issues. This facilitates implementation and maintenance of distributed archives.
- *Object methods.* We expected that the OODB would let us define class methods that would reside in the database along with the object data. We thought of these as stored procedures, but on a per-class basis. This feature is, however, still not available in the latest version of the OODB, although we were able to provide macros and object functions via our query language.
- *Query tools.* The OODB community was developing OQL and promising nonprocedural access to the data in a standard query language.
- *Development tools.* The OODB products had

good tools that dovetailed with our chosen development environment based on C++.

- *Administrative tools.* The OODB products had a complete set of utilities for data definition, data security, and data replication.

We considered the first four features crucial to facilitate data-mining performance and make it possible to run efficient queries for “needle in a haystack” types of searches.

In spite of these advantages, even during the survey-commissioning phase we recognized that the SDSS would be using the OODB in an untested regime. Other application domains that used the OODB did not have the data-mining features and query performance that the SDSS user community would demand. These problems worsened as our user community became more sophisticated and familiar with the query language.

Inadequate Query Language

The OQL supplied with the product fell far short of what was promised 10 years ago. The current version of the OODB supports only Boolean user-defined operators—that is, the only value returned by the operator is either true or false. These are too restrictive. It would be efficient to have more versatile operators that can perform arbitrary complex tasks and return values of arbitrary type. Query predicates do not support operators for manipulating individual bits of data members. This functionality would be very useful to us given our extensive use of bit lists for our indexing. Furthermore, the ability to directly invoke member functions of persistent classes would let us store analysis and processing code in the archive itself instead of running these methods manually after retrieving the objects. In general, the query language was inadequate, so we had to build our own, with the incumbent implementation and execution costs.

Bugs

We found several serious bugs during the three years of using the OODB that were never resolved—we had to devise workarounds for each of them at considerable development cost to us. The most critical bug that we encountered was incorrect array addressing of single-precision (`float32`) floating-point array members in query predicates. For example, if f is a floating-point array and a user asks for $f[n]$ (the n -th element of f), the OODB returns $f[2n]$ —that is, the array offset is off by a factor of 2! This happened only with `float32` arrays, not with `float64` arrays, which suggested the prob-

lem was the OODB assumed every floating-point array to be `float 64`.

Fatal errors generated during a query automatically caused the C++ application to abort with a core dump. Defining an error handler did not help because the error handler could not avoid the abort—it could only report that a fatal error had occurred. This was unacceptable because a fatal error caused our entire query agent to crash, throwing off all users logged in at the time. We encountered a situation in which a division in the query predicate caused a fatal error (floating exception) if the value of the object attribute in the denominator happened to be 0 for one of the objects encountered in the search. Although it is reasonable to expect the query to be aborted by such an error, it is ridiculous to have to abort the query agent because of it. There was no way to isolate the error to the thread that generated it and continue with the remaining threads.

The missing features and critical bugs just described made the OODB's predicate-query feature unusable for our application. The lack of reliable predicate-query functionality obviously crippled our ability to make effective queries against the SA, and was a serious blow to our progress with SA development. As a result, we had to write our own predicate functionality, which required several person-months of development effort. Our home-cooked predicate query served us well and even let us implement features not available in the OODB (such as inclusion of object methods in query predicates), but it placed an additional burden on software support and maintenance.

Performance Woes

To serve our customers, we needed to be able to access data at more than 50MBps (which would be an hour for a query that examines the entire photo catalog). No matter how much hardware we threw at it, the OODB could not go much more than 0.5 MBps—100 times too slow for our needs. We eventually traced this to the OODB's use of network file system and the lack of high-speed sequential access to the data at the file system level. There was no way around this without reworking the OODB, something the vendor refused to consider.

Another serious problem was that if indices were defined on several quantities in our tag object class, the order in which the terms appeared in the query predicate had to be the same as the order in which the indices were created; otherwise none of the indices was applied! For example, the predicate,

$(i < 11 \ \&\& \ r < 13 \ \&\& \ z < 9),$

where i , r , and z are members of the tag object, would invoke the index only if the index on i was created first, the index on r was created second, and the index on z was created last.

Administrative Problems

Inevitably, we had to change the database schema, adding attributes or classes, creating associations, and making other schema changes. As often as not, this required a complete database reload (and the online schema change often corrupted the current system).

One thing the OODB community has yet to address is a coherent strategy for evolving the database schema when application programs have such a tight coupling to the data and when the data has object pointers that may be split or merged by a schema change. In addition, the OODB lacked the administration tools typical of more mature database management systems.

In fairness, other projects—notably BaBar (www.slac.stanford.edu/BFROOT/www/Public/Computing/Databases)—succeeded in building atop an OODB, but they did it by writing more than a million lines of code above the interface and by using the OODB as a persistent object store (checkin–checkout data) rather than as a data warehouse with ad hoc access.

Migration to RDB

With the bugs and other issues previously discussed (and additional performance inadequacies that we'll describe later on), we had a difficult time justifying the OODB's use, in spite of its conceptual attractiveness, and the time and effort that we invested in building a distributed data-mining engine on top of it. Apart from the issues relating to the particular database product on which we had deployed the SA, object persistence was proving to be too expensive for us because of poor query language support, the data model's relative inflexibility to frequent changes, a total lack of query optimization, and lastly—but perhaps most debilitating for data mining—inadequate I/O efficiency and support for data striping.

Fortunately, in the last 10 years the database landscape has shifted. RDBs have improved in the following ways:

To serve our customers, we needed to be able to access data at more than 50 MBps. No matter how much hardware we threw at it, the OODB could not go much more than 0.5 MBps—100 times too slow for our needs.

- RDBs have matured as the de facto standard database technology embraced by the computer industry, with wide tools support and much improved quality and functionality in language integration and database extension mechanisms that provide some of the benefits promised by OODBs.
- Fierce competition among the major RDB vendors (for example, Oracle, IBM, and Microsoft) has resulted in superb I/O optimization. Seven years ago, when we decided to adopt an OODB for the SA, object-oriented (OO) database technology outperformed existing relational technology. However, in the last few years, the leading OO database vendors have not kept up with the order-of-magnitude increase in raw I/O performance achievable with redundant array of independent disk (RAID) systems.
- The advent of Web services, SOAP, and XML enables objects on the fly—that is, we can retrieve data in object form irrespective of how the raw data is stored on disk, providing dynamic object interfaces to relational databases and making object persistence unnecessary.

The SkyServer stole the show during the first six months of the EDR. Users quickly realized that this was an easy way to get data out of the EDR.

These developments prompted us to consider migrating the SA to a commercial RDB to significantly improve performance, database administration, and optimization. Our opportunity came with the SDSS SkyServer, which we'll discuss next.

The SDSS SkyServer

We wanted to deploy a version of our data in Microsoft's SQL Server (MS-SQL, for short) to provide an easy-to-use, Web-based option—called SkyServer—for casual users of the soon-to-be-available first public distribution of the SDSS early data release (EDR), which was officially released in June 2001. The choice of Microsoft's products was dictated by the opportunities available—Microsoft was willing to support the project, and one of us (Jim Gray) was intimately familiar with the product. In the absence of that combination, we probably would have used Apache/DB2 on Solaris.

To load the EDR data into SkyServer, we had to modify our data loader module to export data in ASCII comma-separated value (CSV) format

to load into the relational data tables. The original version of the data loader fed data directly into the OODB using the OODB's proprietary Data-Definition Language (DDL). Adding the data loader's CSV export feature was neither difficult nor time-consuming, and we had an MS-SQL EDR database up and running in a matter of weeks. This CSV interface provides a “blood/brain” barrier between the two subsystems and has been invaluable in allowing each component to evolve independently.

We designed the SkyServer to be a user-friendly version of the EDR database, with help and documentation available for novice users, and we promoted it to the lay public and amateur astronomers. Although the interface included an SQL query submission facility, we geared it more toward casual users interested in browsing SDSS data rather than running intensive data-mining queries. We did not expect heavy use of the SQL query page.

We were wrong. The SkyServer stole the show during the first six months of the EDR. Users quickly realized that this was an easy way to get data out of the EDR. Some groups used the interface to crawl the entire site and get a complete extract. Others wrapped the query interface in a collection of Emacs macros and would routinely get personal data extracts. Still others got private copies (we distributed about 40 copies of the full EDR and several hundred copies of the 1 percent subset called the Personal SkyServer.)

The SkyServer's popularity—even with professional astronomers—was because of its versatility, reliability (99.9 percent uptime over the first six months of use), and performance. The experience convinced us that it was a viable alternative as an advanced data-mining facility.

All this contrasted with the persistent and critical problems that we experienced with the object version over the same period, particularly in terms of stability, reliability, and performance. With major data model changes just around the corner, MS-SQL emerged as a far more attractive option.

But this was not a decision to be taken lightly, and we undertook a comparative evaluation of the OODB and MS-SQL with performance benchmarks to formally ascertain whether MS-SQL would meet our demanding needs.

Moving to the Relational Data Model

Our main hurdle was translating our object data model into a relational one. The relational model is “flat;” there are no subclasses. It does not support pointer associations; all associations must be

via key values. We had to convert the hierarchical object data architecture to relational tables. Beyond the loss of the object data model's conceptual ease, the transformation was not difficult. We were able to use views to capture the subclass concept. For example, the Galaxy view is the subset of photo objects that are classified as galaxies and are primary objects. We modeled associations with foreign keys. Data import also was easier partly because we had already written a data loader for our OODB that read the astronomical data in its original flexible image transport system (FITS) format and stuffed it into the OODB using its DDL. As previously discussed, we were able to modify the data loader to export data in a CSV format accepted by MS-SQL for importing files into the relational tables.

We were sorry to lose some of the object data model's benefits:

- *Specifying links between objects with straightforward syntax.* For example, using `"SELECT obj.field ..."` instead of the less-intuitive `JOIN` syntax of relational queries.
- *Array (vector) fields.* Relational tables cannot support arrays. Given that most of our fields are measured in five wavelength bands, it was really convenient to have an array⁵ of each quantity instead of five columns in the table for each quantity measured in all bands.
- *Approachability.* The user community found our simplified SQL less imposing and confusing than standard SQL.

However, the loss of these OODB advantages was not a high price to pay for MS-SQL's considerable advantages.

MS-SQL Advantages

We gained many benefits by migrating to MS-SQL. The following is not a complete list—rather, it describes those advantages that are most relevant to our application regime.

Performance and Stability

MS-SQL query performance on a representative sample of astronomical queries is an order of magnitude superior to the best performance that we could squeeze out of the OODB after all our query engine's optimization and parallelization. Loading performance is much faster with MS-SQL; we can load our current data in several hours rather than several days with the OODB. Subsequently, we have achieved another 10-fold speedup in most of these areas by better design and more modern

hardware. We are nearing data rates of 1 GBps and load rates of 50 MBps.

The MS-SQL RDB's stability also is much better than we experienced with the OODB; uptime is consistently more than 99 percent with the Sky-Server RDB, while we have had daily restarts and downtime with the SA. These OODB failures have been due primarily to the fatal error-handling bug and multithreading issues specific to the SGI platform on which we deployed the OODB. The OODB schema evolution problems and longer loading times also contributed to the downtime.

Query Language

In contrast with the limited SQL functionality that we were able to implement using SXQL, MS-SQL offers full SQL compatibility and the added features of Transact SQL (T-SQL), such as the ability to define variables and temporary tables. MS-SQL contains many features requested by our user community. Unfortunately, we have not been able to implement them in SXQL because of our limited software development budget.

MS-SQL's three most useful features are `GROUP BY`, `ORDER BY`, and `COUNT (*)`. Their absence in SXQL had hampered our development of a suite of test queries for our performance benchmarks because we were unable to translate several test queries—originally developed for a different purpose—into SXQL. SDSS users want to sort, group, and bin query results, which would eliminate a fairly time-consuming data analysis step.

Advanced Query Optimization

Two of MS-SQL's greatest strengths are its optimization ease and its self-optimizing capabilities. In a comparison of major RDB products featured in a *PC Magazine* report,⁷ MS-SQL outscored its competitors in this aspect, although optimization is a core strength of all the major commercial RDB products. A graphical query plan that's viewable before submitting an MS-SQL query provides details on which query steps take the largest fraction of execution time and—in most cases—gives users all the information necessary to improve query performance.

Administration Features

MS-SQL features, such as triggers, stored procedures, and dynamic indices, provide a level of convenience and ease for database administration that we did not have with the OODB. In addition, we can do many optimization tasks, including index management and data reorganization, while the database is online. With the OODB, we found this

problematic enough to be virtually unusable.

Schema Design and Evolution

MS-SQL's built-in data model design tools simplify complex schema development. With the OODB, we had to use a separate software product (Paradigm Plus) to develop a diagrammatic data model before translating it into schema files that the DBMS could understand (the DDL).

The benefits of schema development in MS-SQL crystallized when we contemplated major data model changes to bring detailed information about the SDSS spectroscopic plates' tiling procedure into the OODB. We realized that adding these changes would stretch our existing database to the breaking point.

Schema evolution is another feature that we should have been able to take for granted with a DBMS but have been unable to use in practice with the OODB. Even small schema changes corrupted the existing database data and required a complete data reload.

After several unsuccessful attempts at it—and lacking adequate vendor technical support—we abandoned schema evolution. Instead, we minimized schema changes and always reloaded the entire database when a change was absolutely necessary. This was a significant drain on our limited operations resources because it required several days to implement even a minor schema change. We anticipate that this will not be a problem with MS-SQL because already we have successfully completed several EDR SkyServer schema changes.

Performance Benchmarks

Performance was not the only reason we switched to MS-SQL, but it probably was the most important one. With data volume expected to quadruple over the next three years, we were concerned that all the optimization we had built into our OODB query engine would not be enough if the DBMS itself could not meet certain I/O performance benchmarks. Almost all the queries that had performed poorly on the OODB were I/O-bound.

Before we carried out the performance benchmarks described here, we knew that our SQL server database (the SDSS SkyServer) performed better than the OODB version for certain types of queries. However, we wanted to achieve some objectives by conducting formal benchmarks:

- document that MS-SQL was performing much better with indexed queries;

- ensure that we were giving the OODB a fair shake (we hoped to find that the OODB would perform better with complex queries due to its hierarchical data organization);
- provide a level playing field by running the benchmarks on identical hardware; and
- make every attempt to ensure that the optimization and parallelization features that we built into our OODB server software would be utilized in executing the test queries.

We formulated a set of 25 test queries that ranged from simple index lookups to queries with complex constraints on nonindexed quantities and multiway joins. These queries (available at www.sdss.jhu.edu/sx/pubs and <http://computer.org/cise/cs2003/c5tha.htm>) are a subset of the 35 queries we used to benchmark the SkyServer.⁹

Benchmark Performance Results Analysis

Figure 8 shows the test queries' execution times for three database configurations: MS-SQL, a single-disk OODB (OO-single), and a three-disk partitioned OODB (OO-striped).

On average, MS-SQL is three to 10 times faster than OO-single, and one to three times faster than OO-striped. Figure 9 shows this more clearly, and compares the OO-single system and the OO-striped system execution times to MS-SQL execution times. The OO-striped's three-way data striping yielded a performance improvement of up to a factor of three because of our multithreaded query agent design.

On queries that return very large numbers of objects, the distributed version of the OODB (OO-striped) starts to bog down while extracting requested data fields from matching objects. This is especially evident for tag object queries, because many of the small tag objects load into the cache at the same time. The main thread then must extract the required data fields from the cache, so main-thread CPU use quickly gets into the 80 to 90+ percent range, which limits query execution speed. Figure 10 illustrates the process, and shows the ratio of the times taken to return just an aggregate count of the objects (`with COUNT(*)`) to the total time taken (to return all objects) for each query. The ratio is lowest—that is, the count is much faster than the whole query—when the total number of objects is in the millions. This is a limitation of our distributed-design query agent, which distributes the query search but not the object data extraction.

The joins on the MS-SQL version were fast for all the reasonable queries we formulated. In fact, finding a query that would take more than a few

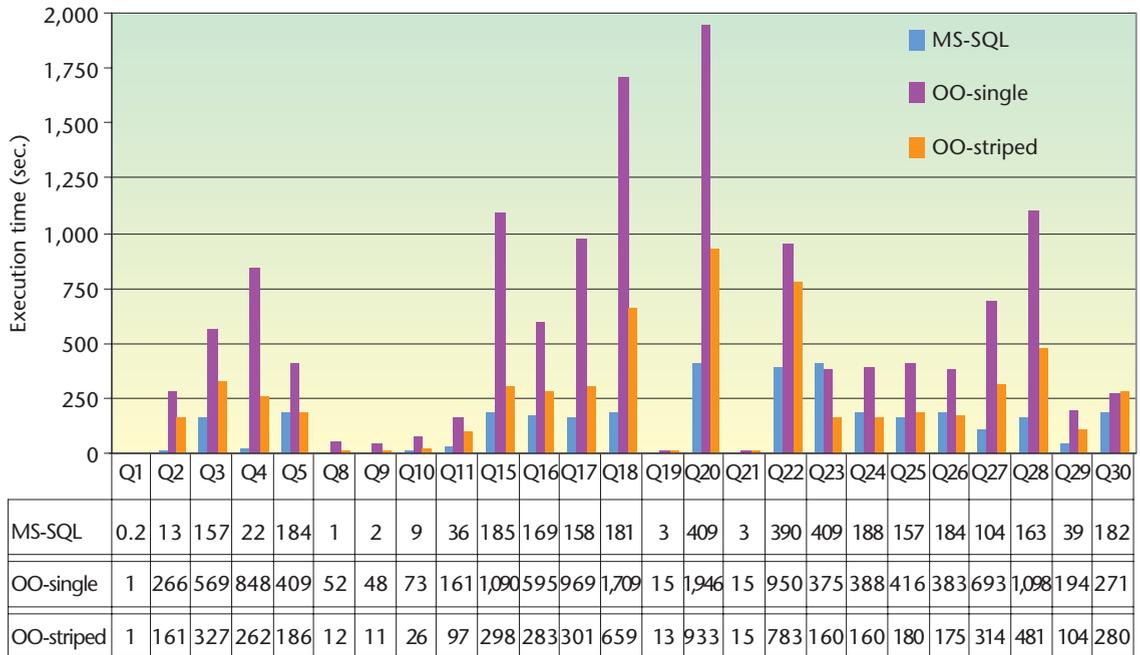


Figure 8. Results of running the 25 test queries listed at www.sdss.jhu.edu and <http://computer.org/cise/cs2003/c5tha.htm> on Microsoft SQL server (MS-SQL), a monolithic object-oriented database management system (OODB) (OO-single), and a data-stripped OODB (OO-stripped).

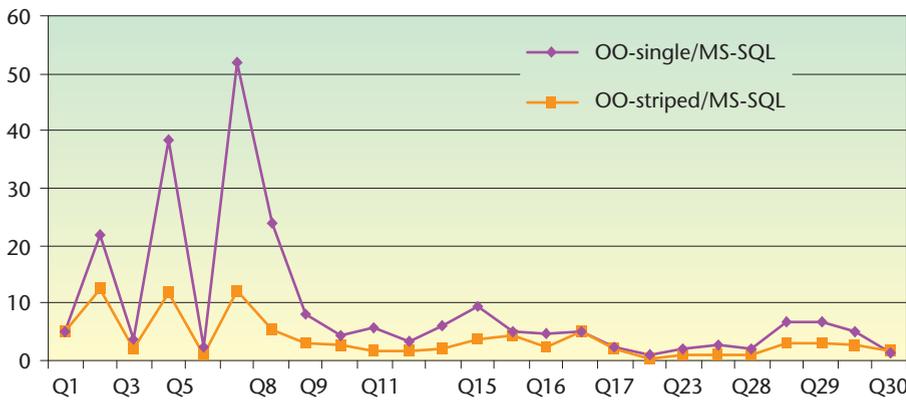


Figure 9. Comparison of query execution times on MS-SQL, with those on the single-disk object (OO-single), object-oriented database management (OODB) system, and the three-disk (OO-stripped) OODB. The graphs show the execution times' ratios plotted for each query.

minutes to execute on the MS-SQL version was hard. The only time the execution was relatively slow was when there was no constraint on an indexed quantity in the query predicate. An example of this situation is in test query Q31, which is a version of Q30 with a much larger limiting value of a nonindexed field in the `photoobj` table. The largest table in our database, `pho-`

`toobj` contains the measured parameters of each object observed by the SDSS telescope's imaging camera. Figure 11 shows the Q31 execution time differences for MS-SQL, OO-single, and OO-stripped. The execution times are almost two orders of magnitude higher, but such a query is rare because users can narrow the search using one of the indexed fields.

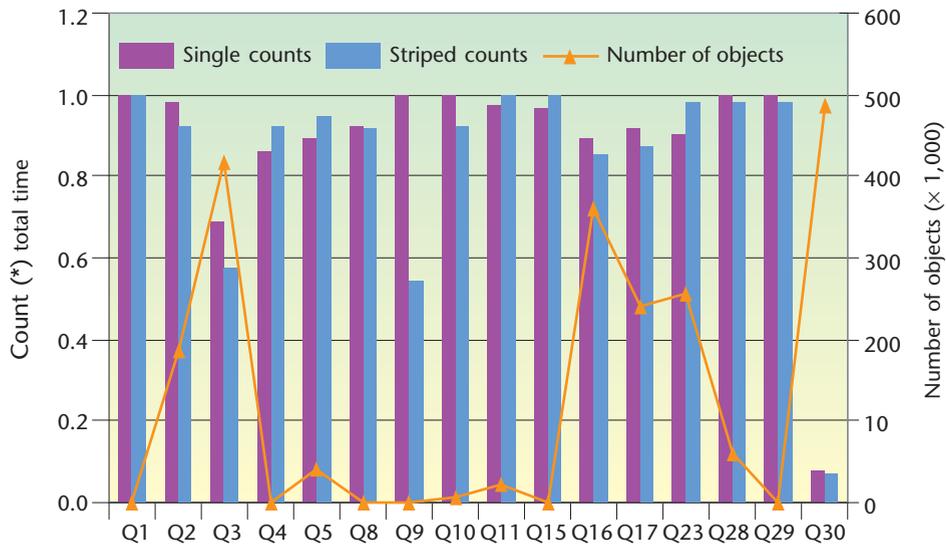


Figure 10. Comparison of query time saved by requesting aggregate counts versus total time for the OO-single and OO-striped versions with respect to the total number of objects returned by each.

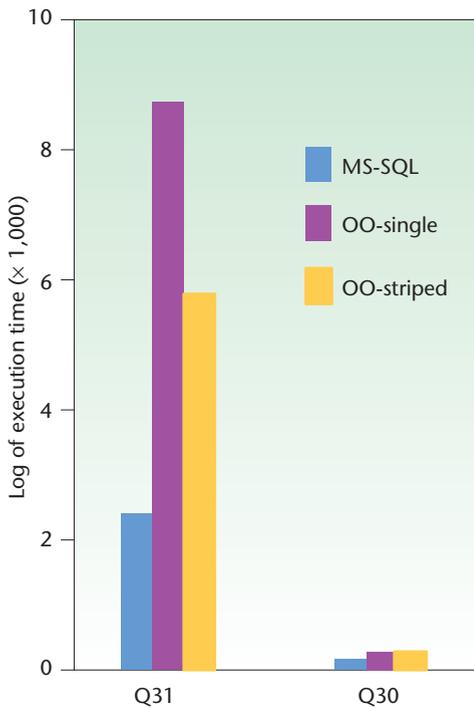


Figure 11. Comparing execution times for two versions of test query Q30. Q31 is a version of Q30 with a much larger limit on a nonindexed quantity.

After initially adopting an object database and expending considerable time and development effort converting it into an advanced data-mining facility, we were forced to abandon it and migrate to a

relational database system. The OODB concept was great—and probably the right decision at the time—but the actual implementation lacked the robustness, the tools, and the scalability our project demanded.

The detailed description of our object-based implementation substantiates the concerted attempt we made to achieve the data-mining objectives of the SA using an OODB. In the end, however, we had to acknowledge that relational database technology and Web-based protocols had advanced to the point where we could satisfy many of our original reasons for adopting an object database with a relational solution, and the dominant manageability, quality, performance, and usability requirements mandated that we use a mainstream product.

Although we only tested Microsoft's SQL Server, all the major commercial RDBs now offer superb query optimization, excellent I/O performance on RAID systems, and advanced support for data-mining applications and data-intensive science. In particular, the ability to encode complex mathematical and statistical analysis directly in a database within stored procedures and functions is crucial for the anticipated need in many branches of science to keep the compute-intensive processing as close to the data as possible to avoid network bottlenecks as the data volumes explode.

As the volume of the SDSS data more than quadruples over the next two to three years, we aim to maintain current data-mining performance by deploying the SDSS databases over a cluster of

linked DB servers. We will achieve load-balancing by either distributing the number of queries over the cluster, partitioning the data across the cluster, or both. We anticipate that this will be quite a challenge, but we are much better positioned to overcome it now that we have the performance and support that we should rightfully expect from our database layer. 

References

1. A. Szalay, "The Sloan Digital Sky Survey," *Computing in Science & Eng.*, vol. 1, no. 2, 1999, pp. 54–62.
2. A. Szalay et al., "Designing and Mining Multiterabyte Astronomy Archives, The Sloan Digital Sky Survey," *Proc. Special Interest Group Management of Data Conf. (SIGMOD)*, ACM Press, 2000, pp. 451–462.
3. A. Thakar et al., "Multi-Threaded Query Agent and Engine for a Very Large Astronomical Database," *Proc. Astronomical Data Analysis Software & Systems IX* (Astronomical Soc. Pacific Conf. series, ADASS), vol. 216, D. Crabtree, N. Manset, and C. Veillet, eds., Astronomical Soc. Pacific, 2000, pp. 231–234.
4. A. Thakar, P. Kunszt, and A. Szalay, "Case Study of Handling Scientific Queries on Very Large Data Sets: The SDSS SA," *Mining the Sky*, *Proc. Max-Planck-Institut für Astrophysik/European Southern Observatory/Max-Planck-Institut für Extraterrestrische Physik Workshop*, A. Garching et al., eds., Springer-Verlag, 2001, pp. 624–630.
5. P. Kunszt et al., "The Indexing of the SDSS SA," *Proc. Astronomical Data Analysis Software & Systems IX* (Astronomical Soc. Pacific Conf. series, ADASS), vol. 216, D. Crabtree, N. Manset, and C. Veillet, eds., Astronomical Soc. Pacific, 2000, pp. 141–144.
6. P. Kunszt, A. Szalay, and A. Thakar, "The Hierarchical Triangular Mesh," *Mining the Sky: Proc. MPA/ESO/MPE Workshop*, A. Garching et al., eds., Springer-Verlag, 2001 pp. 631–637.
7. T. Dyck, "Clash of the Titans," *PC Magazine*, vol. 21, no. 6, 26 Mar., 2002, pp. 122–138.
8. A. Thakar, P. Kunszt, and A. Szalay, "A Parallel, Distributed Archive Template for the NVO," *Proc. Astronomical Data Analysis Software & Systems X* (Astronomical Soc. Pacific Conf. series, ADASS), F. Harnden, F. Primini, and H. Payne, eds., Astronomical Soc. Pacific, 2001, pp. 238–240.
9. J. Gray et al., *Data Mining the SDSS SkyServer Database*, tech. report MSR-TR-2002-01, Microsoft Research, 2002, ftp://ftp.research.microsoft.com/pub/tr/tr-2002-01.pdf.

Ani Thakar is a research scientist in the Center for Astrophysical Sciences at Johns Hopkins University. His research interests include data mining, Grid computing, VLDB, virtual observatories, scientific programming, high-performance computing, interacting galaxies, and astrophysical simulations. He has a BS in physics and computer science from Carleton University, and an MS and a PhD (both in astronomy) from Ohio State University. He is a member of the IEEE Computer Society, the American Astronomical Society, the Astronomical Society of the Pacific, and the Phi Kappa Phi Honor Society. Contact him at the Ctr. for Astrophysical Sciences, Dept. of Physics and Astronomy, The Johns Hopkins Univ., 3701 San Martin Dr., Baltimore, MD 21218-2695; thakar@pha.jhu.edu.

Alex Szalay is Alumni Centennial Professor of Physics and Astronomy at Johns Hopkins University. His research interests include cosmology, large-scale structure of the universe, data mining, Grid computing, VLDB, and virtual observatories. He has a PhD in astrophysics from Eötvös University, Hungary. He is a member of the IEEE Computer Society, the American Astronomical Society, and the ACM. Contact him at the Ctr. for Astrophysical Sciences, Dept. of Physics and Astronomy, The Johns Hopkins Univ., 3701 San Martin Dr., Baltimore, MD 21218-2695; szalay@jhu.edu.

Peter Kunszt is a member of the database group (IT division) at the European Organization for Nuclear Research (CERN). His research interests include data mining and Grid computing. He has a PhD in theoretical physics from the University of Berne, Switzerland. Contact him at CERN, 1211 Geneva 23, Switzerland; peter.kunszt@cern.ch.

Jim Gray is a distinguished engineer in Microsoft's Scaleable Servers Research Group and manager of Microsoft's Bay Area Research Center (BARC). His technical interests include databases, data mining, and transaction processing. He has a PhD in computer science from University of California, Berkeley. He is a member of the ACM, the National Academy of Engineering, the National Academy of Sciences, and the American Academy of Arts & Sciences. Contact him at Microsoft Research, 455 Market St., Ste. 1690, San Francisco, CA 94105; gray@microsoft.com.



Take your e-mail address with you
Get a free e-mail alias
from the IEEE
Computer Society
and
DON'T GET CUT OFF
you@computer.org
Sign up today at
computer.org/WebAccounts/alias.htm