Operational Data Storage Unification

James R. Hamilton

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Technical Report CS-97-16

July 6, 1997

Abstract

We see a world where personal system software is self-installing, self-configuring, self-personalizing, self-administering, and self-healing. In this world, personal system data is visible in a single name space regardless of "owning" application and data location, supports uniform non-procedural query, transactions, and security, is automatically backed up, is recoverable, and supports portable and disconnected operation.

This paper focuses on how database technology can be applied to help achieve many of these goals. We propose an integrated storage manager, the Virtual Object File System (VOFS), which supports structured files, unstructured files, and relational access. VOFS also supports disconnected operations by caching and replicating all data stored via any of the supported persistence APIs. We show how the combination of support for storage integration and disconnected and portable operation found in VOFS can substantially reduce administrative costs and support more compelling client applications.

The VOFS system can be extended to support alternative caching systems, new access methods based upon past navigational paths and temporal access, and different software and data billing schemes.

Acknowledgements

I would like to thank my supervisor Dr. Jacob Slonim and reader Dr. Grant Weddell for advice, comments, and suggestions throughout this project that have both improved the paper and my understanding of the subject material. Dr. Pat Selinger of the IBM Almaden Research Center provided numerous invaluable technical and editorial comments and was always available for technical discussion. I would also like to thank Dr. Paul Larson of Microsoft Research for time, advice, and education in the early stages of the degree program.

The IBM Canada Laboratory provided financial assistance for this work through the CONGESE program.

Table Of Contents

1 Introduction

Most persistent client applications and server[1] subsystems implement private storage managers that are uniquely tailored to the needs of the particular application or subsystem. Client-side applications typically implement persistence by storing into either individual file system files or somewhat more complex storage structures implemented above the file system. Server-side subsystems such as mail, queuing, net news (NNTP), vertical applications, HTTP, directory, groupware, workflow, and other collaborative frameworks generally implement a proprietary storage subsystem built upon the underlying file system.

The advantage of having each persistent subsystem implement its own storage manager is that this storage manager is uniquely tailored to the functional and performance needs of that dependent subsystem. The downside, however, is significant. The most obvious disadvantage is that each individual storage manager must pay implementation and maintenance costs, even though they share much in the way of common technology. Although this is becoming an increasingly large development cost, we argue that it actually remains a small problem when compared to the other implications of the independent storage manager strategy. The major cost of this strategy derives from the necessary co-existence of each disparate subsystem. No unified mechanism exists to: name persistent objects, navigate between persistent objects, apply transactions including persistent changes made to multiple subsystems, support a unified persistent object security model, selectively replicate the persistent store, support disconnected operation, backup and restore the persistent store, efficiently support shared objects, support hierarchical storage management of the persistent objects, recover from both system failure and user error, and support non-procedural query and standard query processors. We also note that rapidly declining disk and memory costs, coupled with faster processors and near universal network connectivity, are enabling dramatic increases in online data quantities, yet the persistent stores of many subsystems are ill-equipped to scale to these data volumes.

Our basic thesis is that the information explosion is forcing many subsystem developers to re-examine their storage management infrastructure and that, since many of the requirements are common amongst these storage management consumers, they can be solved using a single infrastructure. This will lead to development costs savings and, more importantly, will lead to fundamentally improved function being made available to dependent subsystems and users, while at the same time yielding a significant reduction in customer administration costs.

We argue that the major server-side storage managers have converging requirements and point out that these requirements could be best served by implementing their function in a single storage manager or a small number of storage managers that have tightly integrated with underlying system services, for example, security, naming, object browsing and navigation, replication, utilities, administration, management, and monitoring. These requirements are summarized in Table 1. For a complete discussion of server exploitation of a single storage manager, see Appendix A.

Client-side systems, however, are much less organized and evolved from a storage management perspective than server systems, and will therefore be the main focus of this work. We will discuss how requirements are changing and what is driving these changes with respect to client-side systems, and introduce the Virtual Object File System (VOFS). VOFS is a single integrated

[1] When we refer to client and server systems in this paper, we do not preclude N-tier systems. For such systems, by client we mean the end-user system, and by server we mean all other tiers.

- *Uniformity of Operations and Administration Requirements:*
  1. Object naming model
  2. Object navigational support
  3. Non-procedural object query support
  4. Transactional support
  5. Security model
  6. Administration, management, and monitoring
  7. Utilities supported
  8. Support for resource limits and quotas

- *Functional Requirements:*
  1. Support for replication and disconnected operation
  2. Support for recovery from system failure and user error
  3. Support for versioning
  4. Support for properties (metadata storage)
  5. Sharing and concurrency control model

- *Performance Requirements*
  1. Performance
  2. Scalability and parallelism exploitation

Table 1 Storage Management System Requirements

data store that supports the advanced functions summarized in Table 1, yet still supports legacy interfaces, allowing all existing applications to continue to run unchanged. We argue that the VOFS can help increase programmer productivity by raising the level of abstraction for storage management, significantly increase user productivity by offering function previously unavailable, provide unique support for disconnected operations, and substantially reduce client-side administration costs. The reduction of administration costs are perhaps less exciting than the new function made available but, in the end, with 28% [49] of a client system total cost of ownership at stake, solving this problem might be the most important contribution.

2 Client-Side Storage: Revolutionary Unification

Client systems typically store personal data that is either produced by the user of the system or placed there for the exclusive use of the system user. Fairly recently, a storage management problem for client systems has emerged. This increased data volume has been driven by rapidly improving disk technology and an increase in the amount of information available to the client system through the near-universal inter-connectivity of the Internet. As a result, client storage managers face requirements very similar to those summarized in Table 1 for server storage management, except that client systems typically support fewer users, somewhat less online storage, and programs with fairly simple storage requirements. In addition to the storage management problem, it is well documented in both the trade press and the popular press that personal system administrative costs are out of control.

After reviewing the changing client computing environment that is driving the need for a new client storage model, we will introduce a proposed solution. This solution will address the problems mentioned above, storage management and system administration, through the Virtual

Object File System (VOFS). In the following sections, we will motivate the need for this with a detailed discussion of the client environment and its projected needs.

2.1 Current Problems and Changing Demands

2.1.1 Rapidly expanding online storage

Over the last decade online disk storage capacity has increased a thousand-fold [66], somewhat faster than the performance increase in processor technology over the same period. With data capacities increasing faster than the performance of the processors [66] that search them, and recognizing that most search algorithms are sub-linear in time, we begin to see an ever-larger storage management problem on client systems. Assuming linear scaling of search time over the amount of data searched, a 5-second search scaled up over a decade of online information growth would take just over an hour. Clearly the difference between a five-second and a 10 seconds search is immaterial, but as search times approach an hour, a better solution needs to be found. Personal storage now faces many of the storage management problems faced by large database management systems. This is a fairly recent phenomenon.

2.1.2 The Internet makes more data available

The fact that there is much more data on a client now than there was even 5 years ago has a significant impact on the client storage problem. However, an even larger impact is felt when client systems are connected to the Internet, a virtually cost-free publication media. Rather than simply the 10^3 increases experienced through improving disk technology, we now see a further increase in online data quantities of 10^7, fueled by the estimated 12 million connected hosts [14]. Clearly, the local client information store must be loaded, indexed, and administered. Further, for performance reasons and to support disconnected and remote operation, some of the non-local data must also be cached, indexed, and managed locally at the client as well. The management of the local cache is a much larger and more complex problem than the management of the local data alone, yet the local data management problem has never really been solved satisfactorily. Examples of the weaknesses of current storage management solutions are the absence of a single query language to search all local data storage managers, and ever climbing client system administration costs.

2.1.3 Information Integration and Administrative Cost Reduction

Information integration is becoming increasingly important. Years ago, businesses discovered that operational data was enormously valuable if placed online to allow a single query language to access it. Data warehouses are now very common. We're beginning to discover that the same is true of personal data such as mail, schedule, personal documents, presentations, etc. If a user were able to query all this client resident data via a single query interface, considerable value could be realized. For example, "what have I accomplished over the past 7 days?" could be answered by querying all mail read and written, all documents produced, and all meetings attended during this period. With current systems, many resource managers such as mail stores support query, but only through special-purpose, application-specific interfaces. Other resource managers, such as calendars, often support no query interface whatsoever. Further, integrating naming, navigation, query, transactional support, security, and quotas is a clear requirement. Perhaps even more important is the reduction of administrative costs made possible by depending upon only a single resource manager, or if more than is used, exploiting a single interface for all administrative and monitoring functions.

2.1.4 Portable and Disconnected Operation

Two forces are driving the requirement to support portable and disconnected operation. First, client systems are no longer dependent upon locally stored data only. They are now susceptible to the combined failure rates of all networks and all systems on which they depend. Users of NFS will no doubt agree that unintentional disconnection is a fairly common occurrence. What is needed are systems that can continue functioning through these events. Secondly, intentional disconnections are becoming more important as the range of applications extends to the computer automation of mobile tasks (for example meter readers and package delivery) and to increased usage of conventional applications, such as email, in non-traditional situations such as air travel.

Considerable excellent research has been done on caching file systems supporting disconnection by the Coda team at Carnegie Mellon University [27], [63], [64], [65], the Bayou Architecture at Xerox PARC [17], and the cache consistency work at the University of Michigan [30].

In our opinion, this work is excellent and has progressed sufficiently far as to be integrated into commercial systems. However, the research has been focused on file systems. Client systems typically depend upon mail, calendar, web, net news or collaborative information source, and often database access, in addition to file system access. If we choose to solve disconnected operation separately, we would either need to solve it in every resource manager used by the client or to integrate the storage requirements of all client applications into a single store and solve the disconnected operation problem just once. Apart from the immediate appeal of being less expensive to develop, a more important feature of a single disconnected data access solution developed for all data is that it would be much simpler to administer. Having N systems supporting disconnected operation separately would require N sets of terms to be learned, N mechanisms to provide access pattern hints to the file system, N sets of configuration parameters, and N cache coherence resolution mechanisms. When there is a system failure, N-1 more sub-systems may be at fault, making diagnosis and resolution much more difficult.

2.2 Current Client Application Storage Requirements

Prior to introducing a solution to the problems we have described, it is important to understand the storage requirements of current client applications to ensure that we have proposed a sufficiently functional solution. In this section, we will analyze the client application storage requirements across the major application domains.

1. *Web Browsers (HTTP clients):*
   Web browsers cache temporary files in the client file system and currently have fairly simple persistence requirements. However, if we were able to access both files and web pages uniformly [52], with local temporary files being just replicas of a remote server's web pages, then disconnected operation would be supported without further effort.

2. *Desktop Productivity Applications:*
   Most desktop applications depend directly upon the file system for all persistent storage. Examples include Lotus 123, Lotus Freelance, Microsoft Word, and Microsoft PowerPoint. Some of these applications, for example Microsoft Word, use OLE Structured Storage [9] to store multiple file objects in a single file system file. See figure 1 for an example of OLE Structured Storage file.

   OLE Structured Storage is essentially a file system within a file system. Note that within the single file system file shown in figure 1, there are actually 7 streams, each of which is

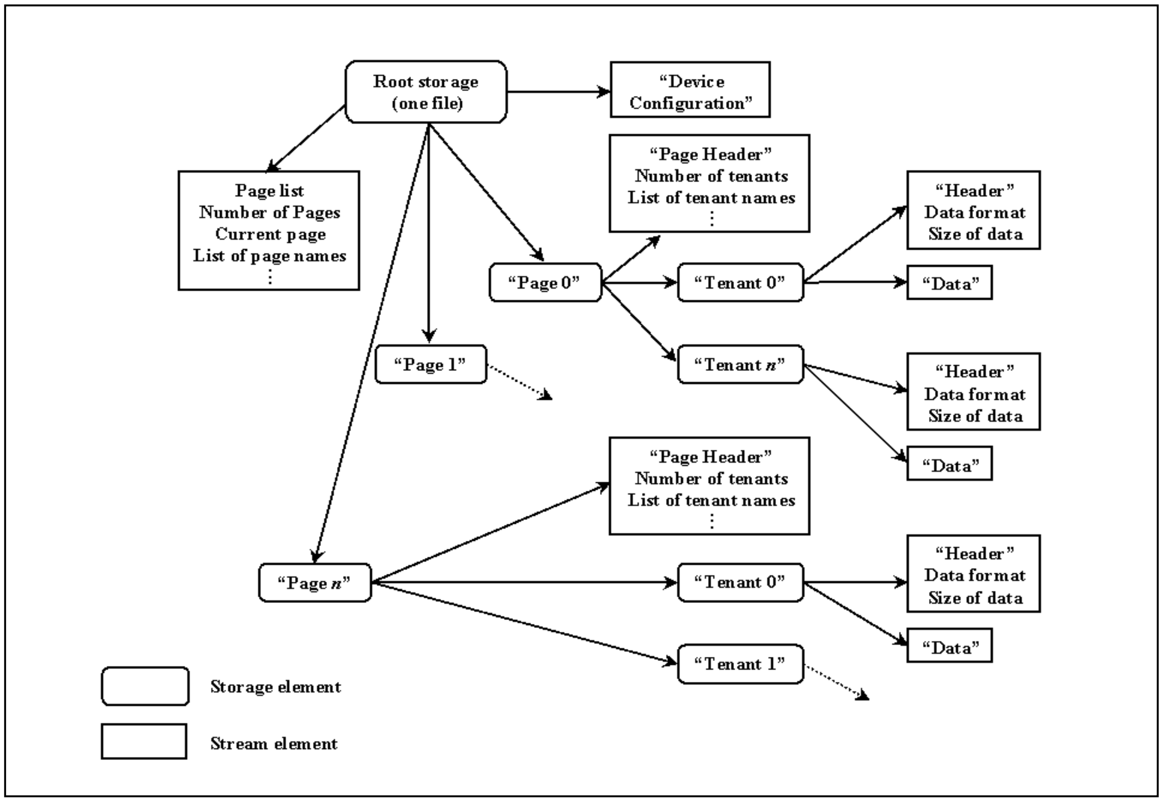


Figure 1 OLE Structured Storage

essentially an independent file. Although the interface defined on streams is somewhat different that the interface supported by the file system on an individual file, the supported function is much the same. The prime motivator for the creation of OLE Structured Storage was the requirement for a single compound document made of possibly many sub-files. This could have easily been implemented within the file system directory hierarchy, but the designers felt that users wanted to be able to treat a compound document as a single file. So from a file system utility perspective, a compound document looks like a single file, but from a data access perspective, it looks like multiple independent stream files. Therefore, when considering desktop applications, whether they use files or structured storage, the actual requirements are much the same as those offered by any modern file system. Desktop applications implement persistence by simply storing each persistent object in a file.

3. *Mail Clients:*
   Mail clients often depend only upon the file system. For example the UNIX MH Mail [60] and Microsoft Internet Explorer mail [53] clients both use the file system directly for the storage of objects. More evolved mail systems such as Lotus Notes and Microsoft Exchange actually implement proprietary data stores to store mail at the client. Clearly, from an implementation complexity standpoint, simply using the file system for persistent storage is the easiest. However, modern mail systems must support disconnected operation and bi-directional replication, and must be able to recover from software and hardware failures at both the client and the server on a transaction basis. Increasingly, mail is being viewed as a mission critical application. What was once a simple application, with limited to no client-side storage requirements, now requires much of the function of a modern database

management system. As a consequence, mail clients are increasingly building advanced, special-purpose, client-side data stores.

4. *Internally Produced and Other Miscellaneous Applications:*
   It is difficult to further classify the plethora of client-side persistent applications that fall outside of mail clients and desktop productivity applications. However if we look at the storage infrastructure on which these are built, most are implemented directly upon the file system, while some of the remaining depend upon either an object or a relational database management system.

2.3 Towards a Solution

We argue that all client-side applications are being driven by user requirements, at different relative speeds and often not along the exact same path, towards needing to support a common set of requirements. Specifically, all client-side applications need to support disconnected operation, recoverable storage, the ability to change clients without data loss (depends upon replication), recovery, versioning, atomic update and transactional integrity, data size scaling, and to require either zero administration or, at most, one way to administer all the persistent stores on the client. Further there is a need for uniformity in client-side persistent object naming, security, navigation, and non-procedural query. In addition, client administration costs must be brought under control. So at the same time we face far more complicated storage management systems driven by these requirements, we also need to simplify client system administration.

We see three possible solutions to these problems:

- *Each application implement a proprietary solution*
  All of these client applications could satisfy the functional requirements listed above by implementing a proprietary database-like layer. However, unless each application depends on the same uniform set of services, no progress will be made towards unification of naming, security, navigation, and non-procedural query. Further, if client administration costs are to be reduced, then we will need to see some unification in that area also. And, once all this work is done, and all storage managers used by the client support a unified administrative framework and all advanced function has been added to these applications, can the memory footprint be controlled? And if their function is identical and administration is identical, why have more than one? In view of the cost and complexity of satisfying these requirements in each application, we would argue that this is not a practical approach.

  Another argument against this approach is that if application developers are freed from worrying about storage management complexities, then functionally richer but less expensive applications will emerge. Further, some storage management problems are sufficiently complex that they likely can’t be affordably solved in some applications. Support for disconnected operation is a great example of one such function. In just the same way that the operating system has taken over some bookkeeping functions that each developer originally needed to solve, we believe that considerable savings can be realized by providing a higher level of abstraction for the storage of persistent data. The advantages of making it easy to write better applications by providing better building blocks surround us. The WWW is an ideal example.

  Prior to the Tim Berners-Lee invention of HTML and the HTTP protocol [5], internet use was primarily restricted to those with good computer skills, most of whom where programmers, academicians, or students. The availability of HTTP as a base infrastructure made much more information available to less computer-literate users which fueled the development of ever-simpler browsers, which were so easy to use that many more users came online, which fueled massive corporate investments in putting information online, which again fed back into bringing more users online. This is a classic example of a positive feedback cycle [66]. This same principle applies equally to client side storage management. If it's easier to write information-intensive applications, then more are written which brings more users, which again fuels more development investment in improving these tools.

Most, if not all agree that a high level data access interface abstraction is required. Many have been proposed, with Microsoft OLE DB [48] being one of the more recent, for solving this problem. Clearly OLE/DB and similar approaches are necessary and make a significant contribution in that a uniform access to legacy data stores is provided. However, we've argued throughout that considerable advantage can be realized if all the data is actually stored in a single integrated storage system. It's that next step that we are attempting to make here.

- *Multiple Storage Managers With Integrated Functional And Administrative Interfaces*
  One might argue that a complete integrated solution, as proposed for the server-side in Appendix A, is overkill for many applications. Essentially, it is easy to agree that storage-intensive client applications can easily be implemented on a commercial database management system just as we suggest for server storage management. But non-data intensive applications clearly don't need most of the function provided by the commercial data management system.

  Therefore, the solution might be to depend upon several storage managers. We agree that many non-data intensive client applications can be hosted upon a file system satisfactorily, however, this is the wrong question. Rather than asking, "Is the file system sufficient?", we ask "Is there a way that these applications can be hosted upon a single integrated storage management system without consuming excessive resources?" As we observed earlier, there are considerable reasons to depend upon a single storage manager from increased function offered to the user, through much reduced administrative costs. So, if it's possible to use a single storage manager, it's the right thing to do.

- *Single integrated storage manager*
  This is perhaps the most radical and, if successful, the most elegant solution, so we will focus on it in this paper. However, it should be remembered, the goal is increased user function with reduced administrative costs through integrated storage. Integrated storage itself isn't the goal so, if we conclude that some aspect of the client storage problem is not addressable in a single integrated data store with current technology, we should simply fall back to providing the second option, several storage managers with integrated functional and administrative interfaces

  This approach costs less from a development perspective because, by delivering advanced storage management function in a single common mechanism, the work is done only once. However, many excellent ideas have failed in the market place by forcing adopters to forget the past and discard their legacy systems and investments. So it is imperative that any integrated storage manager support significant legacy APIs.

3 The Virtual Object File System

The challenge is to provide a single integrated data store which supports the advanced functions described earlier in the paper, but also supports legacy interfaces and allows all existing applications to continue to run unchanged. We propose the Virtual Object File System (VOFS) as a solution to this problem. VOFS is a single integrated data store that supports the advanced functions summarized in Table 1, yet still supports legacy interfaces, allowing all existing applications to continue to run unchanged.

In VOFS, we clearly must support the legacy storage interfaces and we need a storage subsystem that is both extensible and functionally rich enough to be able to add new interfaces if they are sufficiently important. We think that VOFS meets these goals and we'll look in more detail at how it will support file system access, SQL access, and OLE Structured Storage. Although VOFS could be built from first principles, we believe it would be easier to start with an existing relational database management system. From this system we'll be re-using the storage manager which, after some extensions are added, will be the VOFS storage manager. And, we'll take the SQL Query Processor and use this as the SQL query processor for VOFS based queries.

In Appendix A, we demonstrate that an integrated storage manager could support the majority of server-side applications from a functional perspective. Where some doubt may linger is on the performance of such a system. From an SQL perspective, performance is not a problem. The VOFS system will perform as well as the RDBMS from which the query processor and storage manager were extracted. However, the VOFS file system interface will be much more difficult from a performance perspective as it must compete with the path length of typical lightweight file systems implemented directly within the operating system kernel. As a consequence, we have adopted a process model for VOFS very similar to that of the file system (and a fairly significant departure from existing relational database management systems), however we will still need to return to the performance question.

The over-riding design principle is that the system must require as close to zero administration as possible, so before looking at some example APIs and how we would support those APIs, we'll look at the VOFS approach to client system administration.

3.1 VOFS Zero Administration Objective

It is well documented in both the trade press and the popular press that personal system administrative costs are out of control. A 1996 study indicates that 28% of the total cost of ownership of a desktop system is administrative and technical support [49]. A much greater concern highlighted in this same study is that 56% of the cost is in "end users wasted time due to system failures as well as unproductive activities attributed to the extensibility of today's PC environment". A knee jerk reaction to these problems is to propose that users want low function, non-extensible dumb terminals [57]. These low function dumb terminals solve the administrative problems by removing useful function. Further, this solution is completely dependent upon a communications connection. Without the connection, the computer is useless, effectively preventing portable operation.

We agree that these administrative costs must be addressed. One of the driving forces behind the VOFS design is to provide the rich, personally empowering computing environment that is just becoming possible with client systems today, without forcing people to understand the devices. We believe that there is a need for high-resolution displays with powerful graphical processors, rendering information on high-bandwidth 3-dimensional displays. There is a need for powerful processors in support of more natural input devices such as speech or gesture. There is a need for large memories to support the high-powered information appliances that we are describing. Further, even with these large memories, there remains a need for even larger disk drives. Irrespective of Moore's law improving memory price-performance every 18 months [66], depending only upon memory is not cost-effective, and moving-media secondary storage is still required.

The real problem is that system users don't want to have to understand what parts these devices contain, what interrupts these parts are using, and most notably, they don't want to install software. They simply want to use these systems. Rather than reducing the complexity by reducing the value of a system, technology should be used to continue to increase the value of personal systems, while at the same time remove the requirement that a user understand the device. For the most part, the automotive industry has achieved this goal in that people just use cars. Complexity is added every year, and users get value from this complexity, but typically don't understand how it works. Few people understand how the windshield wipers are able to turn on automatically when it begins to rain, or how the headlamps are automatically activated at dusk, but they are willing to purchase these features. With VOFS, we attempt to address the administrative costs without reducing the capability of the underlying personal system.

3.2 The VOFS Model

The VOFS design combines two lines of research: 1) integrated storage, and 2) caching file systems supporting disconnected operation.

Those who are familiar with the work of Satyanarayanan and others on the Coda file system [63] will recognize much of our approach. Coda implements a locally-caching file system that supports disconnected operation. We extend the Coda design by combining it with integrated storage, yielding the advantages of the Coda support for disconnected operation to all storage sub-systems implemented in VOFS. There clearly are some storage requirements that can't be met by VOFS, such as high-volume, continuous media and real time database systems, however we believe that we can efficiently store the bulk of the data found on most personal systems. And, later in this section, we show how VOFS supports a variety of storage management applications, including SQL access, unstructured file access, structured file access, and OLE Structured Storage (compound document storage).

The world we envision is one where computers come with different features and capabilities, available at different price points, just as is the case with automobiles. These computers come with sufficient software in ROM to be able to network boot and download system software and nothing more. An Internet Service Provider will accept connection from this machine and, once connected, will make available that user's "environment". This environment includes whatever system software the user depends upon and a cache of the most recently accessed programs and data. *Note that the client system contains only a local cache of some of the data on the server so, by definition, the client system is always backed up and capable of recovering from system failures and user errors without administrative work on the user's behalf.* All application programs are available on the network. Some could be free, some could be charged by use, and some could be sold with a one-time charge.

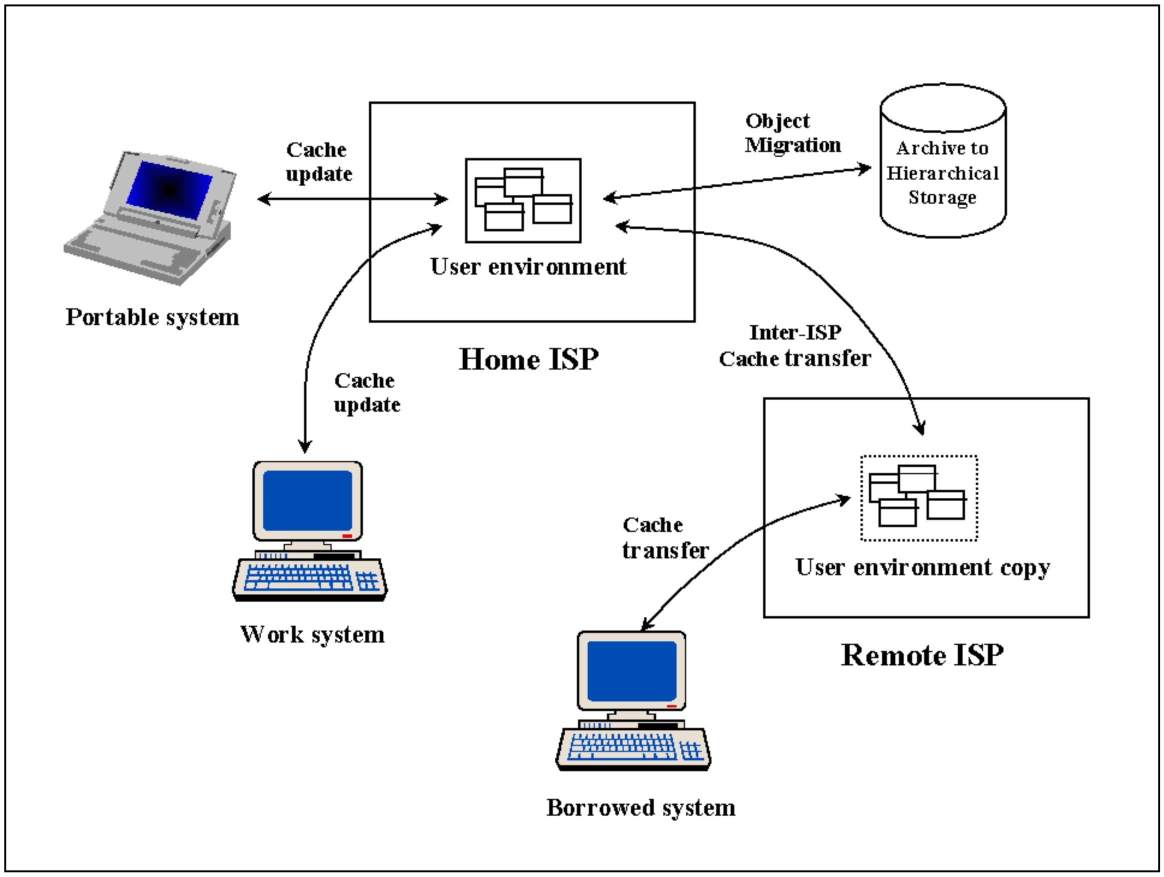


Figure 2 VOFS User Environment

A potential VOFS user environment is show in Figure 2. In this example we have two Internet Service Providers (ISP) shown. The top ISP is the ISP chosen by the user to be their *home ISP* at this point. This ISP is responsible for maintaining all user data and, on request providing it to other *remote ISPs* (still encrypted) when the user accesses the Internet via a different ISP (perhaps when using a guest system at a different location). The remote ISP's have no responsibility beyond delivering the users cached environment during use. They may choose to maintain a cache of the user's data or to discard it.

The home ISP is responsible for maintaining the master copy for the life of the user. The ISP includes a file in the client local cache based upon how recently it was referenced. A file that has not been recently referenced will not be included in the client local cache, and may be moved to less-expensive storage by the ISP. When the client references such a file, it will be faulted down to the local client cache. A user can change home the ISP at will and, when it's done, their complete environment will move to the new provider.

The VOFS client cache approach supports the following uses:

- *Portable and disconnected operation*
  All data and programs used in the recent past are cached locally allowing a user, for example, to pick up a system at any time and head for a taxi to continue working while disconnected. We see intentional disconnection being used in environments where connection back to the network is impossible, perhaps due to remote location, or when connection back to the network is prohibitively expensive, inconvenient, or unnecessary. Over time, we expect that

cellular, PCS, and other forms of radio communications will permit connection back to the network almost regardless of location. Yet, the connection may be expensive, so the ability to disconnect for long periods and work out of the cache, only reconnecting on a cache miss, could save substantial communications costs.

- *Continued operation through network and server failures*
  Just as a user could intentionally disconnect and continue working, they can also continue working through a unintentional disconnection caused by server or network failure.

- *Improved program and data access performance through local caching*
  When using a slow communications path or when using a communication path where the quality of service has a variable charge based upon guaranteed response time, working from a local cache will allow a user to see much faster effective response times. This might allow much cheaper connections to meet their needs.

- *Self installing and self personalizing*
  When a new system is purchased due to an upgrade, theft, or client system failure, in this proposed model, it comes standard with ROM-based software capable of network connection. Once connected to the user's Internet Service Provider, the user's system software and user environment including recently accessed programs and data are recalled on the local system. Even though the new system may require different devices, this change could be handled transparently, since the new dynamically-loadable device drivers are just files that will be faulted down to the local system file cache while those that are not longer needed will eventually age out of the local client cache.

  This is a fundamental improvement over the current situation, where the user must first backup the existing system to some large-capacity storage media, then re-install any applications, and finally restore the original data.

- *Identical environment across many different systems (and support for guest systems)*
  The client system only has a cache (a copy) of the user's environment, system software, programs, and data. As a consequence, if the user uses a different system at home and at work, both systems can support identical user environments. Further, if the user is visiting a branch office in a different city, they can use any desktop in that office by simply connecting to whatever ISP is used in that area and authenticating. If this is not their home ISP, an inter-ISP user environment transfer will be done in much the same way that a bank card can withdraw cash from a different bank in a different city. After the inter-ISP transfer, if required, the user's environment trickles down to the local personal system cache.

- *Data security*
  There are two forms of data security in question: 1) prevention of data loss or destruction, and 2) prevention from unauthorized use or viewing. Prevention of or minimization of data loss or destruction is a side effect of the local system only containing a cache of the user's environment. Another copy is always available at the ISP to which they are connected or will be moved there when next connected. Encrypting data both at the ISP and locally prevents unauthorized use or viewing. If a system is lost or stolen, if the network is monitored, the data is not readable without the users private key.

- *All data ever experienced is always available*
  Each user's ISP maintains all data that has ever been referenced. If it has not been accessed

recently, it may no longer be stored in the local PC cache, but still stored at the ISP. If it is not referenced for long periods of time at the ISP, then it may be moved into progressively less expensive, lower-performing media by hierarchical storage management software. *But it always is there for potential reference. Nothing is ever lost or unavailable.* Even more important, users are freed from having to find files that are no longer needed and erasing them to free space.

3.3 VOFS Process Architecture

Figure 3 shows the VOFS process architecture. We have implemented the VOFS storage manager in much the same way that a typical file system is implemented: in the (protected) operating system kernel address space. The SQL query processor and run-time system are outside the kernel, running in the same address space (process) as the calling program. The file access DLL provides a conventional low-function file store interface for legacy applications and for those applications that are best implemented through such an interface.

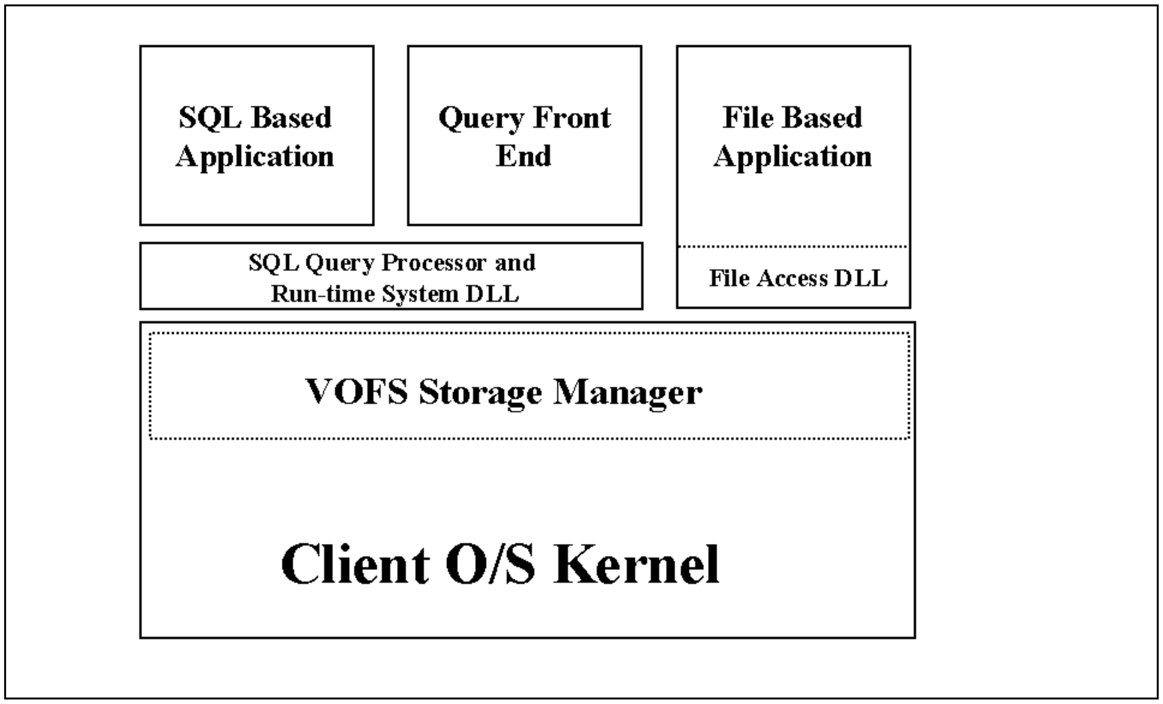


Figure 3 VOFS Process Architecture

3.3.1 Storage Manager

We choose to implement the storage manager in the kernel address space so that it can be accessed from the file system interface via a mode switch to the protected kernel address space without requiring a context switch. Again, this is how many file systems are implemented. We choose not to run the storage manager in the user address space for two reasons: 1) we don’t want a failure in the user program to threaten file system data integrity, and 2) the storage manager maintains shared data structures shared between users and different processes, the most important of which is the buffer cache where recent accessed pages are cached in memory. A side benefit of implementing the storage manager in the operating system kernel is that on systems such as

Windows NT, where one half the 32 bit address space (2 gig) is reserved for the operating system kernel, the entire storage manager can be supported with a large page cache without impacting the address space available for the application programs.

3.3.2 SQL Query Processor and Run-time

The SQL query processor and run-time system are implemented outside the operating system kernel to avoid resource consumption problems that are both dynamic and difficult to bound. The memory consumed by the query processor is dynamic, related to the complexity of the query, and difficult to predict in advance. Further, the run time system should exploit multiple processors and, where intra-query parallelism is useful, it may choose to run multiple threads to satisfy a single SQL request. This behavior could be supported within the kernel, but because the query processor does not depend upon shared state between users, there is no compelling reason to do so. And, without a compelling reason, it shouldn't be put in the kernel, since any kernel-based failures will typically bring the entire system down, whereas software failures in user space only bring that process down and have no impact on the rest of the system.

3.3.3 Implementation Alternatives

Understanding that implementing the storage processor in the kernel address space makes development more difficult, and exposes the system to more risk in the event of a storage manager software fault, is there an alternative? One alternative is to implement the storage manager and SQL query processor and run-time in a separate process. This would put them outside of the kernel address space so the system is not at risk, and outside of the user program address space, so that storage manager data integrity isn't at risk either. While this would work, it would require a context switch on every file I/O request, adding approximately 500 to 1000 instructions of overhead. This is clearly unacceptable.

However, an alternative that should be considered if implementing VOFS on a micro-kernel based operating system such as Next [56], Mach [40], Windows NT [51], and Apple's next major operating system shipment, Rhapsody [1], is to implement the query processor, run time system, and storage manager as a protected sub-system, as shown in figure 4. Micro-kernel systems implement most operating system function in independent user space servers. Communications between application programs and called servers is via high-optimized Local Procedure Calls (LPC). On some research systems, the LPC is actually implemented without a context switch and is very nearly the same cost as the mode switch required to request a system service in a conventional operating system.

Regardless of which VOFS implementation is chosen, the VOFS access APIs are the same, as is the implementation of the underlying VOFS function.

4 VOFS APIs

In order to cover the client application domain, VOFS must be capable of supporting a wide variety of storage management applications. In this section, we survey the storage management spectrum, from the simplest and most lightweight unstructured file storage managers, through structured file storage managers, to higher function SQL access. For each, we'll choose a representative example from the class for purposes of discussion. In other words, the UNIX-like file system chosen to discuss the unstructured file-system API could easily be replaced by the

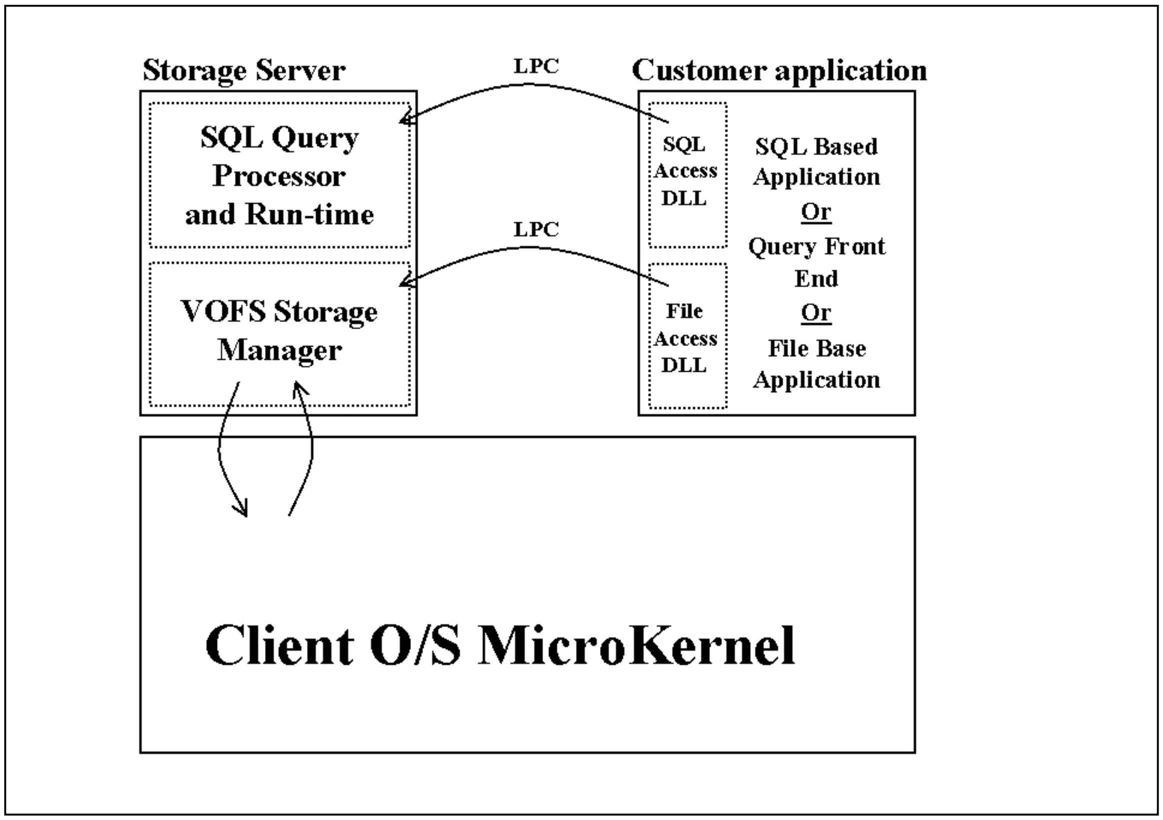


Figure 4 VOFS Micro-Kernel Process Architecture

Win32 file system API, for example. For each persistence class, we have selected an example API as a concrete way of showing how such function would be implemented in VOFS:

- Unstructured file access: UNIX-like file system API and semantics [5]
- Structured file access: Transarc Structured File System [73]
- SQL access: ODBC supporting relational database interface and semantics [45]
- OLE Structured Storage: the compound document support used on Microsoft Windows operating systems (Windows 95, Windows NT, and Windows 3.1) [9]

Our objectives are to implement each persistence API, including the semantics expected by users of this API, in VOFS, and to also ensure that each API shares VOFS metadata and storage through any of the storage APIs. Each storage object must be cataloged in the relational database catalogs to be accessible by the query processor.

4.1 VOFS API: Unstructured File System Design

To solve the problem of making the file store available to the VOFS SQL query processor, it is necessary to impose a relational schema on the file system. We need the following:

- a representation for directories
- a representation for files

- some form of navigation

Technically, additional navigational support isn't required since some relational database management systems support recursion. However, we feel that recursive queries are an unnatural directory traversal abstraction for many file system users. So, although recursion through the hierarchical directory structure will continue to work, we also offer predicates in support of common file system operations.

Designing a file system API upon a relational database management system storage manager is a fairly well understood problem and there are examples of past work. For example, the IBM VM/ESA CMS Shared File System (SFS) [67] is built upon the SQL/DS [33] relational database management system, also available on the VM/ESA platform. The standard VM/ESA file system is built upon a flat name space and it is restricted to read-only file sharing. To offer a shared, hierarchical file system, the IBM design teams chose to build upon the SQL/DS relational database storage manager. SFS is a commercial product in fairly wide usage. An evolved version of this product, the VM/ESA Bit File System, is built upon the same relational infrastructure and supports the Posix file system API under VM/ESA.

Using this example as evidence that a file system API can be satisfactorily supported upon a relational storage manager, we propose a further extension of this approach in VOFS. In the CMS SFS, the file system could not be queried by SQL/DS since the SFS metadata was not represented in the relational catalogs. In VOFS, for each persistent API we implement, we register the metadata describing this stored data in the relational catalogs. As a consequence, all data stored into VOFS by any interface can be queried through the VOFS SQL query processor. And, rather than duplicate all the metadata for each API, we choose to make the relational catalogs, or extensions to them, the primary metadata store for all VOFS-supported APIs.

When designing the relational schema for the file system, we note that we must store file and directory names, and for both, we must store metadata such as creation time, last access time, last changed time, security access control list, etc. A directory entry contains one or more files and directories, so we need a representation of "contains" for the directory. A file entry contains no sub-entries, only data. We note that both a file and directory have names and metadata, and differ only in that directories contain more directories and files, whereas files contain only data. In Figure 5 we show the VOFS schema, which we will discuss shortly. The schema consists of a pair of relational tables (extensions to the relational catalogs) where all file system metadata (everything except the file data itself) is stored.

Files need to be efficiently randomly accessible (for example, they can't be efficiently implemented as a linear linked lists of pages) and sizes of at least 2 GB must be supported. To avoid capacity problems, file capacity should be significantly more than 2 GB and one could argue that 2^64 or 2^63 (positive range of a 64 bit integer) would be wise. The storage system needs to support completely unstructured byte stream files, but should be easily extensible to support record-based structured files. We also need to support non-procedural query against the file system using the VOFS SQL query processor. As a final requirement, traversing the directory structure using the file system API should be no less efficient than conventional file system designs. By this we mean similar instruction path length, with no additional serialization or I/O operations required.

4.1.1 VOFS Schema for Unstructured Files

The schema for representing unstructured files in VOS is shown in Figure 5. The *Vofs* schema can be queried but it is reserved for the file system and new objects can't be created by users in this schema.Thus, *Vofs.FileObjects* and *Vofs.DirectoryEntries* are essentially file system catalogs (metadata). Other schemas may be created through the VOFS SQL interface. In other words, there could be a schema *mydata* and within this schema a table *table1* could be created (*mydata.table1*), but it's not possible to create *vofs.table1* since this schema is reserved for the *vofs* file system. In this design there is only one file system stored in the *FileObject* table, but it would be trivial to introduce a *FileObjectsN* and D*irectoryEntriesN* for each directory tree one wishes to represent.

The *FileObjects* table is indexed on file name. Typical file system access will be via name or, when traversing directories, by `rowid,` which is stored in *DirectoryEntries* (`rowid` allows direct physical row addressing into the *Vofs.FileObject* table). *DirectoryEntries* should be indexed on `DirectoryReference` and we recommend using a clustered index [46], if available, to support index-only storage and access. If a clustered index is not available, an index should be created on `DirectoryReference` and `Entry` to at least get index-only access. The file system root is stored at a known location, rowid 1, in *FileObject*.

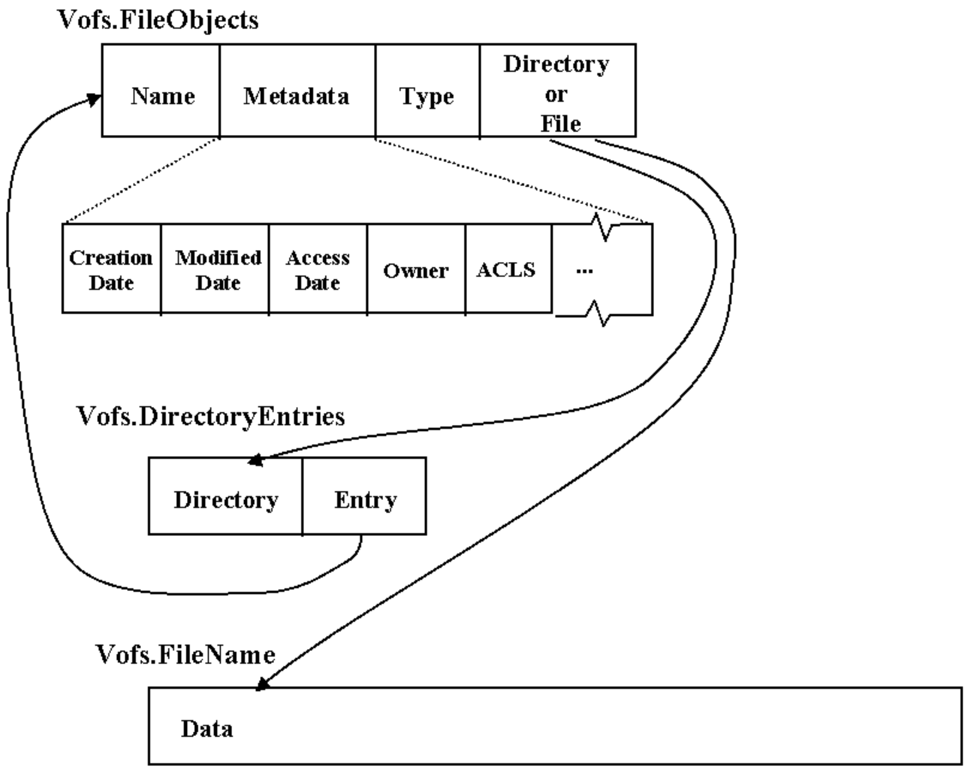
The actual file data is declared to be a field of type *FILE* in the *Vofs.FileObject* row representing the file. However, what is really stored in the row is an internal reference to a separate storage manager object, a table, where the file data is actually stored. This table is cataloged in the SQL catalogs, so it's visible to the relational component of the system. The catalog entry is created using the largest trailing subset of the full path name that will fit (depends on the table name length limit). If this is not unique, then it must be made unique, for example, by suffixing with an ordinal.

Since unstructured files by definition have no internal structure, only a single binary large object column, *data*, is defined for file data table. All the file data is stored in this single row, single column table containing only the blob (the same basic design is used to support structured files in Section 4.2 where the blob is replaced by one column for each field in the structured. Typical database management systems support accessing the entire file, and also accessing arbitrary byte ranges within the file via BLOB access primitives.

4.1.2 Log Management for Unstructured Files

The design that we have chosen is one where all files are optionally under transaction control, but always fully recoverable. The exception is those files stored in the system-defined temporary directory (e.g. `\temp`), which are neither transactional nor recoverable.

We use the storage manager recovery log to ensure that all file system operations are transactional and recoverable. We also log all object references for later use by the cache manager. The cache manager is responsible for keeping recently-accessed objects in the local cache and for evicting objects unlikely to be referenced in the near future (a modified least-recently-used algorithm is used). We use a conventional write-ahead log (WAL) with some minor modifications.



```
Vofs.FileObjects:   (all directories and file names in the VOFS  file system)
      Name varchar(1024)
      Creation_date date
      Modified_date date
      Access_date date
      Other_metadata_fields
      type enumerated type (directory, structured file, unstructured file)
      union {
          data FILE (reference to special table object containing file data)
          directoryReference int (null if a file entry)
       }

Vofs.DirectoryEntries: (1:m mapping from directory: files &directories)
      DirectoryReference
      Entry: rowid (refers to a Vofs.FileObject row)
```

Vofs.FileName: (*one table per file allocated in the file system*)

This table is cataloged and visible through the SQL interface and depending on the type of file it stores, internal structure may be cataloged as well.

Figure 5 VOFS Schema

One potential concern with this approach is massive log growth caused by large changes to file data. For example, it is not uncommon for a desktop application to completely replace an entire file. There are many approaches to handling this problem, including shadow paging [24], adding constraints to the buffer manager such as not over-writing a page until commit time (no undo record needed), or forcing at transaction commit time (no redo record required unless media recovery is supported). Experimentation is required to establish that the system will perform acceptably well using conventional write-ahead logging [24] without implementation tricks to reduce log bandwidth consumption. The good news is that log conserving buffer management and logging techniques are well understood and documented in the research literature [53].

It's important to remember that, although logging results in additional data being written to disk, the write-ahead log protocol allows us to avoid writing the changed data pages to disk at commit time, and to only write synchronously to the log. Clearly the changed data page must eventually be written out, but personal computers have large periods of unused I/O bandwidth where this page can be asynchronously forced to disk.

The log is transparently migrated up to the ISP server either by trickling up during connection or by migrating up on re-connection. Note that it is possible to post-process the log during trickle migration to the server, where undo records would be discarded if the transaction has committed and redo records may be discarded as soon as the changed pages described by the log records are propagated to the server system.

The log is saved indefinitely, as it represents the life history of the user that created it and will be used for many purposes including cache management and to support certain advanced access methods which we will describe in section 5. Also we must keep in mind that storage device capacities are increasing about 60% per year [4], yielding 100 times more capacity every 10 years. So, with rapidly declining storage costs, there will come a time when the administrative costs of figuring out what can be thrown away and what should be kept is far higher than whatever storage is recovered. In *The Revolution Yet to Happen* [4], Jim Gray and Gordon Bell point out that the storage required to store all the text (with a few pictures) read in an average persons lifetime is 60 to 300 GB. Even video, if watched constantly for a lifetime, would only yield one PB. With current technology, it is certainly affordable to keep several years worth of data and logs. And with continuing technology advances, we may be able to keep them for the lifetime of the user.

4.2 VOFS API: Structured File System Design

In the previous section we discussed how various forms of unstructured storage could be implemented in the VOFS system. In this section, we'll look at what extensions would be required to the unstructured file system support to handle structured files. Figure 6 shows generically the types of structuring supported for files. We believe that the support available in VOFS is sufficiently general to allow the support of most structured file system designs. As an example, we'll use the fairly rich Transarc Structured File System (SFS) [73] as an example. SFS defines three structured file organizations: Entry-Sequenced Organization, Relative File Organization, and Clustered-File Organization. For all of these organizations, transactions and recovery are supported. The organizations are built upon different orderings of records within a file. With each file organization, records are described by the user when the file is created, in much the same way that the columns in a relational table are defined when the table is created. Each field is cataloged as a column and each record ends up being a "row" in the data table. Because both the structured file system and the relational query processor share the same

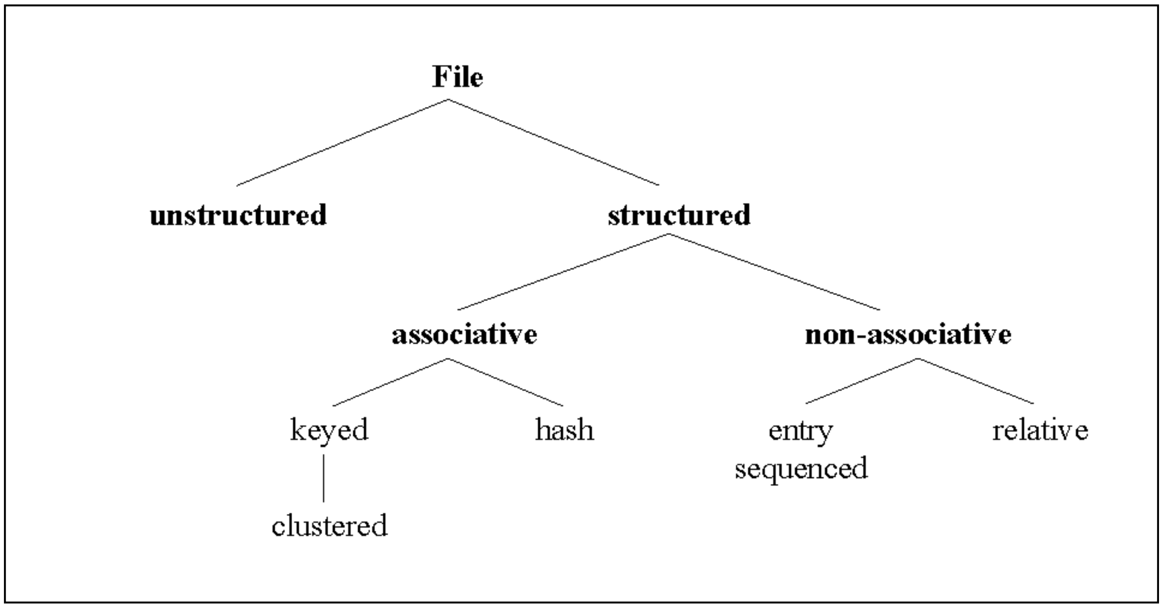


Figure 6 Structured and Unstructured File Types

metadata describing the file, both APIs can symmetrically access the data. All file organizations support secondary B-tree indices on single or composite keys.

The organizations are defined as follows:

- *Entry-Sequenced file*
  New records are always appended on the end of the file and storage is not reclaimed from deleted records until the file is re-organized.

- *Relative File Organization file*
  Records are stored at specific relative record numbers. On insert, a record can be inserted in the first free slot from the beginning, at a specific slot, or appended at the end of a file.

- *Clustered File Organization*
  A B-tree where the leaves of tree contain (key, record) pairs. This is the index structure implemented by Sybase and Microsoft in their database system.

Building on the file organization described in section 4.1 to store unstructured files, implementing the SFS organizations is a relatively straightforward extension. Fields will be cataloged to support relational access to the file as a table. Given that the file is actually just a relational table, we can support the SFS APIs that create secondary indices by simply executing the same code that we would have executed had a “create index” statement come through from the SQL front end. From the relational query processor, any legal SQL query can be posed against the SFS files and they can even be joined with SQL tables. SQL update and delete could also be supported fairly easily, however insert would require some changes to ensure that new records were allocated correctly according to the record allocation constrains of each respective SFS file type.

Since both the VOFS SQL relational component and the VOFS SFS file interface share the same metadata and page formats, the SFS interface can also be used to access tables created by the

VOFS relational component. An SQL clustered index table would appear as an SFS clustered file. A standard SQL table would appear as an entry sequenced file organization, except that only SFS read operations (without much work the bulk of the APIS for adding and modifying data could also be supported – the key problem is that relational tables are unordered so the required modification would be to only insert new rows at the end of the table).

4.3 VOFS API: Relational Database System Design

Since the VOFS infrastructure is implemented upon a relational storage manager, little effort is required to allow the relational query processor to continue to work against this store. But, as we described above, the goal isn't to only put data from different APIs into the same storage manager – we also want to be able to use the relational query processor to pose queries against data stored via the non-relational persistence interfaces. For this reason, when discussing both structured and unstructured files above, we were careful to use the relational catalogs as the metadata store for not just the relational component but for all supported APIs. For example, when designing the unstructured storage interface, we put the record definitions (all fields defined in a record) into the relational catalog column description. We put each file into the relational catalog table definition. As a consequence, all objects in the VOFS, regardless of how they were originally stored, are accessible to the relational query processor.

As mentioned in Section 4.1, we believe that most users will be more comfortable querying the file system using non-recursive SQL queries, however, recursive queries are still supported for those that prefer them and for queries where it seems appropriate. For example, the DB2 Common Server [32] query shown in Figure 7 will return all files contained in the directory tree rooted at '`/dir1/dir2`':

```
with dirtree (pathname, name, subname)

   as

   ((select root.name, root.subname
      from Fofs.FileObject root
      where ROWID=1)
union all
   (select parent.pathname||'/'||parent.name, child.name,
      child.subname
      from Fofs.FileObjects child, dirtree parent
      where parent.subname = child.name))
select name from dirtree
   where substr(dirtree.pathname, 1, 10) = '/dir1/dir2';
```

Figure 7 Recursive Query

However, to slightly simplify typical queries in both the structured and unstructured file systems we provide two additional predicates:

1. *DirectoryContents(FileSystem, Directory):*
   This predicate takes as arguments the file system and the directory of interest. We could omit the file system as an argument on those VOFS implementations where only one file system is supported, for client system ease of management. The predicate evaluates to true for all files and directories contained in the directory named in the second argument.

2. *SubDirectoryContents(FileSyste, Directory):*
   This predicate, like DirectoryContents() above, takes a file system and a directory as arguments. But, rather than returning only those *FileObjects* defined in *Directory*, all recursively contained *FileObjects* are also returned.

Both of these predicates are supported by an additional storage manager access method that returns the RowIDs (RIDs) of qualifying rows. So, although these predicates appear to force full table scans of the *FileObject* table in the example queries below, the query optimizer will typically use the underlying storage manager access method to traverse the directory structure.

The access method support is a fairly simple recursive traversal of the directory structure, returning the RIDs of qualifying *FileObject* records. To support DirectoryContents(), the RIDs of *FileObject* rows contained in the given directory are returned. SubDirectoryContents(), is identical except that all recursively contained *FileObjects* are also returned.

4.3.1 Example Queries

To show that typical queries can reasonably be composed against file system data, and that the resultant interfaces is reasonably easy to use, we will look in more detail at a few representative sample queries and how they would be formulated in the VOFS system.

1. *Find all unstructured files with a certain metadata property (creation date, size>N, etc.):*
```
Select name from FileObjects
   where size>N and type=file;
```

2. *Count all directories in the file system called 'directoryName':*
```
Select count(*) from FileObjects
   where name='directoryName' and type=directory;
```

3. *Find all unstructured files in directory \dir1\dir2:*
```
Select name from FileObjects
   where Directory(vofs, '\dir1\dir2') and type=uFile;
```

4. *Sum the sizes of all unstructured files in or below \dir1\dir2:*
```
Select sum(size) as size from FileObjects
   where SubDirectory(vofs, '\dir1\dir2') and type=ufile;
```

The file system queries presented above are relatively easy to formulate, but we also need to query file contents. For structured files, we can treat the file of records of fields as a table of rows and columns since both are represented identically.

So, to find all records in a structured file, myfile, where the field2 is greater than 12, the following query will work:

```
Select ROWID from myfile where field2>12;
```

4.3.2 Full Text Query Support

The above queries work well for structured files, however we also need to be able to query file contents in unstructured files where we have the file contents represented simply as an unstructured array of bytes. For this we support a full text search access method providing efficient support for full text predicates such as the contains("word") search predicate.

There are three problems with conventional full text index systems: 1) we need to index text in non-pure text files (e.g. lotus freelance files), 2) we need to be able to index and maintain indices on less than all files in the file system, and 3) indices should be dynamically created, updated, and later discarded if not re-used.

4.3.2.1 Full text indices over non-pure text files:

Many files stored in a file system are of known type, but they aren't pure text. For example, Microsoft Word and Lotus WordPro store their respective word processor files in a binary form. For common file types, we use pre-filters that extract whatever text is in the file. For unrecognized files, we have a default binary pre-filter that extracts any known words (words in the dictionary for the current language). The results of these filters are simply used by the full text indexer, but the text itself is not directly stored with the files.

As an additional source of text to index, we also index file metadata such as file name, any user supplied metadata, the creation date, etc. This will help users find pure binary data even when using full text search, provided that the files are well named or that there is descriptive information about the file stored as metadata. Although, clearly other search mechanisms are more appropriate for binary data, such as "containing folder", which is a directory and subdirectory search on "file name".

We now have the capability of indexing arbitrary user data regardless of the data source.

4.3.2.2 Full text indices over sub-sets of the file system:

Few users wish to full text search the entire file system, so it must be possible to index only those files with a certain property [69]. For example, I may be only interested in files created by me. One approach to this problem is to support indices over views. So, if only interested in searching for documents created by "JRH", one could create a view "jrh" as *select * from Vofs.FileObjects where creator='JRH'*. Then a full text index could be created over those files that qualify as members of this view.

4.3.2.3 Full text indices dynamically created, updated, and later discard:

We don't want to force users to create full text indices prior to formulating a query, so we clearly need to be able to dynamically create indices on the fly. We don't want to maintain indices during file update operations since, in most systems, queries are rare whereas updates are common. Therefore we must support update operations to re-synchronize existing indices. We need to be able to use an existing index if an appropriate one already exists and, if no appropriate index is available, to estimate the length of time it will take to create the index and ask if the user wishes this to be done. Since we rely heavily on indices created on the fly, we need to use full text index technology optimized for fast index creation. We can't force users to create an index prior to using it, but we should allow this for those users who know a certain index would be useful.

In the previous paragraph, we said that we should use an existing index if it already exists. Actually determining this is somewhat of a difficult decision, since we may have an index over slightly more files than we are interested in, in which case we must compare the costs of creating an new index with using the existing index and then applying the additional filter predicates. If there is an index over slightly less data than we are interested in, we need to compare the costs of adding the new documents to the existing index to yield the broader index we are interested in, or simply create a new index. Index creation therefore requires cost-based decisions.

Deletion of indices is another operation we don't want to force a user be concerned with. Instead, we depend upon the cache VOFS maintenance system. Just like any other file system object, the user doesn't actually have to delete the object. In the case of user files, they are removed from the local system cache when they haven't been used in some time and stored only at the server. In the case of system created objects such as indices, when they are removed from the local cache, they are also removed from the server but, other than that, the cache management system is unchanged.

We strongly suspect that the most efficient way for the system to manage the full text indices is to keep a single index over the entire file system and post-filter the results of queries against this index down to the set of files interesting to the user that formed the query. However, it depends a great deal on the costs of the indexing and searching operations and may change depending upon the usage characteristics of the users. So rather than defining this as the solution, we have chosen a cost-based solution that should eventually stabilize on the optimum set of indices. If we were to find that, in a large percentage of the cases in real use, a single global index was the right answer, we could remove the cost based indexing component from the design and instead rely exclusively on a global index with post filtering. And, if this turned out to the be right answer, we would no longer require indices over views in our design.

4.3.2.4 Full Text Query Examples

We now have a full text index system defined where arbitrary queries can be submitted. Here are a couple of example queries:

1. *Find all unstructured files in the file system containing the word "interesting':*
```
Select name from Vofs.FileObjects
   Where contains(data, 'interesting') and type=uFile;
```
2. *Find all unstructured files in or below the subdirectory '\dir1\dir2' containing the text 'find this':*
```
Select name from Vofs.FileObjects
   where SubDirectory('\dir1\dir2')
   and contains(data, 'find this') and type=ufile;
```

With this design, we support the storage, manipulation and traversal of unstructured file data, structured file data, and relational data in a single data store. And, since all data access APIs use the same metadata defined in a form expected by the relational query processor (relational catalogs), we can submit non-procedural queries against data store or updated by any of the supported APIs and we can join data stored with one API with data stored via another API.

4.4 VOFS API: OLE Structured Storage Design

As a final example of the flexibility and extensibility of the VOFS storage manager, we consider OLE Structured Storage. As mentioned in Section 2.2, OLE Structured Storage is essentially a file system within a file system. Via the file system APIs, an OLE Structured Storage file looks like a single file but through the OLE Structured Storage APIs, that file contains a hierarchy of *substorages* (directories) and *streams* (unstructured files), as shown in Figure 1. We can exploit our existing file system schema introduced in Sections 4.1 and 4.2 by adding a new structured file type, *storage,* that we represent as a directory. Below each *storage* we have one or more *substorages* represented as directories, and *streams* (files) represented as files (see Figure 8).

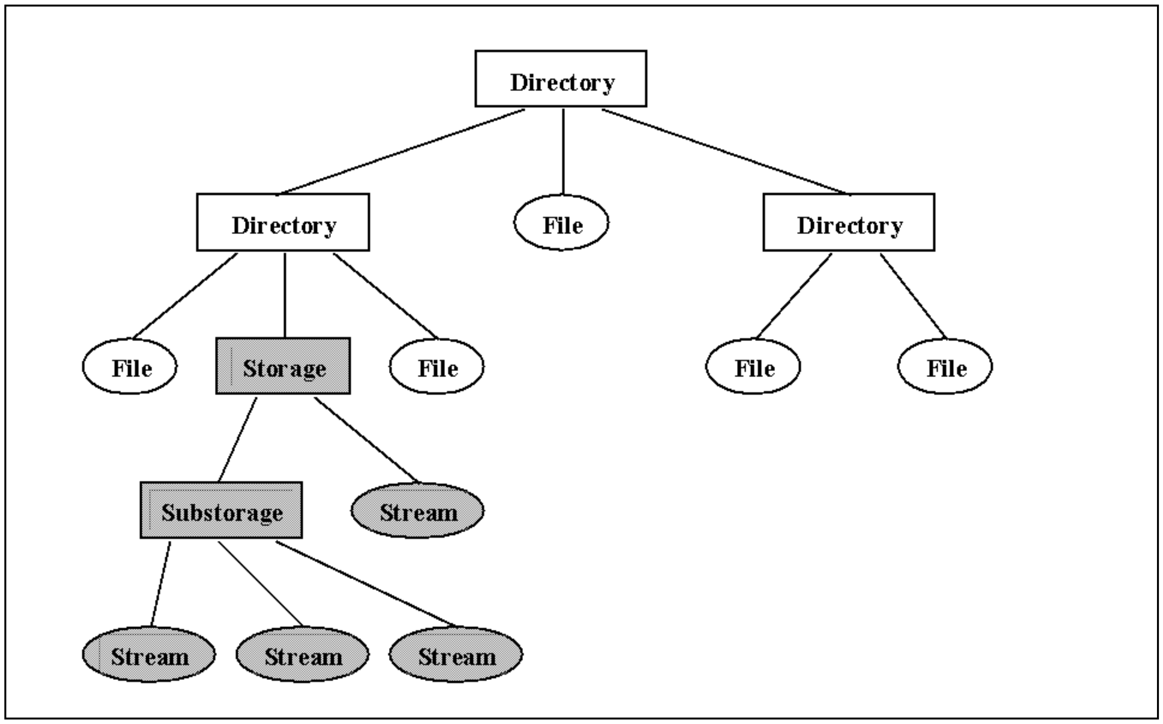


Figure 8 VOFS OLE Structure Storage Design

Essentially what we are doing is introducing three new file system objects: *storage, substorage* and *stream.* The former two are stored as special forms of directories and a *stream* is stored as a special form of a files. Through the file system API, a *storage* appears to be a file whose contents are the linearized contents of all *streams* and *substorages* contained within the *storage*. When viewed through the VOFS OLE Structured Storage API, each *substorage* and *stream* is independently addressable.

5 Further Work

We've argued that integrating the storage managers on a given platform can substantially reduce administrative costs. And, with a single storage subsystem supporting all storage consumers, development investments in this storage subsystem can be amortized over more client applications. As a consequence, applications with fairly simple storage requirements, such as calendaring, can profit from advanced storage management function that would be unaffordable to build on the basis of calendaring alone. We've presented a design for one such system, the Virtual Object File System (VOFS), that offers both an integrated storage manager and incorporates much of the work done in research systems such as Coda in support of portable and disconnected operation.

In the body of the paper we discussed many of the advantages that a VOFS-like approach would offer, however, the VOFS infrastructure could form the basis of future extensions or further research. We discuss several in this section:

5.1 New Access Methods

VOFS support of a *life history*, essentially a log of all object additions, updates, and accesses can be used to support some interesting new access methods.

5.1.1 Past Navigational Paths

The observation here is that there exist many tasks where a user knows where to start but isn't exactly sure how to actually get to the information that they seek despite the fact that they have found it at some time in the past. For example, if an employee wants to look up how to file a dental claim record, they might not remember exactly where the form is but they do remember finding it close to a year ago by starting at the company human resources web page. To support this fairly common access path, from any given page or document, we can show a user all access paths that cross that document as a sequence of hyperlinks. For example, if it was a web page, the user could be shown all sequences of web pages that they have traversed in the past, extending from the current one.

The user's *life history* log ensures that we see all object accesses and, as a consequence, can record all past navigational paths and keep indefinitely all those that are used more than once during a given period, perhaps a year. This form of index was described by Vannevar Bush in *As We May Think* [10], where he called the access method a *trail* and the device used to store information an *Memex*.

This form of index can help people find objects where they remember some object that they accessed in the past that helped them find the object in question.

5.1.2 Temporal Access

Human beings often remember chronologically-related events (including viewing an object), despite the fact that these events may be totally unrelated. For example, we might remember that we came across what looked like a paper on data mining and its applications to basketvall when researching this paper, and just moved on, as it wasn't actually what we were looking for. If we later want to find that paper, it may be difficult, since we didn't really look at the paper, so we know little about it other than the fact that it is out there somewhere. We may be able to find it using full text search on data mining and basketball, but frequently this produces far too many false positives.

Another approach is to use the life history to produce a list of objects that were accessed during the time when this paper was researched (a time period) which should produce only a few hundred or, perhaps, a thousand objects that can be very quickly full text searched. And, given the small number of objects being looked at, few false positives would be expected.

Temporal access is another dimension on which a search can be conducted and is another way to narrow the search space without eliminating the information for which we are searching.

5.2 Advanced Functions

Since all objects are stored in a single data store, we can add support for many of the extensions available with commercial database system such as referential integrity, constraints, triggers, alerters, and other active database extensions. The difference is that these business integrity rules and active data extensions can be supported across all objects in the user's known universe rather than only across those objects stored in a given commercial database management system.

5.3 Extensions to Caching Semantics

The life history log is maintained in support of storage manager recovery and cache management. The cache manager can use conventional cache management techniques to analyze the log and determine which objects to keep in the local cache. However, because we maintain the log indefinitely, it would be possible to apply data mining algorithms [1] to the log to do a much better job of predicting when an object will be accessed and, as a consequence, have it ready in the local data store prior to this predicted access. These cache management improvements allow the system to appear to access data more quickly and to suffer fewer cache misses when operating in disconnected mode. The data mining software could look for clustering relationships (objects normally referenced together), non-sequential patterns (for example, a user referencing a time reporting form every Friday but typically only on Friday) that are typically missed by LRU based cache management systems.

The cache manager is responsible for tracking past object accesses and to predict future accesses. The most conventional use of this information is to manage the cache and attempt to ensure that all objects accessed are cached locally. However, this same information can be used to dynamically create and remove indices. For example, if the cache manager is also tracking queries (rather than just objects accessed), it's possible to determine that it's worth creating an index to support a common query or to remove an index that is rarely used.

The cache manager, in its most extended form, is essentially a user agent responsible for monitoring user activity and evolving the work environment to better suit these activities. In essence, as cache manager function is extended, it becomes more like a user agent and much of the research on user agents can be applied to this problem as well.

5.4 Commercial Extensions

If we assume that the VOFS cache management and logging system described in the body of the paper is secure, then several commercial extensions become possible. For example, software could be charged for by use, since the VOFS life-history log tracks all object (including program) accesses. Any software can be made easily available on the web and run at any time with the VOFS system supporting per-use and one-time charging mechanisms.

In Section 5.1, we discussed how the life history log could be used to support different access methods such as Temporal and Past Navigational Paths. Essentially we are exploiting past user experiences to help them find things in the future. Extending this idea, we could sell access paths of interesting things. For example, if one was interested in studying nuclear medicine, the life history of an expert in nuclear medicine (filtered to include only pertinent objects) could be sold and treated as a not yet experienced bit of history, available for traversal by the purchaser.

And, of course, the flip side is also possible, a user could choose to sell portions of their life history. For example, advertisers might be willing to pay to learn more about the interests of affluent consumers. Clearly these life histories would have to be filtered to prevent loss of excessively personal information, but they would certainly have value for those that might choose to sell them.

6 Conclusion

We have argued that a single integrated storage manager supporting the persistence requirements of client system applications can reduce resource consumption, substantially reduce administrative complexity, and provide advanced function such as transactions and recovery. The basis for this argument is the observation that a single storage manager can support multiple persistence APIs, including file and relational access, and that many applications could be improved if advanced storage management function, such as transactions, was made available to application developers. Implementing these functions in each application is 1) often too expensive to be justified, 2) requires special skills, 3) if each application solve the problem independently, the combined memory, disk, and administrative resources consumed are far higher.

We have proposed the Virtual Object File System as a solution to the integrated storage problem. And we have shown how Coda-like caching support can be added to VOFS to support portable and disconnected operation across client applications. The resulting integrated storage manager can be further exploited to substantially reduce the client system administrative costs, freeing users from having to load software, update software, index data, and remove data no longer in use.

VOFS can be used as a basis for continued research and improvement in access methods, cache management, and easy-to-use information retrieval techniques. Several possible commercial extensions to VOFS are also possible, in support of concepts such as use-based charging of software systems, mechanisms to enable populated object indices to be sold, and mechanisms in support of incrementally charging for data access.

References

[1] Agrawal, R., A. Arning. T. Bollinger, M. Mehta, J. Shafer, R. Srikant, “The Quest Data Mining System”, *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, August 1996.

[2] “Apple Announces Future Macintosh Operation System (OS) Strategy and Roadmap”, Apple Inc., January 1997, http://www.apple.ca/press/0197/NewOSStrategy.html

[3] The Baan Company, http://www.baan.com

[4] Bell, G. and J. N. Gray, “The Revolution Yet to Happen”, *Beyond Calculation: The Next Fifty Years of Computing*, P. J. Denning and R. M. Metcalfe Eds., New York: NY, Springer-Verlag, 1997.

[5] Bell Laboratories, *UNIX Programmer’s Manual Volume 1*, New York, NY: Holt, Rinehart and Winston, 1983.

[6] Berners-Lee, T., “Information Management: A Proposal”, 1989, http://www.w3.org/pub/WWW/History/1989/proposal.html

[7] Borenstein, N. and N. Freed, “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing Format of Internet Message Bodies”, Network Working Group RFC 1521, September 1993, .http://www.ioc.ee/home/tarvi/mime_pem/1521.txt

[8] Brown, K., K. Brown, and K. Brown, Mastering Lotus Notes, Almeda CA: Sybex, 1995.

[9] Brockschmidt, C., *Inside OLE 2nd Ed.*, Redmond, WA: Microsoft Press, 1995.

[10] Bush, V., “As We May Think”, *The Atlantic Monthly*, July 1945.

[11] Carey, M. J., D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vanderberg, “The EXODUS Extensible DBMS Project: An Overview”, *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier Eds., Los Altos, CA: Morgan-Kauffman, 1989.

[12] “Netscape Collabra: Enterprise-Ready Groupware Based On Open Standards”, http://www.netscape.com/comprod/products/communicator/datasheet.html#collabra

[13] Custer, H., *Inside Windows NT*, Redmond, WA: Microsoft Press, 1993.

[14] CyberAtlas: Your Guide to Cyberspace Research. http://www.cyberatlas.com

[15] Data Communications, “The 1996 Data Communications Market Forecast”, Data Communications on the Web, http://www.data.com/Roundups/1996_Market_Forecast.html

[16] Day, M., M. Koontz, and D. Marshall, *NetWare 4.0 NLM Programming*, San Jose, CA: Novell Press, 1993.

[17] Demers, Al, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, B. Welch, “The Bayou Architecture: Support for Data Sharing among Mobile Users”, http://www.parc.xerox.com/csl/projects/bayou/pubs/ba-mcw-94/www/MobileWorkshop_1.html

[18] Denning, P.J. and R. M. Metcalfe, *Beyond Calculation*, New York, NY: Springer-Verlag, 1997.

[19] Distributed Computing Environment (DCE), http://www.opengroup.org/tech/dce/

[20] “Build on a Solid Foundation”, Matrox Electronic Systems, http://www.matrox.com/videoweb/digisuit.htm

[21] Dun & Bradstreet, Inc., http://www.dnb.com

[22] Lotus Domino, http://domino.lotus.com/

[23] Garg, M. The DB2 Common Server eclipse.torolab.ibm.com net news server, private correspondence, March 24, 1997.

[24] Gray, J., and A. Reuter, *Transaction Processing: Concepts and Techniques*, San Mateo, CA: Morgan Kaufmann, 1993.

[25] Gray, J., “Thesis: Queues are Databases”, High Performance Transaction Processing (HPTS) Workshop Position Paper, 1995, http://www.research.microsoft.com/research/barc/gray/QueueIsDB.doc

[26] Gray, J., “Evolution of Data Management”, *IEEE Computer*, October 1996.

[27] Kistler, J., and M. Satyanarayanan, “Disconnected Operation in the Coda File System”, *ACM Transactions on Computer Systems*, February 1992.

[28] Hass, L. M., W. Chang, G. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita, “Startburst Mid-Flight: As the Dust Clears”, *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[29] Hoffman, D. L., W. D. Kalsbeek, and T.P. Novak, “Internet and Web Use in the U.S.”, *Communications of the ACM*, December 1996.

[30] Honeyman, P. and L. B. Huston, “Communications Consistency in Mobile File Systems”, October 1995, http://www.citi.umich.edu/mobile.html

[31] *IBM Application System/400 Technology*, IBM publication: SA21-9540, 1988.

[32] IBM DB2 Family, http://www.software.ibm.com/data/db2/

[33] IBM DB2 for VSE & VM, http://www.software.ibm.com/data/sql

[34] IBM Flowmark, http://www.software.ibm.com/ad/flowmark

[35] IBM MQSeries Product Family Home Page, http://www.hursley.ibm.com/mqseries

[36] "Illustra Product Description", http://www.informix.com/informix/corpinfo/zines/whitpprs/bloor/descr.htm

[37] Informix Software, http://www.informix.com

[38] *Information technology - Open systems interconnection - The directory*, ITU-T Recommendation X.500, ISO/IEC International Standard 9594, November 1993.

[39] J.D. Edwards & Company, http://www.jdedwards.com

[40] Leffler, S. J., M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Reading, MA: Addison-Wesley, 1989.

[41] "The Mach Project Home Page", http://www.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html

[42] Marshak, D. S., "Microsoft Exchange Leaps to 5.0", December 1996, http://www.microsoft.com/exchange/marshak.htm

[43] McConkey, S., M. Fridrich, G. Brignolo, "Scalable DCE on Tandem's Parallel Servers", Tandem Computers Inc., http://www.isoft.com/supplement/3_Tandem.html

[44] Mockapetris, P. "Domain Names - Implementation And Specification", RFC1035, ISI, November 1987, http://sunsite.auc.dk/RFC/rfc/rfc1035.html.

[45] *Microsoft ODBC 2.0 Programmer's Reference and SDK Guide*, Redmond, WA: Microsoft Press, 1994.

[46] *Microsoft SQL Server: Transact-SQL Reference*, Microsoft Corporation, 1995.

[47] "Microsoft "Falcon" Enters Beta Test", Microsoft Press Release, August 1996, http://www.microsoft.com/corpinfo/pres/1996/aug96/falconpr.htm

[48] "White Paper : Universal Data Access – OLE DB", Microsoft Corp, 1996. http://www.microsoft.com/oledb/prodinfo/wpapers/wpapers.htm

[49] "Microsoft Strategy for Reducing Cost of Owning PCs", 1996 http://www.microsoft.com.com/windows/zerowp.htm"

[50] "Microsoft Windows NT Directory Services: Building the Future with Next Generation Windows NT Directory Services", *MSDN Library*, November 1996.

[51] *Microsoft Windows NT Workstation Resource Kit*, Redmond, WA: Microsoft Press, 1996.

[52] "Microsoft Outlines the Active Platform: Comprehensive Foundation for the Next Generation Of Internet Software Development", Microsoft Press Release, October 28, 1996. http://www.microsoft.com/corpinfo/press/1996/oct96/actplatpr.htm

[53] Mohan, C., D. J. Haderle, B. G. Lindsey, H. Pirahesh, P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Transactions on Database Systems*, March 1992.

[54] "Internet Mail and News", Microsoft Technical White Paper, 1997. http://www.microsoft.com/ie/support/docs/tech30/imn.htm

[55] Netscape Communications Corporation, http://www.netscape.com

[56] Next Software, Inc., http://www.next.com

[57] Network Computer, Inc. http://www.nc.com/

[58] Oracle Corp., Applications, http://www.oracle.com/products/applications

[59] PeopleSoft, Inc, http://www.peoplesoft.com

[60] Peek, J., *MH & xmh E-mail for Users & Programmers*, Sebastopol, CA: O'Reilly & Associates, 1992.

[61] Rashid, R., D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones, "Mach: A System Software kernel" *Proceedings of the 34th Computer Society International Conference COMPCON 89*, February 1989 http://www.cs.cmu.edu/afs/cs/project/mach/public/www/doc/abstracts/syskernel.html

[62] SAP AG, http://www.sap.com

[63] Satyanarayanan, M., "Coda: A Highly Available File System for a Distributed Workstation Environment", *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, September 1989.

[64] Satyanarayanan, M., "Mobile Information Access", *IEEE Personal Communications*, February 1996.

[65] Satyanarayanan, M., "Fundamental Challenges in Mobile Computing", *Fifteenth ACM Symposium on Principles of Distributed Computing*, May 1996.

[66] Schaller, B., "The Origin, Nature, and Implications of "MOORE'S LAW"", http://www.research.microsoft.com/research/barc/gray/Moore_Law.html

[67] Stone, R. L., T. S. Nettleship, J. Curtiss, "VM/ESA CMS Shared File System", *IBM Systems Journal*, Vol. 30, No. 1, 1991.

[68] Stonebraker, M., and L. A. Rowe, "The Design of POSTGRES", *Proceedings of ACM SIGMOD Conference*, 1986.

[69] Stonkebraker, M., "The Case for Partial Indexes", *SIGMOD Record*, December 1989.

[70] Stross, R. E., *The Microsoft Way*, Reading, MA: Addison-Wesley, 1996.

[71] "TELE-TV", http://www.oracle.com/corporate/annual_report/html/custmed.html

[72] Transarc Corp, "Encina Product Overview", http://www.transarc.com/Public/ProdServ/Product/Encina/ENCoverview.html

[73] Transarc Corp, "Encina SFS Programming Guide", http://www.transarc.com/Public/Documentation/encina/windows/sfs_prog.html

[74] Vaskevitch, D., "Database in Crisis and Transition: A Technical Agenda for the Year 2001", *Proceedings of ACM SIGMOD Conference*, 1994.

[75] Vaskevitch, D., *Client/Server Strategies, 2nd Ed.*, Foster City, CA: IDG Books, 1995.

Appendix A. Server-Side Storage: Evolutionary Unification

Our basic thesis is that the requirements placed on server-side storage management systems are evolving towards a common set and, as a consequence, would best be implemented using a single back-end data store. We will first discuss the advantages of server-side storage unification[2], and then demonstrate the feasibility of using this approach to implement common server applications.

A.1 Advantages of Unification

One of the single most important problems facing server system owners today is the rapidly rising cost of systems administration. As software and hardware costs continue to decline and online data volumes increase, the percentage of the system total cost of ownership spent on administration is becoming an increasingly larger component of system cost [49]. Reducing the administration costs can best be addressed by making the system easier to install and administer and, more importantly, by reducing the number of subsystems in total. This would substantially reduce the administrative costs and, as a side effect, would provide uniformity in object naming, navigation, query, transaction management, security, administration, utilities, and resource quotas.

The clear advantages of a single unified storage manager are reduced administration costs and uniformity of operations and administration. A less-obvious advantage of storage unification is improved resolution of resource conflicts and sub-optimizations common on multi-server systems.

For example, consider a system with both a mail server and a relational database management system installed. Since most relational database management systems rely on a fixed allocation of memory resources, the administrator must choose some portion of the system's memory for the relational database, while saving the remainder for the mail server. However, any fixed memory resource allocation will necessarily be a sub-optimization in a load-varying environment. If too much memory is allocated to the database, then the mail system may be sufficiently memory-constrained to force the system into thrashing, causing everything on the system to run slowly. If too little memory is allocated to the relational database management system, it will run much more slowly. And, if the loads on the two servers change over time, the memory allocation won't be able to evolve with these changes, forcing more administrative work upon the system owners.

Clearly this problem is solvable by designing storage sub-systems to require no statically assigned resources. However, the problem is actually much more difficult than simply sharing resources. There must be mechanisms to share information between the resource managers about past and estimated future access patterns and the resultant estimated cache requirements. In most cases, both servers have a need for more memory (it's a rare subsystem that doesn't profit from more memory), so a determination must be made as to which server needs the memory most. Further complicating this resource allocation problem are quality of service commitments made by either resource manager. Two solutions to this problem seem possible: 1) design each resource manager of interest to support a common resource negotiation protocol or, 2) support all data-intensive server-side resource managers with a common storage manager.

[2] Note that by unification we are not referring to unification across different platforms, but across different subsystems within the same platform.

The first solution is weak for two reasons: 1) two full-function storage managers must be developed and maintained, and 2) a common resource allocation negotiation protocol must be devised for all shared resources, and this protocol itself cannot require significant overhead. And, assuming this were possible, it would also be necessary to develop support for the myriad of operational and administrative requirements mentioned above. However, if a single resource manager were used for both sub-systems, uniformity of operation is assured and the resource sharing problem is reduced to the classic "resource sharing within a single storage manager" problem. This is not an easy problem to solve, but it is one that has already been addressed modestly well by the storage managers of most commercial database management systems. It is clearly easier to solve this single resource-sharing problem than it is to solve three (within each resource manager and between the two resource managers).

A.2 Is Unification Possible?

Clearly, there are advantages to implementing a unified storage management solution. A reasonable approach to building the proposed integrated storage manager is to start with the storage manager of an existing relational database management system, since an SQL query processor is one of the significant storage manager clients that must be supported. And, it seems sensible to start with an existing component that already meets many of the requirements of the proposed integrated storage manager.

Client side applications are often written against a file system, and only a minority require the function offered by a commercial database management system. In the body of the paper, we have shown that the VOFS integrated storage system was capable of hosting these applications. Server side applications are typically considerably more demanding of the storage subsystems on which they depend, primarily because data scaling requirements are larger, the number of users are larger, and most objects must be shared between users. As a consequence, the storage requirements of many server-side applications can be satisfied by a commercial database management system, although a few might be better or more easily written against a file system API, and a very small minority truly do require a special purpose proprietary store. Of these three application classes, 1) commercial database, 2) file system, and 3) proprietary, the first two groups requirements could be satisfied by an integrated storage manager such as VOFS.

In what follows, we'll look at the storage requirement group that major server applications fall into and conclude that a storage manager such as VOFS would satisfy the bulk of the requirements.

1. *Mail Servers:*
   Data Communications reports that the electronic mail market is sizeable, estimated at $255 million in 1995 with a projected growth rate of 12% for 1996 [15]. The three leading mail systems currently in use are 1) Lotus Notes[8], 2) Microsoft Exchange [42], and 3) Netscape Mail [55]. Most mail systems depend upon a highly scalable, multi-user server where the bulk of the data is stored, with fairly low scale persistent caches at the client. In the case of Notes and Exchange, the server is based upon a proprietary data store that shares many features with commercial database engines. However, since neither support cluster configurations, large-scale installations must install many individual servers and either hand partition the mail databases or waste resources by replicating all mail to all servers. We've seen both solutions in use and neither is particularly easy to administer. Further, MIME[3] is

[3] MIME (Multipurpose Internet Mail Extensions): a mechanism supporting the attachment of non-text content such as sound, video, pictures, etc. in an email message 0.

nearly universally supported, enabling a huge increase in the average size of email messages. And, with the Internet connecting virtually all email users, the volume of email message is on the rise as well.

We predict that mail system servers will eventually depend upon commercial storage managers, partly due to the large development expense described above, but an even stronger motivator will be customer requirements. As mail increasingly becomes a "mission critical system" customers will not want to maintain multiple storage managers. Further, they will continue to bring robustness, availability, scaling, and management requirements to the mail server vendors at a pace that makes any other development approach look unaffordable. The net result is that email systems will either have to implement cluster parallelism and much of the infrastructure found in a modern commercial database management system, or depend upon an existing store.

2. *Queuing Servers*:
   Message queuing systems, often classified as Message-Oriented-Middleware, offer secure, reliable, and asynchronous message delivery. The Message-Oriented-Middleware market segment is estimated to be approximately $42 million in 1995 with an expected growth rate of 62% for 1996 [15]. No clear leader has yet emerged in this fairly new middleware domain. Two example systems are the IBM MQ Series [35], a strong contender for leadership, and Microsoft's Falcon [45], currently in beta test, which implements the MQ Series interface on Windows NT. Both MQ and Falcon depend upon special-purpose, proprietary data stores.

   Message queuing systems, like mail systems, end up requiring much of the infrastructure found in a modern relational storage manager. Jim Gray makes this argument convincingly in his paper *Thesis: Queues are Databases* [25]. Gray argues that current relational database management systems, with the addition of *read_past locks*, *read_through locks*, and *notification*, are sufficient to support queues and provide competitive performance.

   Customer requirements for message-oriented-middleware will evolve in much the same way as that of mail servers. As queues become more important to the operation of large businesses, the robustness, availability, scaling, and management requirements will make the proprietary data store approach inefficient. Ease of use and administration will likely be the prime differentiaters between queuing offerings.

3. *Vertical Application Servers:*
   The vertical application market includes human resources systems, finance systems, manufacturing resource planning, sales, and distribution applications. Some example solution providers in this category are SAP [62] , Oracle [58], Peoplesoft [59], Baan [1], Dun and Bradstreet [21], and J.D. Edwards [39]. Increasingly, companies are replacing internally-produced and maintained applications with those developed by such providers. Since these providers can leverage their investments over much more than a single customer, they can typically offer more function and better quality than internally-produced applications. All of these providers have already made the transition to depending upon commercial database management systems. In the case of Peoplesoft, both the data and the client programs are stored in the relational database.

   It is interesting to note that the main reason customers are replacing internally-developed applications with these commercial applications is that the commercial application provider can amortize research and development costs over a much large customer base. Purchasing such an application is therefore less expensive than building a customized one in-house. And, in turn, this is also one of the reasons why the application providers choose to depend upon commercial database management systems rather than constructing proprietary stores. Of this list of application providers, all but Oracle and Dun & Bradstreet support most leading database management systems. However, there is limited data definition language (DDL) similarity between these database products, and they have almost nothing in common when it comes to administration and performance monitoring. The application suppliers have done the right thing in depending upon relational database management systems for their persistence requirements, but functional, performance, and quality differences between these database products remains a significant expense for them.

4. *HTTP Servers:*
   Originally, all HTML files were stored directly in file systems. However, with more and more content being dynamically computed, it is becoming increasingly common for HTTP servers to dynamically construct pages loaded with live results queried from database management systems. Further, the complexity of managing web sites with page counts exceeding five to ten thousand is such that storing even static pages in a database management system is an effective way of dealing with the administrative complexity. Finally, many operating systems don't efficiently support thousands of concurrently open files, again making the database management system a reasonable choice.

   In addition to the web content storage evolution from file systems to relational database management systems, it is becoming increasingly common for resource managers to directly support HTTP as a native protocol, allowing direct query and retrieval of information stored by the resource manager. For example, the Lotus Domino [22] and the Informix Illustra [36] servers both support HTTP directly.

   As with the previous application domain, that of vertical applications, the evolution towards storing all HTTP server-accessed data in a single database management system is progressing.

5. *NNTP Servers:*
   Net news servers are read-mostly, file-based data stores that are moderate in their data moving requirements. Such servers have very similar storage requirements to HTTP servers in that they must deal with an extremely large number of typically quite small files. News servers work well in file systems from a performance perspective, but the large number of small files frequently causes problems just as it does with HTTP servers. For example, if a FAT file system is installed on a 2 GB partition (small by news standards), the cluster size will be 64K [51]. However, the average size of a news message is only 2.66K [23], wasting over 95.8% percent of the disk space. Better file systems, such as NTFS [51] or the UNIX Berkeley fast file system [40], avoid this problem by allowing tunable cluster sizes, but still don't fare that well with this application. The large number of concurrently open files can be a problem with both these systems. The other implication of attempting to store a large number of small files is that, depending on the file system design and configuration, it's possible to run out of file pointers (*inodes* on UNIX systems) before exhausting the available disk space.

   Higher scale, higher function data stores could avoid many of these problems and, more important, the savings in administration by only having one data store to backup, restore, monitor, etc. is significant.

6. *Directory Systems:*
   Some examples of directory services are Novel NDS [16], Windows NT Directory Services [50], DCE Directory Services [19], Domain Name Services [44], and X.500 [38]. Most directory systems are implemented as special-purpose distributed database management systems and are typically implemented upon proprietary data stores. Again, we see the same scaling and robustness requirements on these data stores as those found in commercial database management systems.

   There is no evidence that most directory services wouldn't be served as well, if not better, were they dependent upon a commercial database storage systems. In fact, to achieve better scaling, Tandem implemented their DCE CDS directory upon Non/Stop SQL, their relational database management system [43]. We know of one other company working on producing more scalable directory servers using a relational database infrastructure.

7. *Groupware, Workflow, and other Collaborative Frameworks*:
   Lotus Notes [8] is perhaps the best known example of this class of application. It is dependent on a proprietary data store, and the same storage manager is used on both the client and the server. Other applications are typically built upon mail systems such as Lotus Notes or Microsoft Exchange, or upon news servers such as Netscape's Collabra server [12]. IBM Flowmark [33] is an example of a workflow-specialized system that is built upon a commercial database management system.

   The server-side storage requirements for this class of applications are very similar to those of advanced mail systems. Consequently, the arguments that we made to host mail systems on commercial database management systems are equally valid here. The most important advantage is the ability to manage a single data store. The key advantages to using a single store in this application domain are 1) much reduced administration costs, and 2) uniform query across all objects in the store, and 3) development cost reductions. As an example of the opportunity for cost reductions, we note that cluster-based parallelism has already been added to many database management systems, but not yet to most mail servers.

8. *Video and other Continuous Media Servers:*
   Continuous media systems can be divided into two broad classes. The low scale class is made up of inexpensive systems that serve a small number of streams to a small number of users (for example, video editing systems typically process a handful of streams for a single user). Low-scale video editing and serving systems can be and are constructed upon modern file systems. For example, Digisuite from Matrox [20] is build directly upon NTFS.

   These low-scale personal video and editing systems can easily be served from within a commercial database management system, as most modern systems implement fairly exotic storage managers supporting extent-based allocations and striping of data across a large number of disks (supporting multiple parallel input streams). A 6 MB/sec MPEG2 video stream is only moving 21.6 GB per hour off of disk, which is well within reach of modern database systems as established by published data backup rates.

   High-end video servers, such as Oracle's TELE-TV [71], are capable of concurrently serving more than 100,000 users. These systems depend upon proprietary, isochronous data stores. High scale video servers remain beyond the capabilities of the integrated storage manager.

To summarize, mail, queuing, and NNTP servers require scalability, but are not typically using a commercial database now, or the concept of transactions. Vertical application servers, HTTP servers, directory systems, and collaborative frameworks are well along the path to using a commercial database. With video servers, it is difficult to justify using a unified storage manager at the high-end, but at the low-end it is feasible.

A.3 Resistance to Unification

Given the feasibility and the advantages of building application servers upon a single underlying data storage product, why are so many still built with special purpose data stores? We see several possibilities:

- *Prefer specialized design*
  One quick answer is that designers often prefer to implement their own solution rather than depend upon a previously-produced solution built by someone else. Although this clearly does happen occasionally, one would suppose Darwinian evolution would eventually win out and those designers that build upon bigger components will get more done and, therefore, be more successful.

- *Not ubiquitous*
  This is a valid concern. Designers are typically loath to depend upon a separately purchased product that customers must install. The obvious solution is to bundle a storage manager more capable than typical file systems. The IBM AS/400 [31] is the only system that we know of that has taken this approach.

  OS/400, the native operating system on the AS/400, includes an integrated storage manager as part of the base operating system. The file system depends upon this storage manager as does a separately purchasable SQL Query Processor, mail system, spreadsheet, and many others. AS/400 applications are all built upon this single storage manager. This strongly supports our claim that were a storage manager included in the base operating system, developers are much more likely to depend upon it than write their own. It is also worth noting that the AS/400 is well known for ease-of-use and low administrative costs. Since all applications running on an AS/400 share the same storage manager, they are able to achieve uniformity of naming, navigation, non-procedural object query, transaction management, security, administration, management, monitoring, quotas, and limits.

- *Inadequate performance*
  We can address the performance concern by drawing on the considerable research that has been done over the past 10 years in the area of extensible database management systems. At first glance it would appear that extensible database management systems and integrated storage management have little in common but, on closer inspection, they are attempting to solve many of the same problems. Our work on integrated storage attempts to show that most major server subsystems can be implemented within a single storage manager. Extensible database management systems research attempts to show that a single database system can support non-traditional data types and applications such as CAD/CAM, office systems, statistical databases, VLSI design, support for versioning, expert systems, and text applications. In effect, integrated storage and extensible database management systems are really just two different approaches with quite different motivations towards solving much the same problem: that of storing all server data in a single store.

  Three fairly well known extensible database research efforts are Postgres [67], Starburst [28], and Exodus [10]. All have published limited research results showing competitive performance. Both Postgres and Starburst have evolved into the commercial products

Informix [37] and DB2 [32] respectively, further strengthening the claim that they both work and have acceptable performance.

In conclusion on the performance issue, decisive proof of competitive performance can only be obtained through performance tests on real running systems. However, we have the commercial success of the AS/400 integrated storage manager as one point of supporting evidence and much of the work done on the performance of extensible database systems as another data point.

- *A heavyweight solution*
  This is the most commonly used reason for not implementing an application with storage requirements using an existing or unified storage manager. Typically, the designers agree that using an existing storage manager rather than writing a proprietary one is the right approach in general. For any given application, it's easy to arrive at the conclusion that much of the function offered by a general purpose commercial database management system isn't required and that this function may waste system resources. Oftentimes, the conclusion that additional function is not required is not borne out over time. For example, few would have anticipated the need for cluster parallelism by mail servers, but large enterprises mail installations are encountering scaling problems.

A.4 Conclusion

We have shown that a unified storage manager such as VOFS can satisfy the storage requirements of most server-side applications. The advantages of this approach on the server side go beyond administrative cost-reduction and unification of services, and also include the resolution of resource contention problems that arise when multiple storage managers are in use.

In general, it would appear that much of the server storage management integration that we are advocating has begun, and gradual progress for some application subsystems is being made. Storage managers are, under customer pressure, integrating with the underlying system security, administration, monitoring, and naming infrastructures. For those applications that can be written to an industry standard SQL interface, such as the vertical application area, external storage managers are the universal choice.

One impediment that must be overcome is that these data stores must either be ubiquitous, packaged in the base operating system or at least guaranteed to be installed on each server. Service such as HTTP, NNTP, and continuous media systems likely require lower level interface to the storage manager than provided by current relational database management system. Research in extensible database management systems establishes that this can be done but, until these interfaces are provided, these storage consumers will not be integrated.