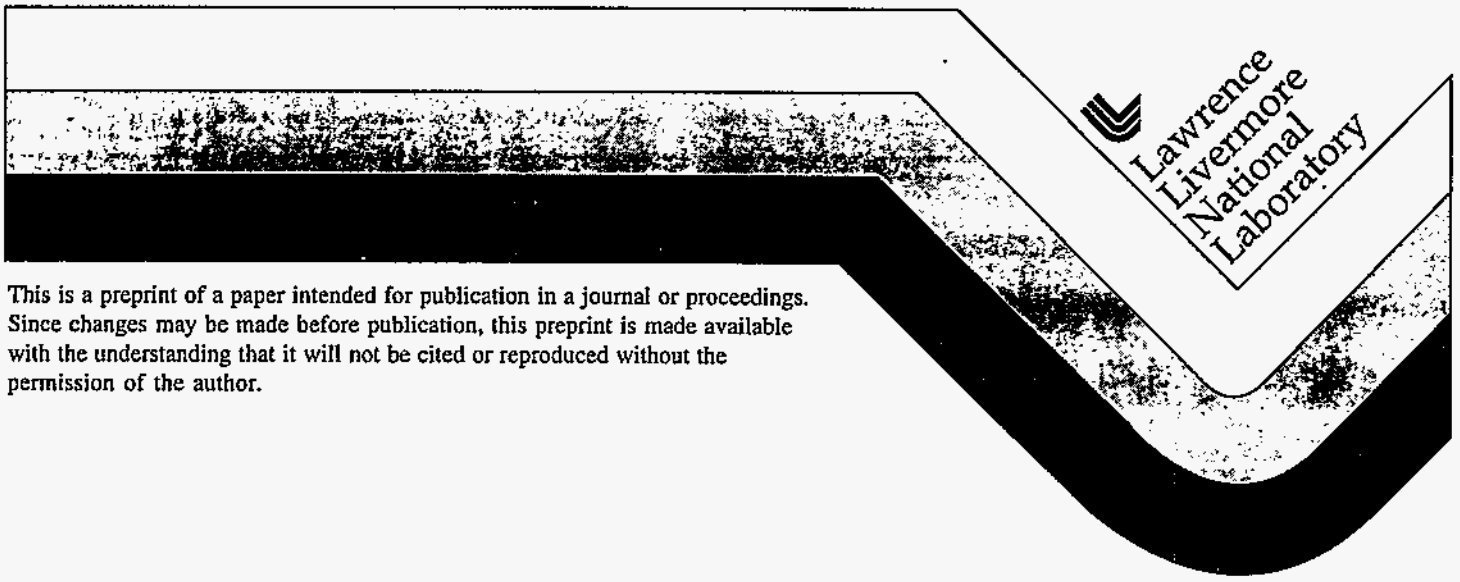
CONF-9604139--1
UCRL-JC-123875
PREPRINT

Metadata for Balanced Performance

R. Musick
P. Brown
R. Troy
D. Fisher
S. Louis
J. McGraw

This paper was prepared for submittal to the
IEEE Metadata Conference
Washington, DC
April 15-19, 1996

April 1996



DISCLAIMER

Metadata for Balanced Performance*

Paul Brown, Richard Troy
University of California at Berkeley
Computer Science Division, Berkeley, CA 94720
{pbrown, rtroy}@postgres.berkeley.edu

Dave Fisher, Steve Louis, James R. McGraw, Ron Musick
Lawrence Livermore National Laboratory
P.O. Box 808, Livermore, CA 94551
{dsf, stlouis, jmcgraw, rmusick}@llnl.gov

Abstract

Data and information intensive industries require advanced data management capabilities incorporated with large capacity storage. Performance in this environment is, in part, a function of individual storage and data management system performance, but most importantly a function of the level of their integration. This paper focuses on integration, in particular on the issue of how to use shared metadata to facilitate high performance interfaces between Mass Storage Systems (MSS) and advanced data management clients. Current MSS interfaces are based on traditional file system interaction. Increasing functionality at the interface can enhance performance by permitting clients to influence data placement, generate accurate cost estimates of I/O, and describe impending I/O activity. Flexible mechanisms are needed for providing this functionality without compromising the generality of the interface; we are proposing active metadata sharing. We present an architecture that details how the shared metadata fits into the overall system architecture and control structure, along with a first cut at what the metadata model should look like.

*This work was funded in part by the National Aeronautics and Space Administration (NASA) EOSDIS grant with University of California at Berkeley, reference number 3579-IST94-0086 and under contract number L-4947 at Lawrence Livermore National Laboratory. Portions of this work were performed under the auspices of the U.S. Department of Energy's Mathematical, Information and Computational Sciences Program (grant KC 07-01-03)and by the Lawrence Livermore National Laboratory under contract number W-7405-ENG-48.

1 Introduction

It is becoming commonplace for large enterprises to be considering mining, analyzing or visualizing datasets of terabyte sizes today, with plans to store and access petabytes in the near future. The sheer size of the data alone requires cost effective storage solutions to be based in part on tertiary storage. These storage solutions must provide high performance access to data in order to support analysis tasks over the huge amounts of data involved. A necessary element in achieving high performance is fast individual storage and data management systems. However, the overall system performance will be driven by the level of integration between these two pieces. Currently, interfaces to Mass Storage Systems (MSS) are limited to the traditional file system style of interaction. The data management client is forced to view storage as a black box that gives up no control or information beyond what is required for basic I/O. Interfaces like this unbalance high performance solutions because no matter how advanced either endpoint is (data management, or storage), overall system performance will be dragged down to the capabilities of the interface between the two.

This paper focuses on integration – how to use shared metadata to facilitate high performance interfaces between mass storage systems and advanced data management clients. Examples of these clients include DBMSs, analysis-oriented object stores, intelligent layout tools, and high level applications. A flexible mechanism is needed for sharing information between these clients and the storage system. We present an architecture based on active metadata sharing that details how the shared metadata fits into the overall data management system architecture and control structure. An initial metadata model is included at the end of the paper.

One of the drivers for this effort arises out of a collaboration sponsored by NASA for its Earth Observing System, Distributed Information System (EOSDIS). EOSDIS requires massive amounts of diverse data to be stored, organized, distributed, visualized and analyzed. The collaboration as a whole considers the end-to-end system from the mass store to the desktop; storage, networking, distributed file systems, extensible data base management, and visualization. Good end-to-end performance requires that the various subsystems be well integrated, including the EOSDIS client DBMS and tertiary storage. The use of shared metadata as a focal point for subsystem integration is a controllable, flexible method for enabling integration and the applicability extends well beyond the specific NASA effort.

The paper discusses several topics. Section 2 motivates the new functionality being added to the interface to mass storage. Section 3 introduces the major features of the interface and describes an architecture based on an active metadata model that transforms the metadata system into an active part of data management. The central component of this architecture is the Interface Data Repository (IDR). The IDR safely and accurately describes to any client where its data objects are stored and the costs associated with manipulating those objects. It also permits some control over the placement and movement of data in the MSS. Section 4 discusses several of the key issues involved with

this work. Section 5 closes with a brief statement of future work. An initial metadata model can be found in the Appendix.

2 Motivating the Interface Functionality

The interface between the MSS and advanced data management clients plays a critical role in the performance of the overall system. We claim there are four functionalities that will significantly enhance that performance.

- **Flight plans:** The MSS needs information on the current and near term I/O plans of the client in order to effectively schedule internal operations across all clients (for example, pre-fetching client data).
- **Information on data access costs:** The MSS client needs estimates of how long I/O operations will take in order to intelligently schedule operations.
- **Influence over data layout:** The client must have some degree of control over how the data is laid out in the storage system. The client should also be able to specify the criticality of directives, from "must comply" to "suggested advice".
- **Transaction semantics:** The interface should provide ACID semantics, to protect client and MSS data and operations.

Flight Plans

Flight plans allow the MSS to more intelligently schedule internal operations. The MSS is the only entity that is completely "aware" of its own resources, and of the current I/O requests that it must satisfy from all clients. If the MSS has access to information about jobs that will be requested in the near future, then its scheduler will be better able to optimize overall system throughput[1], and by doing so improve overall system performance. For example, a DBMS client could alert the MSS that it is now beginning a specified sequence of I/O operations in response to a query. Seeing the entire sequence in advance, the MSS could minimize the number of tape mounts performed during the course of the query by pre-staging the data that will be used at a later point in the query *before* that tape is unmounted. This will reduce the total time and effort spent on the job. Without the flight plan such optimization is not possible.

Data access costs

The ability to make cost estimates is particularly valuable to DBMS clients. Traditionally, DBMSs that need high performance put all data on fast random access disks. DBMS query optimizers plan out how to most efficiently retrieve information to minimize the resource cost of a query (space, cpu, I/O, time) [6], based on the easily computed performance characteristics of the disks. A DBMS that can produce the same quality of time

[1]There are other potential metrics that an MSS might optimize for instead, e.g. minimum response times. Site policy will determine which metric is in effect.

cost information running on top of an MSS with tertiary storage would gain substantial capacity, without sacrificing much in terms of performance. However, traditionally MSSs do not even provide the most basic information – whether specific data is on a disk or a tape. The differences between the two classes of media (average transfer rates, latency, seek rates, difference between the transfer times in best and worst cases) are significant and have a large impact on optimizer performance. An MSS interface that provides good information on the time cost of data access throughout its storage hierarchy is necessary to enable effective query optimization. This kind of interface meshs well with current query optimization research [5, 7] that is adapting the algorithms to deal with tertiary device characteristics.

Data Placement

The case for being able to influence data placement in the MSS is best made from the viewpoint of an intelligent data layout client. Briefly, tape-based mediums in tertiary storage are linear access devices. By far the most efficient way to sequentially access two objects on tape is to put them on the same tape, one after another. Intelligent data layout is based on the principle that data should be stored on these devices *in the order that it is most likely to be retrieved*, which is not necessarily the order in which it was stored. Optimass [2] implemented this concept in global climate data modeling and has shown performance improvements of 2 to 15 times faster I/O speeds over a set of 70 queries against the data. Optimass determines better layouts for the data automatically, however, the actual tapes that get built according to the layouts must be built by hand. This is because currently clients can not influence data placement in mass storage systems.

Beyond enabling raw performance improvements, providing this functionality in the interface is crucial for controlling error. Certain client data (e.g. DBMS indexes) have a zero tolerance for bit-level errors, which translates to a zero-tolerance for the use of most tape-based storage devices. With no mechanism to influence this situation, the robustness of certain MSS clients is significantly degraded.

Transaction Semantics

Finally, the interface must provide good transaction semantics. There are four important properties of all transactions: Atomicity, Consistency, Isolation, and Durability (ACID) [4]. Atomicity requires that each transaction (e.g., I/O operation) be "all-or-nothing." Consistency requires that the information about the state of the system always remain in a consistent state. For example, part of the information cannot show that data has been moved, while other parts show the move has not been completed. Isolation requires that two different operations cannot interfere with each other, even though they may be running concurrently. Durability requires that once a transaction commits, its changes survive, even if there is a subsequent system crash. The interface is the focal point for the metadata that drives both the data management clients, and the storage system. Without these four properties, both the safety of the data being stored and the correctness of the applications being run would be at stake.

3 Approach

The Interface Data Repository provides the functionality described above through the following key features:

- **An abstract view of MSS resources.** Clients can gather information about their stored data through abstractions of MSS stores. The abstractions describe performance characteristics of logical stores and the locations of data objects on those stores. Data management clients can use these abstractions to determine the expected I/O costs of different data movements (which is a critical component of query optimization). The MSS can change the physical devices in its domain and create new storage services with minimal impact on the external clients. These abstractions permit the IDR to be implemented across different MSSs with many different kinds of storage hardware.

- **Data movement specifications by clients.** Clients can directly request all types of data movement, including I/O, caching, and migration. Clients can also recommend relative placement strategies for data blocks being moved. Class of service directives allow for keeping certain critical data on the most reliable devices, and for specifying the degree of criticality of hints.

- **Advanced notice of impending data movement requests.** Clients can choose to provide this type of information. It gives the MSS greater latitude in its optimization of resource usage, which should further enhance performance.

By appropriately structuring the metadata that is shared between the clients and the MSS, a general-purpose interface can be defined that provides the needed high degree of functionality to the data management clients. This section introduces the interface architecture. Section 3.1 gives an operational overview of the Interface Data Repository, and describes how it interacts with the MSS and MSS clients. Section 3.2 describes the metadata contained in the IDR, the parties responsible for maintaining the tables, and the access privileges to various portions of the data. Section 3.3 identifies a few active triggers within the IDR for notifying various parties of relevant changes to the IDR. Section 3.4 provides a simple example of using the IDR.

3.1 Operational Overview

The interface architecture is pictured in Figure 1. The figure shows three major software components: an MSS, an IDR and a client. The heavy lines represent the movement of data to/from the stores. These data paths are separate from the control paths to avoid the need for the IDR to store and forward all of the data coming from the MSS (third party transfer). The narrow lines in the figure represent pathways for specifying commands and responses between these components. The IDR contains metadata

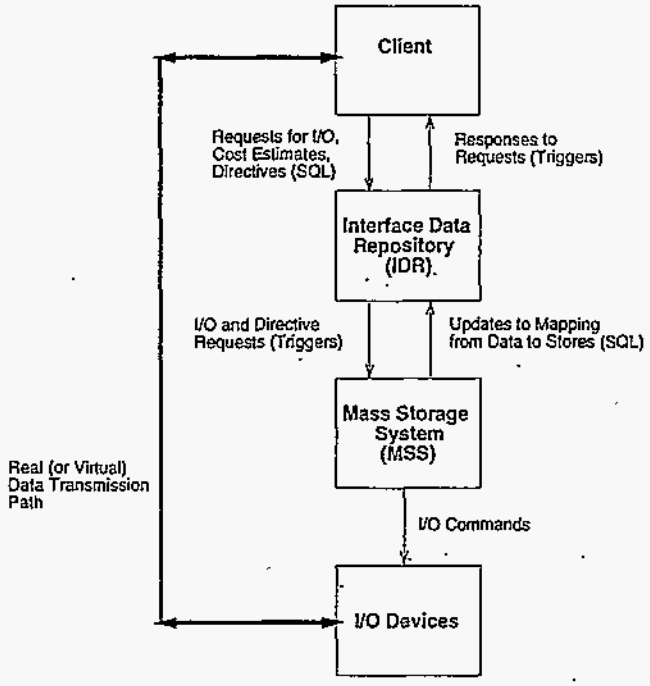


Figure 1: **Proposed Interface Architecture**

which describes the holdings of an MSS, along with data representing clients' requests, and is implemented in a relational DBMS (RDBMS). Typically, an RDBMS includes functionality such as a query language, active database features (e.g., rules and triggers), ACID transaction semantics, and security. For the IDR design, we assume that the query language used to access and modify the content of the IDR is SQL92 [1]. The MSS manages its stores, implements site policy, and honors requests as best it can given what it knows about its entire work load. A client focuses on optimizing the requests it makes based on what it knows about the available stores, and the data objects that currently reside there.

Normal operation of the system distributes responsibility for actions between the MSS and its clients. The MSS has primary responsibility for maintaining most of the information in the IDR. It keeps a mapping of all data objects to their current virtual stores. As the MSS migrates objects to different stores, it updates the IDR to reflect these moves. The MSS also maintains the performance characteristics for each of the virtual stores. The client's responsibility is to post information to the IDR to request I/O activity, issue directives, or to announce an intent to request future I/O activity. With this information, an MSS may better optimize its overall performance. For example, an MSS may decide to pre-stage data to a more favorable location in anticipation of future I/O requests. Through standard SQL queries, a client can acquire information needed to make cost estimates for executing various I/O operations as a part of its query optimization activities. Clients can access only those portions of information in

the IDR related to their own objects. Finally, DBMS triggers are used to provide the active interface and notify the various components of changes in the IDR metadata that require attention (e.g., notifying the MSS of new I/O requests).

3.2 IDR Metadata Schema

The IDR metadata is organized as a standard relational schema. At present we have identified six tables that need to be maintained: Bundles, Stores, Blocks, Block-Equivalency, Movement, and Movement-Associations. This section describes the content and maintenance of these tables in an informal form. For those familiar with RDBMS notation, the Appendix contains a formal relational schema.

The Bundles table associates names of data bundles (names understood by the client) with unique MSS identifiers. A DBMS client will likely view a bundle as equivalent to a relation or a class. On the other hand, non-DBMS clients may prefer to view bundles as an uninterpreted strings of bytes of arbitrary length (i.e., files). Client operations on bundles can use client domain names and bundle-related offsets and lengths. This relieves clients of having to know more about the internal I/O operations of the MSS than they need or want to know. Bundles are divided into blocks of equal length. Block length is represented in the Bundle table as "BlockSize". The MSS is responsible for maintaining the content of the Bundles table.

The Stores table represents an abstracted model of a physical device (called a virtual store) that provides physical storage. The physical media underlying a row in this table may be a discreet unit, or a collection of units. For example, a virtual store could be an individual tape, a set of tapes managed in striped fashion, or even a hierarchy of heterogeneous stores. Among other things, this table contains the performance characteristics of these virtual stores. This table associates a unique "StoreId" with service and performance information such as capacity, default block size, cost of storage, mount time, access time, transfer rate, reliability level. We refer to this collection of performance information as "Class of Service (COS)." The key point for a client is that the service and performance information reflect as accurately as possible the costs for using each kind of store.

A more advanced feature of the Stores table is a second type of entry that is managed by each client. The IDR schema treats portions of main memory as a type of virtual storage device. Each client may allocate some main memory storage that it will use as space for moving data between itself and the MSS. It then places information about this memory into the Stores table. The information within the IDR about this memory is adequate for formulating any necessary I/O commands to and from these locations. By making portions of main memory part of the logical storage hierarchy, the IDR can represent all forms of I/O, data caching, and data migration in a uniform manner. How exactly to best operate within this new storage model is still a research issue.

The Blocks table describes the mapping from bundles to stores. Each bundle can be divided into an arbitrary number of blocks of uniform size. The blocks table can be used

to find the sequence of blocks that make up a bundle. In addition, the table indicates the virtual store and location on that store where the block is located. The Blocks table reflects the possibility that different portions of an object may reside on different types or instances of storage media. All portions of the Block table are maintained by the MSS.

The Block-Equivalency table represents situations where a block is stored more than once. This table allows the MSS and the owner of the blocks to locate redundant copies of a block. The MSS is responsible for maintaining this table. Both the MSS and the client (depending on policy) can use it for specific optimization purposes. Equivalent blocks may have different performance characteristics if located on different stores.

.The Movement table contains requests for the movement of data between Stores. The data object to be moved is identified as a particular block. The place to which data is moved can be specified generically in terms of a desired class of service or specifically as a "DesiredStorageId" and "DesiredAddressOnStore" pair. In the latter case, the address is a virtual address that permits different Movement table entries to relatively position blocks in a manner deemed favorable by a client. A movement can either be a "copy" or a "move." At present, this table has one field describing the priority of the proposed move, which is intended to cover a number of movement issues. A low priority number would indicate this move is mild advice to the MSS about the expected use of the data. A high priority number is essentially an I/O directive that should be carried out exactly as specified if at all possible. Another field in the table, "EquivalentBlockOk", permits the client to specify whether or not the MSS may use an equivalent block to the block specified for satisfying this request. Finally, the "Status" field indicates the completion status of this requested movement. All fields in the Movement table (except the Status field) are managed by the client.

The Movement-Associations table permits the client to logically group related Movement entries. This grouping conveys to the MSS that all movements within a group are part of a larger plan. This knowledge of groupings may assist the MSS in making improved scheduling decisions for optimizing performance.

3.3 IDR Rules and Triggers

The functionality of the IDR extends beyond simply maintaining the metadata described above. The IDR is responsible for informing the MSS and the clients when interesting events occur; thus the IDR is an active interface that participates in the overall data management solution.

An RDBMS implementation of the IDR permits the definition of various rules and triggers (or alerts) that govern the actions that can be taken when certain events occur. This section briefly describes a few examples of rules and triggers, making it possible to illustrate the expected use of the IDR in practice.

DBMS rules describe actions that must be taken when specific access to a database match pre-defined conditions. For example, "on insert to Movement do ..." Such rules

are executed in all cases on behalf of all applications accessing a given database. A trigger, or alerter, on the other hand, is used for application specific communications, and can be raised under very exacting conditions. An alert may be "fired" by a rule, or "raised" by an application. In either case, what occurs depends on "who" is listening for the alert. In essence, an alert is a notification mechanism.

This triggering system can be used in numerous useful ways, such as implementing new, tailored interface schemes. For example, a Commands table could contain high-level commands that operate on Bundles (or portions of Bundles) instead of Blocks. A specific alert could translate the original high-level request into appropriate inserts into Movements rows for each Block. Examples of other rules and trigger concepts we expect to include in the IDR schema include alerting the MSS when new entries are made in the Movement table, and alerting the relevant client DBMS when a Movement status changes.

3.4 Simple Example of IDR Use

We can illustrate how this interface architecture works by studying a simple problem. Take the client to be a DBMS. When a client connects with the IDR, it must establish its identity and at that time receives the capability to view the portions of the IDR metadata that it manages. The client may then issue SQL commands to acquire information on the general location of its Bundles and attributes about the Stores on which those Bundles reside. At this time, the client may add into the Stores table information about the location of main memory space that the MSS may use for moving data, as described in Section 3.2.

When the client DBMS receives a query from one of its users, it begins to formulate possible execution plans to satisfy the query. As a part of query optimization, the client is likely to issue further SQL queries to the IDR to calculate the costs of various data movements associated with each plan. After evaluating some number of possible plans, a query execution plan is chosen. The client DBMS then posts in the IDR Movement table the specific I/O operations needed, along with any appropriate grouping of these moves represented in the Movement-Association table.

Inserting the I/O requests in the Movement table will set off a trigger for the MSS to begin executing the plan. The MSS can then view the new I/O requests with all other pending requests from other clients, and optimize the execution of all requests according to its established site policies. As the steps in the plan are carried out, the MSS can use SQL transactions to update the state of the data mappings and indicate the completion status of those operations. These changes to the IDR will then set off trigger(s) for the client DBMS to be notified of progress.

Using the IDR schema, a MSS may have additional opportunities for optimization, beyond intelligent ordering of data movements from the Movement table. The Movement-Association table allows an MSS to understand that a set of moves are related toward a common goal. Assume a group of movements reads data from two different

tapes in a staggered fashion (as might be required in a sort-merge algorithm). Further assume that only one tape drive is available. The MSS could decide to cache one of the tapes contents to disk to minimize mount time. If at a later time another tape drive becomes available, the MSS can decide to dispense with the caching and revert to direct moves from tape to memory.

4 Issues

There are many issues involved with building this interface. For example, with respect to the IDR metadata model there are questions such as: how exactly should the I/O cost estimates be translated to the client, should there be quality of service guarantees, what types of relations between blocks in bundles should be specifiable? In this section we focus instead on the system architecture issues that have come up.

There are two important policy issues. First, the interface must provide general-purpose access to the MSS storage facilities for all clients. This implies that only the MSS should exert total control over the scheduling and use of MSS resources, and so client directives of the "must comply" character must have a escape options for the MSS. Potential options include: "directive refused due to" or "directive delayed until". Second, the interface must separate implementation techniques from policy. Most MSS policies are site-dependent, and are based on specific performance and reliability objectives. The interface cannot impose specific policies, but should allow for the definition, association, and execution of whatever policies a particular MSS site has chosen. For the interface to be broadly acceptable, it must reflect this policy-neutral position.

One of the implementation issues is whether the control structure supports a transactional interface. As currently defined, the IDR is a reflection of the actual state of affairs as maintained by the MSS. The question is whether the underlying processes in the interface provide ACID semantics for the actual I/O operations. The simplest scenario (in terms of implementation) for providing the interface architecture on top of an existing MSS would be to have the IDR as a separate entity from the actual metadata for the data held within the MSS. In this case, assurance of the ACID properties will depend on the exact nature of the implementation of changes to the MSS internal data and the corresponding changes to the IDR. Full confidence in atomicity, consistency, and durability between the IDR and MSS versions of the metadata may be difficult to ensure.

An alternate IDR implementation strategy avoids these uncertainties. The MSS could make the IDR the one and only repository of all of its metadata. This implementation strategy would automatically provide ACID transaction properties for all updates to MSS tables, which would provide end-to-end assurances for all users of the IDR.

We are taking this approach for a prototype implementation, which is depicted in Figure 2. The MSS platform will be the High Performance Storage System (HPSS). HPSS is a collaborative development effort between IBM and four national laboratories,

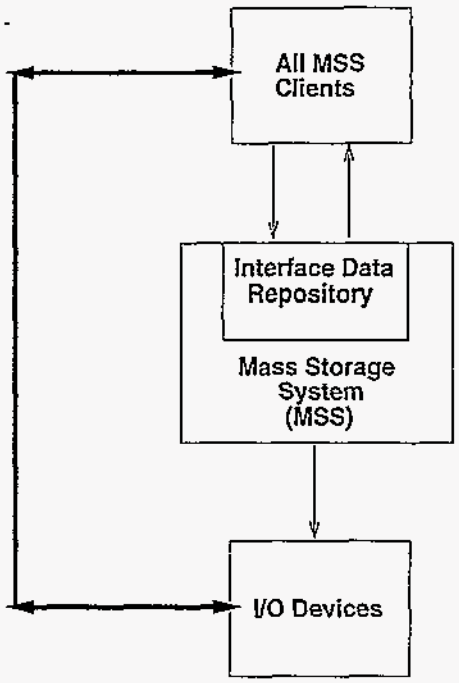


Figure 2: Implementation Architecture

including LLNL. HPSS build on standards where they exist; it is consistent with the IEEE Mass Storage Reference Model [3]. The initial implementation of the IDR will be accomplished using DB2 to extend the HPSS interface. We believe that DB2 provides the necessary performance levels required for the IDR implementation and it does so through a standard interface to the SQL92 standard query language.

We plan to develop an experimental version of HPSS (independent of the production versions) to implement the IDR. Our implementation will replace the tool that stores all of the HPSS metadata. Currently all non-volatile HPSS system metadata, including COS, Bitfiles, Storage Segments, and Virtual Volumes, are stored in B-tree based data structures controlled by Transarc's Encina and Structured File Service products. Access and update of this metadata is currently performed via the issuing of DCE RPCs to the HPSS Metadata Manager. The Metadata Manager is the only HPSS component that calls the Encina and SFS internal interfaces directly. Essentially, Encina will be replaced with DB2, modifying the HPSS Metadata Manager system calls to reflect this change, thus making it possible to move all HPSS metadata into DB2. The IDR can then be implemented as a view of the HPSS metadata. The scheme provides a standard security mechanism that will shield the client from data that is not part of the IDR abstractions, and data unrelated to that client. The resulting design smoothly merges MSS metadata with the IDR metadata, resolves any consistency issues, provides the MSS with quality COTS products for internal management, and provides the highest level of performance possible out of the IDR concept.

The final issue we need-to mention concerns current MSS users and how they might be affected by an IDR, particularly if they do not need higher performance. The implementation strategy we have chosen retains the possibility for existing MSS users to communicate with HPSS without any change. We do not expect the replacement of the HPSS metadata scheme (from flat files to an RDBMS) to adversely affect the performance of HPSS for these clients.

5 Future Work

The Interface Data Repository is nearing the end of the preliminary design phase, and we are about to begin the first prototype implementation. Some of the design decisions, like the control structure, are fairly stable. Other elements of the design are expected to undergo significant change over the next year or so. In particular, the metadata schema described in the Appendix is already slightly out of date, and will likely see the most changes as our experiences from the prototype and from interactions with the DBMS and MSS communities are incorporated into the project. As part of demonstrating the performance enhancements of the prototype, one of our tasks will be to create a performance benchmark. The long-term goal of this project is to develop an interface structure and metadata schema that is powerful and flexible enough to be accepted by the MSS and DBMS communities as the standard for a high performance DBMS/MSS interface.

References

[1] ISO/IEC 9075. *Information Technology - Database Languages - SQL 1992.* American National Standards Institute, NY, NY, 1992.

[2] T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani. Efficient organization and access of multi-dimensional datasets on tertiary storage systems. In *Special Issue on Scientific Databases, Information Systems Journal.* Pergammon Press, 1995.

[3] IEEE Storage System Standards Working Group. *Reference Model for Open Storage Systems Interconnections: Mass Storage Reference Model, Version 5.* IEEE, 1994.

[4] T. Harder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computer Surveys*, 15(4), 1983.

[5] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. Technical Report Sequoia 2000 report 92/13, University of California, Berkeley, 1992.

[6] A. Rueter and J. Gray. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[7] S. Sarawagi. Database systems for efficient access to tertiary memory. In *Proceedings Fourteenth IEEE Symposium on Mass Storage Systems, Monterey, CA*, 1995.

A Appendix

This is an SQL92 standard relational schema which represents the metadata which would be required for the IDR. This schema is not complete, but is included to give more substance to the discussion in the text. A full implementation will require a host of other metadata not previously mentioned, such as what users are authorized, or what tape drives are available.

In keeping with its modest objectives, the naming of tables and attributes is kept deliberately abstract so as to avoid the confusion which would result from attaching more over-loaded terms taken from a specific jargon. A rough mapping of tables to concept is included.

Bundles:

Bundles are the groupings of data. A Bundle is collection of blocks of no specified length. Bundles have Identifiers (BundleId) which are unique within the MSS. Bundles have Names which are used to identify them to the outside world.

Note that BlockSize is an attribute of a Bundle, not a Block, thus fixing the block size for a particular bundle. This simplifies access, because each block need not be interrogated in order to specify which particular block a "byte offset" belongs in, and thus, what happens when/if a (copy of) a block is put on a store which has different requirements. There are other sensible solutions too.

Storage:

Starting with CostPerKHour we get into defining the performance costs associated with getting to a block. Each of the following attributes (XchTime, AccessTime, etc) is a record of the mean - or some other, more statistically sophisticated measure - of this aspect of the Store's performance. Note that this is not necessarily an exhaustive list of what might be desired.

The inclusion of a ClassOfService reference is optional. If it is included, some of the I/O characteristics of a Store could be omitted, such as DataXsferRate. The argument for inclusion is that it permits an easier search for stores of a given characteristic. But then, a COS table, describing classes of service, and an association table, could perhaps do better.

The inclusion of CollectionId is intended to give a user some information about whether two particular Stores might be a part of different "collections", such as a tape robot serving a collection of 500 tapes, and hence, provide a modest insight into physical layout without loosing any abstraction. An implication might be that if two Stores are of the same CollectionId, then they are less likely to be concurrently accessed in real time.

This draws on the fact that the MSS must manage the metadata associated with such collections already (in a metadata schema we have chosen not to outline). CollectionId could have other uses too. It has been deliberately left undefined.

Blocks:

Bundles are divided into smaller units - called Blocks - for distribution among the Stores. This table manages the mapping of each Bundle's Blocks to the Stores. The situation is complicated by the fact that a Block may be stored several times. Also recall that Blocksize has been put as an attribute of Bundle fixing all blocks in a Bundle to the same size, simplifying otherwise complex operations. This need not be so, but an alternate scheme would need to concern itself with whether a block move from one store to another with different block sizes might cause problems.

Movement:

The movement table is not really necessary in the schema, but we have provided it to simplify some operations, and to make requests easier to express and to track. Virtually all of the attributes which describe a Movement could instead be expressed as some set of SQL statements over the tables above. One exception to this is the Priority attribute. This is intended to permit the user to specify whether a specific request is a "hint," to assist with caching if space is available, or a "demand".

Note that a Class Of Service table, COS, is mentioned, but not defined. The purpose of this table is to permit various classes of service to be defined independently of Stores, and let users specify Block movement to a class of service instead of a specific Store. if it is desired that an entire bundle be so moved, a simple SQL statement can populate the Movement table with requests for each block.

Movement-Associations:

The Movement-Associations table provides the MSS with the ability to discover what Movements are associated. This information is useful for interpreting the users intentions and in making hopefully optimal caching decisions.

To use this ability properly, a User should associate movements which are logically associated, and semantically at the same "level." This would be the case, for example, if a database were doing a "join" and needed to concurrently read from more than one entity at a time. if any one of these reads were blocked, the entire operation is effectively blocked. When such a blockage might otherwise be inevitable, the MSS may cache resources as appropriate to relieve the blockage.

The implementation of Move-1 and Move-n is certainly NOT the only way in which this could be implemented. For systems permitting it, the attributes should be "row" level pointers, TIDs, OIDs and the like, and NOT integers. Another alternative would be to add an attribute to the Movement table itself. There is nothing wrong with that idea, but we expect that many Movements will not have associations, and so do not need the attribute.

```
CREATE TABLE Bundles
(
BundleId            integer         not null, Primary Key,
BundleName          varchar(20)     not null,
BlockSize           integer         not null,
);

CREATE TABLE Stores
(
StoreId             integer         not null, Primary Key,
Capacity            integer         not null,
defaultBlockSize    integer         not null, /** in K bytes **/
Reliable            boolean         default False,
CostperKSperHour    decimal(5,3)    not null, /** in dollars **/
XchTime             decimal(10,5)   not null, /** in milli-seconds **/
AccessTime          decimal(10,5)   not null, /**in milli-seconds **/
DataXsferRate       decimal(10,5)   not null, /** K per milli-second **/
CollectionId        integer         not null, Foreign Key References(Collections)
);

CREATE TABLE Blocks
(
BlockId             integer         not null, Primary Key,
StoreId             integer         not null, Foreign Key References(Stores),
AddressonStore      integer         not null,
BundleId            integer         not null, Foreign Key References(Bundles),
BundleOffSet        integer         not null
);

CREATE TABLE Block-Equivalency
(
CanonicalBlock      integer         not null, Foreign Key References(Blocks.BlockId),
EquivalentBlock     integer         not null, Foreign Key References(Blocks.BlockId)
);
```

Table 1: **Metadata Schema**

```
CREATE TABLE Movement
(
MovementId              integer         not null, Primary Key,
BlockId                 integer         not null, Foreign Key References(Blocks),
EquivBlockOK            boolean         not null,
ClassOfService          integer         Foreign Key References(COS)
                                        check (if not null
                                                (DesiredStoreId is null
                                                AND DesiredAddressonStore is null)),
DesiredStoreId          integer         Foreign Key References(Stores),
                                        check (if not null
                                                (DesiredAddressonStore is not null
                                                AND ClassOfService is null)),
DesiredAddressonStore   integer         ,
Priority                integer         not null,
MovementType            varchar(20)     not null, check in  'Move', 'Copy' ,
Status                  varchar(20)     not null
);

CREATE TABLE Movement-Associations
(
Move-1                  integer         not null, Foreign Key References(Movement.MovementId),
Move-n                  integer         not null, Foreign Key References(Movement.MovementId),
);
```

Table 2: **Metadata Schema, continued**