CONF-961209--1
ANL-HEP-CP--96-40

Flexible Storage Services for Parallel Data Mining

David M. Malon and Edward N. May

Argonne National Laboratory*
malon@anl.gov, may@anl.gov

13 May 1996

RECEIVED
APR 14 1997
OSTI

Abstract

The demands of mining and analyzing vast amounts of data often lead scientists to supercomputer centers, with their high-performance parallel processors and large-scale hierarchical storage. Once there, however, clients quickly come face to face with a number of harsh realities. Common constraints are:

- disk space, while impressive in aggregate on machines with more than 100 nodes, generally amounts to only a couple of gigabytes per node;
- local disk space is *scratch* space—every query starts and ends with no data on compute nodes' local disks;
- mass storage is generally a (widely) shared resource, and is not user-configurable;
- machine use is scheduled—no daemon processes may be left running;
- while some nodes may be "closer" than others (e.g., HIPPI-connected) to mass storage, current schedulers tend nonetheless to allow users to specify only the number of nodes desired, not their I/O topology;
- mass storage access from multiple nodes may in fact be routed through a single node (e.g., a distinguished I/O node per rack).

*The submitted manuscript has been authored by a contractor of the U.S. Government under contract No. W-31-109-Eng-38. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

One consequence of these conditions is that the efficacy of various approaches to physical data organization—for example, clustering data segments containing like objects versus striping them across multiple Unitree hierarchies—may depend, not only upon the computing platform's architecture, the data model, and the nature of typical queries, but also upon dynamic storage system configurations, and even upon the particular collection of compute nodes assigned to process a given query. In the face of these considerations, a fundamental requirement of storage management for persistence servers is flexibility—servers must support data storage across a heterogeneous mix of media and access methods, and must be able to reconfigure their data storage efficiently. This paper describes an architecture for storage management that addresses such concerns, and an implementation that has been tested both on parallel supercomputers and on networks of Unix workstations.

Introduction

On a dedicated parallel or distributed query processing platform, or even on a collection of machines on which one owns local storage, one can often physically partition data among processors, either disjointly or with judicious replication, and parallelize queries accordingly. On many large-scale machines used by scientists and engineers, however, all that a client may do is to request time on *any* P nodes, not one of which has resident data, though each has local scratch space. In general, all nodes have access to mass storage (e.g., Unitree) and to a shared, remotely-mounted file system (e.g., the Andrew File System) through which home directory services are provided. A parallel file system such as IBM's Vesta or PIOFS [6] may also be available; this may, however, be configured to serve only as short-term or scratch space, and hence, may need to be preloaded from multilevel mass storage.

It is characteristic of our motivating applications in high energy physics that queries can often be parallelized almost arbitrarily in principle. We may seek, for example, all events satisfying a certain criterion (whose determination may be quite computationally intensive), but those events are often essentially independent, or conditionally independent given experimental run parameters. Event collections, or collections of runs, which correspond to series of events under fixed experimental conditions, are candidates for implementation as *ParSets* [5], or for implementation in other ways that make explicit the potential for parallelism. The problem, though, is this: the appropriate number of processors for a query may not be a characteristic of the query (an arbitrarily large number could be used), nor even of the physical data store design (we may not be able to choose a set of n nodes that together provide local access—disjoint or otherwise—to the data); rather, it may depend upon such details as the accidental assignment of the query to a set of processors—8 processors

in a single rack, for example, may be less effective than 4 in multiple racks because of I/O bottlenecks.

To address these considerations, and to provide a testbed for alternative data organizations in light of these issues, we have designed and implemented storage services for an object persistence manager, with the goal of efficient and flexible utilization of parallel and distributed computational and storage architectures. Among our design criteria are:

- access to every persistent object from every query node;
- extensible support for a variety of storage mechanisms, including local and remote disk, raw RAID, Unitree file systems, raw device access to DD2 and 8mm tape, parallel file systems such as (formerly) IBM's Vesta and (currently) IBM's PIOFS, and Internet data access via standard FTP and HTTP mechanisms or cgi-bin scripts;
- support for efficient reorganization of data, including striping and reclustering, without knowledge of object schemata;
- support for data replication;
- support for multiple access paths to data;
- portability to heterogeneous distributed architectures.

Each of these criteria will be addressed in the following sections.

Persistent Storage Access Model

The ODMG-93 Release 1.2 C++ binding defines the user interface to our persistence manager—briefly, persistent objects reside in a database instantiated by the ODMG-defined class d_Database, and references to persistent objects of class T are made via a template class d_Ref$< T >$. Complete details appear in the current ODMG draft standard [1]. Argonne's portable ODMG-compliant frontend to object persistence managers is described elsewhere [2]. Internal to the Argonne object persistence manager, data are organized logically into *stores.* Within each store, objects are allocated in contiguous blocks of bytes called *segments.* Segments are the fundamental units of data transfer to and from a query process—when a reference to a persistent object is dereferenced, the corresponding segment containing the object is located and moved or mapped into memory.

For a variety of reasons, segments are generally inappropriate as the highest-level units for storage management—there are too many of them, and they are far too

small for efficient storage and retrieval on high-latency devices (unless they are far too large to be appropriate for memory caching when a single object is touched).

Segments may instead be aggregated arbitrarily to form *folios*, which are the units in which storage devices will deal with the data. Examples of folios include an ordinary Unix file containing one or more segments, a raw DD2 tape partition containing n consecutive segments of a store, a raw RAID partition contiguously containing every k^{th} segment of a store, and a Unitree file containing the k most frequently accessed segments. Folios have appeared in other lightweight object persistence managers such as PTool [3], but the design differences between PTool and the Argonne software are substantial. In PTool, a persistent pointer names the folio in which the object is contained. In our software, folios are *orthogonal* to persistent pointers. We can reorganize segments into arbitrary folio arrangements, and all persistent pointers will be unchanged and valid. A storage server is free to rearrange segments without knowledge of segment contents, and without concern for external references to objects within the segment.

Other object systems such as SHORE (Scalable Heterogeneous Object REpository [4]) provide more fully featured persistence frameworks, but pay less attention to matters of physical data reorganization on heterogeneous distributed multilevel storage architectures. In SHORE, object data are stored on *pages* (corresponding roughly to segments) allocated from disk *volumes*, each of which is managed by a single server. In our software, segments are allocated as stores grow, but a store's segments may be arbitrarily distributed across multiple folios and multiple storage devices; moreover, they may be reshuffled as the need arises, and are not managed by any single server.

Segment Locators, Segment Servers, and Retrieval Rules

Physical location and access method identification for segments are maintained in metadata—in this case, data about the storage configuration—represented by an ordered list of retrieval rules. Dereferencing a pointer to a persistent object identifies the store number and segment number containing the object data. A Segment Locator, which is in general replicated on each compute node, provides a *whohas()* method to map this ordered-pair segment identification into a segment retrieval rule, customarily the first retrieval rule in the list matching the segment id. A *getServer()* method in turn accepts a retrieval rule as an argument, and returns a pointer to a segment server capable of reading and writing the corresponding segment.

Segment Servers

An abstract SegmentServer base class defines the interface to the handful of methods that all segment servers must provide, such as segment creation, retrieval, and updating. Segment servers for particular devices are derived from this base class. To add a new storage medium to the list of supported devices, one need only implement the SegmentServer methods for the particular device, add a new reserved word to the retrieval rule lexicon, and update the getServer method to associate the new reserved word with the new Segment Server—all other code continues to operate unchanged.

Retrieval Rules

What does a retrieval rule look like? For a single segment, it may be as simple as associating a hostname (or localhost), a device type, and an address (for example, a Unix file name, an offset for a raw device, or a partition number and an offset for a tape) to a (store, segment number) pair. A single rule may define a placement convention for all the segments in a store by use of a wildcard character in place of the segment number. Such a rule might specify, for example, that each segment k should be stored in Unitree with the pathname /mss/username/MyDatabase.k). If a particular segment is to be handled differently, perhaps because it has been locally cached, all that is necessary is to place a rule corresponding to that particular segment number *earlier* in the rule list than the default rule for segments of that store. Retrieval rules may also define segment-level striping (e.g., place segments in contiguous blocks of four, in round-robin fashion, into three Unitree files, repeating the process until the files are 32 segments long, then build three new Unitree files and repeat the process).

How Storage Services Are Used

In practice, one often begins with a master retrieval rule list that describes the location of data segments (often in mass storage). When segments are cached or copied or moved to another storage location, a new retrieval rule list is derived from the original by placing a rule for reaching the new location earlier in the retrieval rule list. A segment locator using the new rule list will match the segment id to the retrieval rule it encounters first. If the old rule described only the location of the relocated segment (and not, for example, a rule for finding all segments in the store), it may be deleted from the rule list.

Rearranging data is straightforward: a utility for that purpose can be built essentially by incorporating a segment locator to read the current retrieval rules, and an array of segment servers to read segments and write them elsewhere. The point

is that no knowledge of the underlying object schemata is required—storage can be managed orthogonally to the data store's content.

Local Caching and Replication

When a segment is copied, a corresponding updated retrieval rule list reflects its new location. Since each query node may have its own segment locator, each may use its own retrieval rule list. The consequence is that there is a choice—one could have each node i retrieve data from node j rather than from mass storage, for example, by sharing node j's rule list, or have each node talk only to its own disks and to mass storage by not sharing retrieval rule lists.

Preloading Local Disks

Recommended policy on many massively parallel architectures is to preload data, particularly shared read-only data, onto local disks before running the computational portion of a job. This is important in order to avoid serial bottlenecks (often especially paralyzing when hundreds of nodes are trying to read the same AFS- or NFS-mounted file, or even different files from the same file system). Systems often provide utilities to copy data to P parallel nodes in $\log P$ time. When data segments are preloaded, corresponding retrieval rule lists are built to reflect the new locations. As noted above, these lists may be different on every query node, but when the same segment is replicated, things are generally simpler. If segment k of a certain data store contains, for example, calibration data needed in the analysis of each event, segment k may be replicated on each query node's local disk, and a corresponding retrieval rule matching the segment id to, for example, a disk file named /tmp/scratch/username/MyDatabase.k on localhost, would likely be identical on every node.

Multiple Access Paths

The architecture allows multiple retrieval rules for a single data segment. The design is intended eventually to support, for example, finding the first matching retrieval rule for a segment, trying it, and if it fails to return in an acceptable amount of time, trying the next matching rule, or even associating estimated costs with each candidate retrieval rule and optimizing the choice. We have implemented neither of these approaches to date. Different processes may, however, follow different access paths to the same data by using different retrieval rule lists.

Status

We have implemented the architecture described above on the Argonne 128-node IBM SP system, and on networks of Unix workstations. We have developed segment servers for all of the storage devices mentioned in the text. We have used these facilities both to provide access to storage on parallel and distributed platforms with a heterogeneous mix of storage media, and as a testbed to begin study of the complicated issues involved in physical data organization—alternative striping strategies, caching, replication, use of multilevel storage, and the role of parallel file systems. The ability to rapidly reconfigure our storage utilization without worrying about data store contents has proven invaluable.

Future work on the storage server itself will involve tools for managing retrieval rule lists, and smart (e.g., configuration-aware) tools for automating storage reorganization.

Acknowledgment

The authors gratefully acknowledge use of the Argonne High-Performance Computing Research Facility. The HPCRF is funded principally by the U.S. Department of Energy, Mathematical, Information and Computational Sciences Division (ER-31).

References

[1] R.G.G. Cattell et al, *The Object Database Standard: ODMG-93 Release 1.2* (Morgan Kaufmann, San Francisco, 1996).

[2] D.M. Malon and E.N. May, *On Persistence Interfaces for Scientific Data Stores* (submitted for publication, 1996).

[3] R. L. Grossman and X. Qin, "Ptool: a scalable persistent object manager," *Proceedings of SIGMOD 94,* ACM, 1994, page 510.

[4] M. Carey, D. DeWitt, J. Naughton, J. Solomon, et al, "Shoring up persistent applications," *Proceedings of SIGMOD 94,* ACM, 1994.

[5] D.J. DeWitt, J.F. Naughton, J.C. Schafer, S. Venkataraman, "ParSets for parallelizing OODBMS traversals: implementation and performance," *Computer Sciences Department Technical Report, University of Wisconsin, Madison*, 1994.

[6] *IBM AIX Parallel I/O File System: Installation, Administration, and Use* (International Business Machines Corporation, Document Number SH34-6065-01, 1995).