The Design and Implementation of the Inversion File System

Michael A. Olson – University of California at Berkeley[1]

ABSTRACT

This paper describes the design, implementation, and performance of the Inversion file system. Inversion provides a rich set of services to file system users, and manages a large tertiary data store. Inversion is built on top of the POSTGRES database system, and takes advantage of low-level DBMS services to provide transaction protection, fine-grained time travel, and fast crash recovery for user files and file system metadata. Inversion gets between 30% and 80% of the throughput of ULTRIX NFS backed by a non-volatile RAM cache. In addition, Inversion allows users to provide code for execution directly in the file system manager, yielding performance as much as seven times better than that of ULTRIX NFS.

Introduction

Conventional file systems handle naming and layout of chunks of user data. Users may move around in the file system's namespace, and may typically examine a small set of attributes of any given chunk of data. Most file systems guarantee some degree of consistency of user data. These observations make it possible to categorize conventional file systems as rudimentary database systems.

Conventional database systems, on the other hand, allow users to define objects with new attributes, and to query these attributes easily. Consistency guarantees are typically much stronger than in file systems. Database systems frequently use an underlying file system to store user data. Virtually no commercially-available database system exports a file system interface.

This paper describes the design and implementation of a file system built on top of a database system. This file system, called ‘‘Inversion’’ because the conventional roles of the file system and database system are inverted, runs on top of POSTGRES [MOSH92] version 4.0.1. It supports file storage on any device managed by POSTGRES, and provides useful services not found in many conventional file systems.

The Inversion file system provides transactions and fine-grained time travel to users by taking advantage of the POSTGRES no-overwrite storage manager. File data are stored in the database, so that file updates are transaction-protected. In addition, the user may ask to see the state of the file system at any time in the past. All transactions that had committed as of that time will be visible, so the file system state will be exactly the same as it was at that moment. This is an important improvement on the coarse-grained time travel provided by other systems.

Another feature provided by Inversion is fast recovery. No file system consistency checker needs to run on the Inversion file system after a crash, since recovery is managed by the POSTGRES storage manager. File system recovery is essentially instantaneous. Any updates that were in progress at the time of the crash, but had not committed, will be rolled back. Any committed updates are guaranteed to be persistent across crashes.

In addition, files in the Inversion file system may be located on any device managed by POSTGRES. The Inversion namespace is uniform across devices. This means that the Inversion file system can span multiple devices (and device types) transparently. For example, the current system manages data stored on a 327Gbyte Sony optical disk WORM jukebox, and on magnetic disk. In the near future, a 9 TByte Metrum VHS-form factor tape jukebox will also be supported.

Finally, the fact that Inversion is built on top of POSTGRES makes it possible to issue *ad hoc* queries on the file system metadata, or even file data itself. Instead of mastering the use of many different programs, the user may examine the file system's structure and contents by formulating simple POSTQUEL queries. In addition, indices may be defined to make file system operations run faster, at the user's discretion.

The system described here currently supports a group of physical scientists researching global

[1] This research was sponsored by the University of California and Digital Equipment Corporation under Digital's flagship research project ‘‘Sequoia 2000: Large Capacity Object Servers to Support Global Change Research.’’ Other industrial and government partners include the California Department of Water Resources, United States Geological Survey, MCI, ESL, Hewlett Packard, RSI, SAIC, PictureTel, Metrum Information Storage, and Hughes Aircraft Corporation. Additional funding was provided by the Army Research Office under grant number DAAL03-91-0183.

climatic change as part of the Sequoia 2000 research project [STON91]. For this user community, transaction protection and fine-grained time travel are important services. The amount of storage managed requires novel fast recovery techniques like those provided by Inversion.

The rest of this paper is organized as follows. First, related work in file systems and in database management systems is presented. Next, the architecture of the POSTGRES database system is summarized, with attention to the features used by Inversion. Then the design and implementation of Inversion are described. A discussion of user-level access to Inversion files follows that. Next, Inversion's performance is measured on a benchmark based on the access patterns of its primary users. Finally, the conclusion summarizes the major points of the paper, and instructions on obtaining the code are given.

Related Work

File systems researchers have lately concentrated on providing new services to administrators and to users. Important areas of research include transaction protection, viewing past states of the file system (''time travel''), and attribute-based naming strategies.

QuickSilver [CABR88] is an early example of a file system that allows users to protect file changes with transactions.

The Wisconsin Storage System (WiSS) [CHOU85] was an early implementation of a storage manager supporting access to large data objects. WiSS decomposes large objects into pages, changes to which are protected by transaction boundaries. The WiSS client controls physical layout of object pages, making it easy to implement clustering strategies appropriate to particular large object applications. Indices on logical page locations make object traversal fast.

The EXODUS storage manager [CARE86] provides a set of low-level abstractions for managing large data objects. It supports efficient versioning of these objects. Users can extend the system to support new object types and operations on them.

Episode [CHUT92] embeds transaction protection in the file system directly for file system metadata changes. These transactions permit faster recovery after a crash than do graph-traversal programs like *fsck*(8). The file system manages a write-ahead log of directory updates, and can detect and remove transaction-inconsistent states quickly. However, Episode does not provide transaction protection to users, so user files may be left inconsistent by a system crash.

Log-structured file systems [ROSE91, SELT93] append file system changes to the end of a log on disk. A special ''cleaner'' process periodically reorganizes storage to recover space occupied by obsolete data. [SELT90] proposes extending such systems with support for transactions, and support for time travel could be added with appropriate changes to the cleaner process.

Finally, several libraries and toolkits have recently appeared that offer transactional and other services to users. 4.4BSD includes a database access method library, **db**(3), which provides keyed access to user data [SELT92]. This library includes support for transactions, allowing users to make consistent changes to files managed by the library. Kala [SIMM92] offers primitives allowing users to implement tailored transaction management and version control systems on a persistent programming language data store.

As very large storage devices, such as optical disk and tape jukeboxes, become widely available, many researchers are investigating ways of saving historical file system states automatically. Users can then travel in time over the file system, viewing old file contents at will.

The Plan 9 file system [QUIN91] periodically snapshots file system contents to an optical disk jukebox. Only changed files need to be copied; the system automatically reconstructs the complete state of the file system from the set of snapshots on the jukebox. The granularity with which snapshots are taken is configurable, but is currently once a day.

3DFS [ROOM92] uses a similar snapshot strategy to capture and recover file system state. This system extends the file system's namespace to include timestamps, making it possible to use programs like *ls*(1) and *cat*(1) to look at a directory's historical state, but complicating the user interface and breaking things like shell filename globbing.

Finally, there has been much activity lately in providing more robust query capabilities on file systems. The standard UNIX file system supports only rudimentary query tools, like *ls* and *find*.

[SECH91] describes a strategy for doing attribute-based lookups on files, where attributes are not limited to name, size, creation time, and so forth. [SECH91] makes the point that managing the namespace of attributes is nearly as difficult as managing the namespace of files.

The *Semantic File System* (SFS), described in [GIFF91], implements attribute-based naming by allowing users to express queries as operations in the file system namespace. ''Virtual'' directory names may be constructed to refer to all of those files whose attributes match values in the directory name. The query mechanism is somewhat restrictive – the current implementation supports only conjuncts, for example – but the authors plan to extend the syntax.

SFS allows users to install *transducers*, or procedures that compute attribute values for particular files. The results of these transducers can be indexed in Btrees for fast lookup later. An NFS daemon accepts requests from network clients and operates on SFS, providing these features to ordinary users without requiring them to add code to their systems.

The Inversion file system supports transactions for both user data and file system metadata. It permits finer-grained time travel than either Plan 9 or 3DFS. Like SFS, Inversion is extensible and supports indexing. It has a richer query language than SFS, but does not at present support access via NFS.

Overview Of The POSTGRES Database System

Inversion is able to provide so rich a set of services because it is built on top of a next-generation database system. This section gives an overview of the database system's architecture, with an emphasis on those feature used by Inversion.

The No-Overwrite Storage Manager

The POSTGRES database system [MOSH92] uses a novel no-overwrite technique for managing storage. This technique allows the user to see the entire history of the database and obviates the need for a conventional write-ahead log, speeding recovery [STON87]. When a record is updated or deleted, the original record is marked invalid, but remains in place. For updates, a new record containing the new values is added to the database. By using transaction start times and a special status file which indicates whether or not a transaction has committed, POSTGRES can present a transaction-consistent view of the database at any moment in history. This capability is referred to as *time travel*. Since only the start time and commit state of a transaction must be recorded in the status file, no special log processing is required at crash recovery time.

Periodically, obsolete records must be garbage-collected from the database, and either moved elsewhere or physically deleted. If the records are not saved elsewhere, some historical state of the database is lost. If time travel is desired, the records must be saved forever somewhere. This process is referred to as *record archiving*.

POSTGRES includes a special-purpose process, called the *vacuum cleaner*, that archives records. Obsolete records are physically removed from the table in which they originally appeared, and are moved to an archive.

Type and Function Extensibility

POSTGRES allows users to define new types for use in the database system. In addition, users may write functions in C or in POSTQUEL, the query language used by POSTGRES. These functions may be registered with the database system, and will be dynamically loaded by the data manager when they are invoked.

Inversion takes advantage of these two capabilities to provide strong typing on user files, and to support classification functions that describe files.

The Device Manager Switch

POSTGRES allows administrators to incorporate new storage devices by writing a small set of interface routines [STON93, OLSO92]. Based on the *bdevsw* switch in UNIX, the POSTGRES device manager switch registers the devices that are available to the database system. For each device, the required interface routines are listed. These routines are specific to the database system, and include, for example, code to create new tables and to commit transactions.

Version 4.0.1 of POSTGRES supports storage on non-volatile RAM, magnetic disk, and a 327GByte Sony optical disk WORM jukebox. The non-volatile RAM and Sony jukebox device managers operate on raw devices. In the current system, the magnetic disk device manager uses the underlying UNIX file system to store data, but a future release of POSTGRES will probably change this.

Accesses to data are location-transparent – the database manager finds the device storing the data and issues calls through the device manager switch to manipulate it. This allows users to store data on any device known to POSTGRES and manage it all identically. Logically, the database has no upper limit on its size.

The Design And Implementation Of Inversion

Inversion provides file system services to users by taking advantage of database services provided by POSTGRES. Strictly speaking, the Inversion file system is a small set of routines that are compiled into the POSTGRES data manager. Requests for file system data call these routines, which carry out the required database operations.

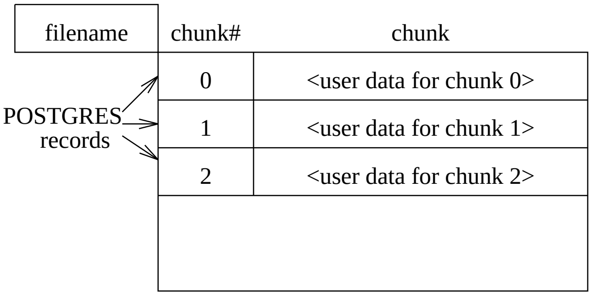
This section describes the Inversion support routines and how files are stored in the database system.

Decomposing Files into Tables

Files, generally viewed by users as byte streams, are stored in conventional file systems as a series of data blocks. The Inversion file system similarly ‘‘chunks’’ user data. Figure 1 shows the schema used to store file data in POSTGRES tables.

For every file, a uniquely-named table is created. File data are collected into chunks slightly smaller than 8kBytes. The size of the chunk is calculated so that a single record will fit exactly on a POSTGRES data manager page. This page size was chosen early in the design of POSTGRES, and was intended to make magnetic disk transfers fast. Although Inversion does not require magnetic disk in order to function, the inherited page size survives.

When a user writes a new data chunk to a file, a record is created consisting of the *chunk number*, or index of this chunk into the file, and the data chunk. This record is appended to the table storing the file. Multiple small sequential writes during a single transaction are coalesced to maximize the size of the chunk stored in each database record.



chunk#	chunk
0	<user data for chunk 0>
1	<user data for chunk 1>
2	<user data for chunk 2>

Figure 1: Decomposition of files into tables in Inversion

The Inversion file system provides a set of interface routines to create, open, close, read, write, and seek on files. Byte-oriented operations are turned into operations on chunks by calculating the chunk numbers of the affected chunks.

A file is located on particular device manager at creation. From that point on, accesses are device-transparent, both to the user and to the Inversion file system itself. The underlying device manager is called to instantiate blocks of the table storing the file.

When a file is modified, the records storing changed chunks are replaced in the normal way: the old record is marked as deleted by the current transaction, and the new record is marked as inserted by the current transaction. In order to speed up seeks on files, Inversion maintains a Btree index on the chunk number attribute.

Namespace and Metadata Management

Inversion stores the file system namespace in a table

```
naming(filename = char[],
     parentid = object_id,
     file = object_id)
```

where `filename` is the character string name of the file, `file` is the file's unique object identifier in the database (akin to an inode number in a conventional UNIX file system), and `parentid` is the object identifier of the directory containing the file.

A hierarchical namespace is imposed by having individual files point at their parent's `naming` entries. For example, the entries to construct the pathname ''/etc/passwd'' might be as shown in Table 1.

filename	*parentid*	*file*
/	0	810
etc	810	1076
passwd	1076	23114

Table 1: `naming` table entries for ''/etc/passwd''

The root directory, named ''/'', appears in every POSTGRES database as shipped from Berkeley. A single database corresponds to a mount point in conventional file system architectures; all of the files stored by Inversion in a single database are rooted at ''/'' in that database.

Inversion includes routines to parse pathnames in order to find desired files, and to construct pathnames for particular `file` identifiers. Various Btree indices on the `naming` table speed up these operations.

Besides the file system namespace, Inversion must manage additional metadata for every file. For example, the file's owner, type, size, and last access, modification, and creation times must be recorded. These attributes are stored in the table

```
fileatt(file = object_id,
        owner = owner_id,
        type = type_id,
        size = longlong,
        ctime = time,
        mtime = time,
        atime = time)
```

where the `file` entry matches the `file` entry in the `naming` table. A simple two-way table join of `naming` and `fileatt` can construct all the metadata for a given Inversion file.

The `naming` and `fileatt` tables manage only file system metadata. The actual file data are stored in separate tables. The name of the table storing data for a particular file is computed from the `file` identifier in the `naming` table. For example, the identifier of ''/etc/passwd'' in Table 1 was 23114. The name of the POSTGRES table storing data chunks for /etc/passwd would be `inv23114`.

Exploiting Type and Function Extensibility

Inversion supports typing of user files. A new file type is declared by issuing a `define type` command to the database system [MOSH92]. Once this command has been issued, files may be assigned the new type. POSTGRES will automatically enforce type checking when, for example, functions are called that operate on the file.

Functions that operate on a particular type may also be registered with the database system; this applies to file types as well as to smaller database types like money and time. These functions may be invoked from the query language, and their results examined.

file type	*defined functions*
ASCII document	keywords, wordcount, linecount
troff document	keywords, wordcount, linecount, fonts, sizes
Coastal Zone Color Scanner satellite image	pixelavg, pixelcount, getpixel
Advanced Very High Resolution Radiometer satellite image	snow, pixelcount, pixelavg, getpixel, getband

Table 2: Example file types and functions

Several databases storing Inversion files exist at Berkeley storing many different types of files. For example, documentation, Hierarchical Data Format files, and images from different kinds of satellites are all stored as different file types. Table 2 lists some of the existing types and functions that operate on them. Invoking a function from the query language is easy[2]:

```
retrieve (filename)
      where "RISC" in keywords(file)
```

would find all the files stored by Inversion for which the `keywords` function was defined, and whose keywords included ‘‘RISC’’.

Adding new functions to POSTGRES is straightforward. Functions may be written in C or in POSTQUEL. In release 4.0.1 of the database system, these functions are dynamically loaded into the data manager process and executed with its permissions. A future version of POSTGRES will support an RPC interface to address the obvious security problems raised by this approach.

Caching and Layout Policy

Inversion does not implement cache management or layout policies independent of those used by the POSTGRES database system for regular user tables. There are two reasons for this. First, POSTGRES already implements reasonably good policies for relational data. Second, we have chosen to concentrate on providing new capabilities to clients of the file system, rather than on extensive low-level performance tuning.

This section describes cache management and layout policies implemented by POSTGRES. Inversion inherits these policies unchanged from the data manager.

Cache Management

POSTGRES maintains an in-memory shared cache of recently used 8KByte data pages. The size of this cache is tunable when the file system is installed; as shipped, the system uses 64 buffers, but the version in use locally uses 300.

Data pages are kicked out of this cache in LRU order, regardless of the device from which they came. Dirty pages are written to backing store before being deleted from the cache. How they are written depends on the device backing them; for magnetic disk pages in POSTGRES 4.0.1, pages are written to the file system buffer cache, but are not necessarily forced to disk.

Individual devices are managed via the device manager switch table. Every device manager may use a private cache for its data. For example, the file system buffer cache is a secondary buffer cache for magnetic disk pages in POSTGRES.

A more interesting example of device manager caching is the Sony optical disk device manager. Due to extremely high setup costs (many seconds to load an optical platter) and relatively low transfer rates, using the jukebox directly for every transfer would be very slow. Instead, the Sony jukebox device manager caches recently-used blocks on magnetic disk. The size of this cache is tunable, and defaults to 10MBytes.

Data Layout

POSTGRES uses several strategies to improve performance by exploiting locality of reference. First, the selection of a relatively large page size (8192 bytes) means that a single data page in memory contains a good deal of user data. Second, individual device managers are free to implement sensible layout policies of their own on backing store.

Since the magnetic disk device manager uses the native UNIX file system, it inherits the layout policy used by that code. For systems that use cylinder group strategies like that described in [MCKU84], data for a single file are kept close together. As smart SCSI disks proliferate, this strategy becomes less effective, since many SCSI controllers silently remap blocks to physically distant locations on the storage medium.

The Sony jukebox device manager allocates tables in units of *extents*, where an extent is a collection of physically contiguous 8KByte data pages. The extent size is tunable when POSTGRES is installed, but defaults to 16 pages. The choice of extent size involves a tradeoff; for small tables, much of the extent will go unused, while large tables would benefit from the overhead reductions in transferring very large extents.

[2] The example queries in this paper have been simplified somewhat for presentation. In particular, the range variables and long names used by POSTQUEL have been removed. The intent is to show how queries are expressed, not to introduce the reader to the intricacies of POSTQUEL syntax.

Services Provided by Inversion

Because it is built on top of an extensible database management system, Inversion is able to provide the following services:

- Transaction protection for changes to file data and file system metadata.
- Fast recovery after a system crash.
- Time travel on any data managed by Inversion, including metadata, data, and functions defined by users that operate on files.
- Typed files, with user-defined functions that can operate on them.
- Management of very large files.
- Strong consistency guarantees.
- Powerful query support on the file system's contents and metadata.

Transaction Protection

Transaction protection allows users to make multiple interdependent changes to a set of files atomically. For example, programmers working on a large software project may need to be able to check in several fixed source code files at the same time. If the system crashes when some, but not all, of the files have been checked in, then the software project's master directory will be in an inconsistent state.

Similarly, file system metadata changes must be made atomically. For example, when a new file is created in a directory, the directory (or file system namespace) must be updated, and the new file must be created. If only one of these operations takes place, then the file system's structure is corrupt.

Inversion supports transactions encompassing changes to arbitrary numbers of files, and commits or aborts all changes atomically. The transaction mechanism is provided by POSTGRES. No special code was required for Inversion.

In addition, a standard database two-phase locking protocol [GRAY76] allows concurrent access to files while preventing simultaneous changes from interfering with one another.

Fast Recovery

The POSTGRES transaction mechanism was designed to permit fast recovery of the database system after a crash. Data stored by Inversion is recoverable in the same way as ordinary relational data. No special boot-time file system check program needs to be run. By examining the commit state of records encountered in the database, any changes that were not committed before a system crash are automatically detected and ignored.

The only difficulties arise when the physical storage medium is damaged, or when garbage has been written to the medium by hardware or software failures. Inversion could detect these cases by making all blocks *self-identifying*; every block could be tagged with its `file` identifier and block number. Although the current version of the system does not do this, space has been reserved in the tables storing file data for this purpose.

Time Travel

POSTGRES allows users to examine any transaction-consistent historical state of the database. Since transactions are committed atomically, users can ‘‘change time’’ to any instant in history, and see the database exactly as they would have seen it then.

Inversion inherits this fine-grained time travel from the data manager. All old versions of files are visible. Since user-defined functions are stored in the database in the same way that ordinary files are, users can even run old versions of these functions.

Inversion does not create copies of entire files every time a change is made. Instead, only the changed blocks are saved. The appropriate historical version of a file is constructed using an index on all of the file's available data, including both old and current blocks. For files in which the user has no interest in maintaining history, POSTGRES can be instructed not to save old versions.

As was mentioned in the section on related work, both Plan 9 [QUIN91] and 3DFS [ROOM92] support time travel, but only at relatively coarse granularity. These systems snapshot file system state once a day or so. Intervening states of the file system are not visible.

The ability to see all of history can be important; for example, it allows users to undelete files removed accidentally, or to recover a working version of a program which they have changed. Inversion provides no direct support for annotating versions of files (though this capability would be easy to add as a user-defined function), but if it did, it would provide a superset of the services offered by revision control programs like *rcs*(1).

Typed Files

Conventional file systems offer little or no support for typing of files; conventions have evolved among users to deal with this. For example, C programs by convention have suffixes of ‘‘.c’’ on most UNIX systems.

Some systems, such as Locus [WALK83] and CODA [KIST90], support typing, but provide only a small number of file types, and do not permit users to add functions managing these types easily. [GIFF91] makes it easy to add new functions to the storage system. Inversion supports strong typing and allows users to add new functions easily. Functions operating on Inversion files may be written in C or in POSTQUEL, and will be dynamically loaded and executed on demand by the database system.

Large Files

The practical upper limit on file sizes in the current UNIX Fast File System is 4GBytes. Inversion files can be 17.6TBytes in length. Support for very large files is important in managing scientific datasets. Researchers are currently forced to break large files up into pieces and to reassemble them inside applications; this is not necessary under Inversion.

Consistency Guarantees

Since many files have complicated structure and are semantically rich, it is important to guarantee that they remain structurally consistent. The symbol table and text space of a program, for example, contain mutually dependent entries, and neither should be changed without corresponding changes to the other. Use of transaction processing and the POSTGRES rules system [MOSH92] can guarantee this consistency.

More fundamentally, scientific users have historically managed some of their data in databases, and some in file systems. Typically, the reason for this has been that the database did not provide good support for management of large data files. This meant that records in the database could refer to files entirely outside the control of the database management code. Dividing responsibility in this way made it impossible to guarantee that references were maintained correctly. Inversion alleviates this problem by allowing users to store both tabular and file data in the same storage management system.

Query Processing

Since all Inversion data are managed by POSTGRES in tables, the user may run arbitrarily complex queries over the file system's namespace, metadata, file contents, and user-defined functions in order to find files of interest. A full-function query language makes it possible to do very sophisticated searches of the file system easily.

Services Under Investigation

In addition to the services listed above, we are exploring several other novel features in Inversion.

Users are often interested in saving only compressed versions of their files. Random access to compressed files is typically impractical; files compressed sequentially must be entirely uncompressed before random access on them is efficient.

Inversion supports compression and uncompression of ‘‘chunks’’ of user files. Special indices are maintained indicating the sizes of the uncompressed and compressed chunks. Random access on the uncompressed version is straightforward. Inversion determines which compressed chunk contains the bytes of interest, uncompresses it, and returns the user only the desired data. This approach provides good storage utilization and maintains reasonable random access times for files. We are investigating suitable compression strategies for the scientific data files stored in Inversion at Berkeley.

File migration is also of interest to users of very large file systems [MILL93]. Files that meet some selection criteria should be moved from fast, expensive storage like magnetic disk to slower, cheaper storage, such as magnetic tape. We are exploring strategies for using the POSTGRES predicate rules system to allow users and administrators to define migration policies. Arbitrarily complex rules controlling the locations of files or groups of files would be declared to the database manager. When a file met the announced conditions, it would be moved from one location in the storage hierarchy to another. The primary advantage of this strategy over more conventional ones is its flexibility; the rules system allows detailed migration conditions to be set up for as many different kinds of files as necessary.

Finally, distributed file systems are a subject of strong interest among both computer scientists and physical scientists. Users would like to have their files located nearby, but to have access to files stored at remote sites. For Inversion, this has implications in database cache management, migration, transaction control, and locking. Several researchers at Berkeley are exploring these issues.

Comparison to Other Database Systems

Many relational databases support *binary large objects*, or BLOBs. Typically, BLOB values can be stored in the database or fetched from it, but not manipulated from the query language in any useful way.

POSTGRES supports large object storage by creating Inversion files to store object data. All of the services available to Inversion users are also available to users of BLOBs. This includes strong typing, the ability to manipulate BLOBs from the query language, and a file-oriented interface to the data they contain. Commercial vendors of relational databases do not offer these services. Some research systems, such as Starburst [HAAS90], do offer typed large objects. Starburst provides access to large objects via an extension to the SQL cursor mechanism [LEHM89].

The integration of large database objects with Inversion means that two different clients can share data that they use in different ways. The same Inversion file can be used by a database application and by a file system client simultaneously. This means that existing programs, which store their data in a file system, continue to work. New applications can be developed that use the database directly, and can operate on the same data as the older code.

Access To Inversion Files

User files stored in Inversion may be opened, read, and written using calls modeled on those supported for ordinary UNIX files. The current implementation requires programmers to link a special library in order to access Inversion file data.

```
int
p_creat(char *path, int mode)

int
p_open(char *fname, int mode,
        int timestamp)

int
p_close(int fd)

int
p_read(int fd, char *buf,
        int len)

int
p_write(int fd, char *buf,
        int len)

int
p_lseek(int fd,
        long offset_high,
        long offset_low,
        int whence)
```

Figure 2: Interface routines for Inversion clients

The routines that manipulate Inversion files are shown in Figure 2. The important differences are in the routines p_open and p_lseek. Since the user may ask to see any historical state of the file system, the p_open call includes a parameter to specify the time for which the file should be viewed. Historical files may not be opened for writing. POSTGRES supports storage of objects up to 17.6TBytes in size, which means that an Inversion file may be that big. The extra parameter to p_lseek allows the user to specify a wider range of byte positions. Finally, the mode flag to p_open and p_creat encodes the device on which the file should reside at creation time.

Inversion also supports three interface new routines, p_begin(), p_commit(), and p_abort(). These routines begin, commit, and abort a transaction, respectively. Neither POSTGRES nor Inversion supports nested transactions, so a single application program may only have one transaction active at any time.

In the near term, we plan to provide NFS access to Inversion. In order to do so, we will be forced to support the standard interfaces for creating, opening, and seeking on files. We plan to do so, but to provide new fnctl() support to provide access to time travel and very large files.

However, we are unsure how to support transactions via NFS. The NFS protocol makes every operation an atomic transaction, which severely limits the utility of transactions in Inversion. We are most likely to follow the protocol specification, and to provide no multi-operation transaction protection for Inversion files accessed via NFS. Users who want the richer services may still link with the special library, and users who simply want to list directory or file contents will not need to concern themselves with transaction management.

Finally, Inversion supports ad-hoc queries on file system metadata by using the POSTQUEL query language processor. Users may run the query language monitor program to execute arbitrarily complex queries. For example, the query

```
retrieve (filename)
  where owner(file) = "mao"
     and (filetype(file) = "movie"
     or filetype(file) = "sound")
     and dir(file) = "/users/mao"
```

would return the names of all movie or sound files owned by user ‘‘mao’’ and found in the directory /users/mao.

Inversion currently stores several hundred satellite images from by the Thematic Mapper satellite a device which records five spectral bands for each image. A function has been written to find snow in these images. POSTGRES permits the query

```
retrieve (snow(file), filename)
  where filetype(file) = "tm"
    and snow(file)/size(file) > 0.5
    and month_of(file) = "April"
```

which will find all TM images stored anywhere in the file system which are from the month of April and which contain more than 50% snow. The snow function returns a count of the number of pixels that contain snow in the image. The query returns the actual number of pixels covered by snow and the name of the file storing the image.

The expressive power of a full-fledged query language is clear. However, the language can also be cumbersome and difficult for database novices to master. Although we have no plans to do so, a simpler query interface like that used by the Semantic File System [GIFF91] could be constructed. Similarly, an NFS server could manage time travel by extending the file system namespace and passing dates along to the database system for processing. This approach has been explored by [ROOM92].

Performance Of The Inversion File System

This section presents measurements of the performance of the Inversion file system. Inversion is intended to support physical scientists working on the Sequoia 2000 project [STON91], [KATZ91]. In general, these scientists use Inversion as a network file server. The system configuration evaluated here is that used by Sequoia researchers. Inversion is compared to NFS [SAND85] running on identical hardware.

System Configuration

Inversion was installed on a DECsystem 5900 with 128MBytes of main memory. The operating system running on the machine was ULTRIX 4.2.

Files were located on a 1.3GByte DEC RZ58 disk drive attached to the DECsystem 5900.

Files were opened, read, and written from a remote client running on a DECstation 3100 under ULTRIX 4.2. Client/server communication was via TCP/IP over a 10Mbit/sec Ethernet.

Inversion was compared to the ULTRIX 4.2 implementation of NFS. The NFS server was run on the same DECsystem 5900, using the same disk, as Inversion. The NFS client was the same DECstation 3100.

The NFS implementation on the DECsystem 5900 used a service called PRESTOserve to speed up writes. To guarantee that NFS servers remain stateless, NFS must force every write to stable storage synchronously [SAND85]. PRESTOserve consists of a board containing 1MByte of battery-backed RAM and driver software to cache NFS writes in non-volatile memory. As will be seen below, this substantially improved the write throughput of NFS under ULTRIX. This non-volatile memory was not used by Inversion.

The Benchmark

The benchmark consisted of the following operations:

- Create a 25MByte file.
- Measure the latency to read or write a single byte at a random location in the file.
- Read 1MByte in a single large transfer.
- Read 1MByte sequentially in page-sized units. The page size was chosen to be efficient for the file system under test.
- Read 1MByte in page-sized units distributed at random throughout the file.
- Repeat the 1MByte transfer tests, writing instead of reading.

All caches were flushed before each test. These tests measure worst-case throughput for operations that Sequoia researchers are likely to carry out.

Benchmark Results

Figure 3 shows the elapsed time to create a 25MByte file under Inversion and under ULTRIX NFS. As shown, Inversion gets about 36% of the throughput of NFS for file creation. This difference is due primarily to the extra overhead in maintaining indices in Inversion. For every page written to the file, Inversion must create a Btree index entry so that the page can be located quickly later. Btree writes are interleaved with data file writes, penalizing Inversion by forcing the disk head to move frequently. The NFS implementation does not maintain as much indexing information on the data file, and so can postpone writing its index until all data blocks have been written. This means that NFS writes the data file sequentially, improving throughput.

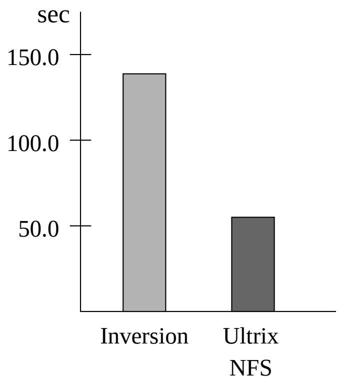


Figure 3: 25MByte file creation times

Figure 4 shows the overhead for reading or writing a single byte at a random location in the 25MByte file just created.

Since all caches were flushed prior to running the test, a disk access is required. For single-byte reads, Inversion gets 70 percent of the throughput of NFS. Single-byte writes are slightly worse; Inversion is 61 percent of NFS. Since Inversion never overwrites data in place, a new entry must be written to the Btree block index, accounting for the difference.

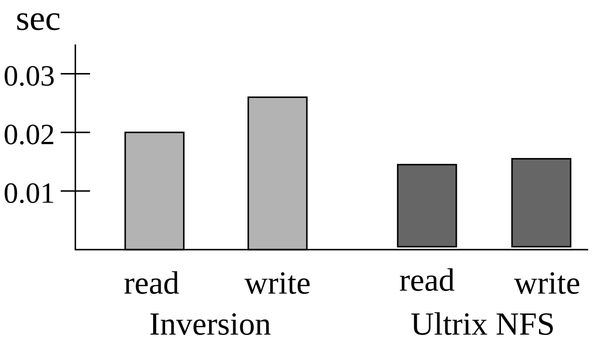


Figure 4: Random byte access

Figure 5 compares Inversion to ULTRIX NFS on large and small reads. The first pair of bars compares throughput using a single large transfer to move data from the server to the client. In this case, Inversion gets eighty percent of the throughput of NFS. When smaller transfers are used, Inversion drops to 47 percent of NFS. Profiling reveals that extra work is done in allocating and copying buffers in Inversion. If some of this overhead were eliminated, Inversion's performance could be brought closer to that of NFS. Since single-byte transfer times are much closer under the two file systems, there is reason to believe that tuning will improve Inversion.

The final pair of bars in Figure 6 compares the transfer rates of Inversion and ULTRIX NFS when the pages read are distributed at random throughout the 25MByte file. In this case, Inversion gets 43 percent the throughput of NFS. The additional overhead incurred by traversing the Btree page index in Inversion accounts for much of the slowdown.

Figure 6 presents the write performance of Inversion and ULTRIX NFS. The tests run were identical to those performed for Figure 5, except that reads became writes. In these tests, the effect of the PRESTOserve board used by NFS is dramatic.

Since NFS must flush every write to stable storage, Inversion should have much better performance than NFS without non-volatile RAM. The reason for this is that NFS is forced to treat every write as a single transaction, and commit it to disk immediately. Inversion, however, can obey the transaction constraints imposed by the client program, and commit a large number of writes simultaneously.

Figure 6 shows that Inversion is slower than ULTRIX NFS backed by PRESTOserve. For a single large write request, Inversion gets 43 percent the throughput of NFS. For page-sized sequential writes, Inversion does worse, getting only 31 percent of NFS' throughput. For random accesses, Inversion has only 28 percent the performance of NFS. In fact, the NFS measurements show no degradation due to random accesses, since the whole 1MByte write fits in the PRESTOserve cache, and is not flushed to disk.

Evaluation of Benchmark Results

The benchmark results indicate that Inversion is penalized for not using a non-volatile RAM buffer such as PRESTOserve, and by its relatively heavyweight network communication protocol, which is based on TCP/IP.

An obvious strategy would be to disable PRESTOserve and rerun the benchmark. We used production file systems to collect the measurements shown here. Both the Ultrix and Inversion file systems are served by the same DECsystem 5900, and political considerations made it impossible to reconfigure the Ultrix NFS server for this test.

Another sensible strategy would be to run the benchmark on local file systems, so that network communication costs and the benefit of PRESTOserve were eliminated. [STON93] presents the results of such a benchmark on a 12-processor Sequent Symmetry machine running the Dynix operating system. Those results show that Inversion

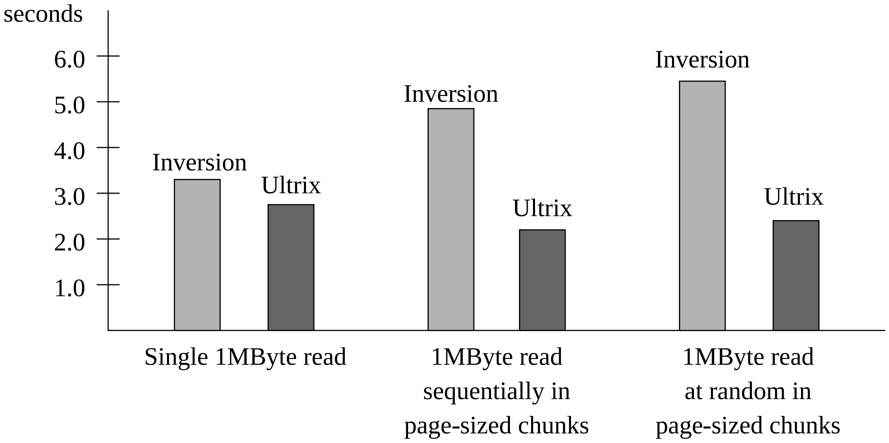


Figure 5: Read throughput

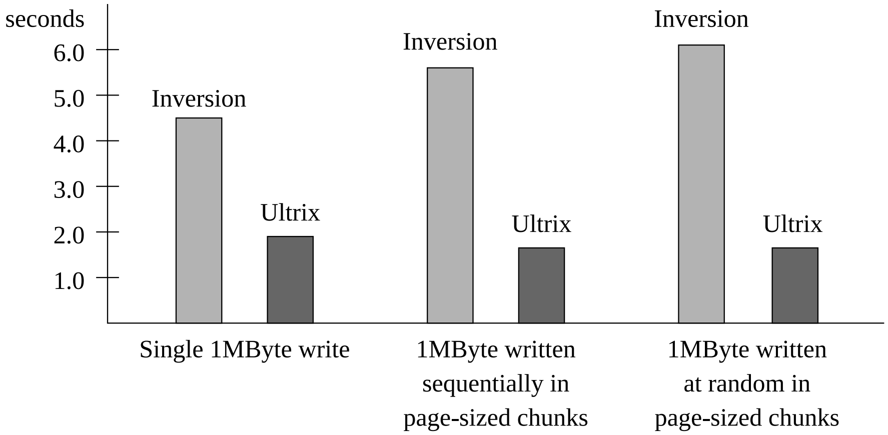


Figure 6: Write throughput

gets better than 90% of the throughput of the native file system on large sequential transfers, and roughly 70% of the throughput on small, uniformly random transfers.

A final strategy is to exploit the extensibility of Inversion to run the benchmark directly in the file system, rather than using a separate application program. The results of such an implementation are presented below.

In this case, the routines for the benchmark were declared to POSTGRES as user-defined functions, and were dynamically loaded into the POSTGRES data manager on invocation. This represents the best performance available to users under Inversion, since the benchmark and the file system are running in the same address space, and no data must be copied between them. Note that the same files can be used simultaneously by dynamically-loaded code and by the more conventional client/server architecture.

Table 3 shows the performance of the single-process benchmark. Comparable performance numbers for ULTRIX are not included, since the native ULTRIX file system does not support this approach. For convenience, the performance of client/server Inversion and ULTRIX NFS, presented in the previous section, are included.
The elapsed time for each test is reported in seconds. The measurements shown are the means of ten runs. In all cases, the standard deviation was negligible.

As Table 3 shows, the single-process Inversion benchmark is faster than either of the network benchmarks in virtually all categories. The important exception is in random write time, for which ULTRIX NFS using PRESTOserve is fastest, since no disk seeks are required. Note, however, that the single-process implementation of Inversion is faster than ULTRIX with PRESTOserve for sequential transfers. File creation is slower in both Inversion benchmarks, due to the overhead of creating the Btree index of blocks.

The important comparison is between Inversion running on two machines and Inversion running in a single process. For 1MByte operations, remote access adds between three and five seconds to the elapsed time of each test. It is clear that the client/server communication protocol used by the file system is much too heavy-weight, and should be optimized. The current implementation uses TCP/IP for communication. Given optimization of the protocol, it is reasonable to expect performance within fifty percent of ULTRIX NFS and PRESTOserve from Inversion.

Conclusions

Inversion provides significant new services to file system users by adding a small amount of code to the POSTGRES extensible database system. These services include transaction protection for file updates, fast recovery in the face of system crashes, time travel on file data and metadata, strong typing, the ability to add new functions that operate on file types to the storage system, and support for complex queries on the file system name and and attribute spaces.

The current implementation of the system requires clients to link a special library in order to use Inversion files. In the near future, we plan to extend the system with support for NFS, although the NFS interface will probably not support transactions.

Performance of the system as a network file server is reasonable, although continued tuning is certainly necessary. Depending on the access pattern, Inversion is between 30 and 80 percent as fast as the native ULTRIX file system over NFS carrying out the same operations. ULTRIX is able to exploit a large non-volatile RAM cache that is not used by Inversion, which skews performance in favor of ULTRIX. For applications in which performance is critical, users can arrange for their code to be run by the Inversion file system directly, by creating user-

Operation	*Inversion client/server*	*ULTRIX NFS*	*Inversion single process*
Create 25MByte file	141.5	50.6	111.6
Single 1MByte read	3.4	2.8	0.4
Page-sized sequential 1MByte read	4.8	2.2	0.4
Page-sized random 1MByte read	5.5	2.4	0.8
Single 1MByte write	4.6	2.0	1.4
Page-sized sequential 1MByte write	5.6	1.7	1.4
Page-sized random 1MByte write	6.0	1.7	2.9
Read single byte	0.02	0.01	0.01
Write single byte	0.03	0.02	0.02

Table 3: Elapsed time in seconds for benchmark tests in three configurations

defined functions and registering them with the database system. In this case, performance is nearly as good as the native ULTRIX file system used locally.

The Inversion installation at Berkeley currently manages approximately seven hundred megabytes of user file data, spread across magnetic, magneto-optical, and write-once optical disks. A number of special-purpose functions that operate on satellite image files have been written and are in regular use. More are under development by Sequoia 2000 researchers.

Availability

Inversion is supported in release 4.0.1 of the POSTGRES database system. POSTGRES 4.0.1 is available for anonymous ftp from postgres.CS.Berkeley.EDU (128.32.149.1) in directory pub/, as file postgres-v4r0r1.tar.Z. If you prefer to be mailed a tape, you may send a check for US $150.00 to

POSTGRES Project
557 Evans Hall
University of California at Berkeley
Berkeley, CA 94720

Attn: Claire Mosher

Be sure to specify the kind of tape you want. We are able to write 9track tapes at 1600bpi and 6250bpi, Exabyte cartridges, TK50s, and QIC tapes.

Acknowledgments

Wei Hong, Randy Katz, Ray Larson, Margo Seltzer, Mike Stonebraker, and Mark Sullivan offered guidance during early phases of the design of Inversion. John Kohl reviewed an early draft of this paper, and made useful suggestions for its improvement. Jim Frew and the entire Sequoia 2000 community have been gracious test subjects, exercising the file system and helping to identify its problems.

References

[CABR88] Cabrera, L., and Wyllie, J., ‘‘QuickSilver Distributed File Services: An Architecture for Horizontal Growth’’, *Proc. 2nd IEEE Conference on Computer Workstations*, March 1988.

[CARE86] Carey, M. *et al.*, ‘‘Object and File Management in the Exodus Extensible Database System,’’ *Proc. 1986 VLDB Conference*, Kyoto, Japan, August 1986.

[CHOU85] Chou, H., DeWitt, D., Katz, R., and Klug, A., ‘‘Design and Implementation of the Wisconsin Storage System’’, *Software Practice and Experience*, **15**(10), October 1985.

[CHUT92] Chutani, S., *et al.*, ‘‘The Episode File System’’, *Proc. USENIX Winter 1992 Technical Conference*, San Francisco, CA, January 1992.

[GIFF91] Gifford, D., Jouvelot, P., Sheldon, M., and O’Toole, J., ‘‘Semantic File Systems’’, *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991.

[GRAY76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., ‘‘Granularity of locks and degrees of consistency in a large shared data base’’, *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, pp. 365-394.

[HAAS90] Haas, L. *et al.*, ‘‘Starburst Midflight: As the Dust Clears,’’ *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[KATZ91] Katz, R., *et al.*, ‘‘Robo-line Storage: Low Latency, High Capacity Storage Systems Over Geographically Distributed Networks,’’ Sequoia 2000 Technical Report 91/3, UC Berkeley, October 1991.

[KIST91] Kistler, J. J. and Satyanarayanan, M., ‘‘Disconnected Operation in the CODA File System’’, *Proc. Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991.

[LEHM89] Lehman, T., ‘‘Long Field Support in Starburst,’’ *Proc. 1989 VLDB Conference*, Amsterdam, Netherlands, September 1989.

[MCKU84] McKusick, M., Joy, W., Leffler, S., and Fabry, R., ‘‘A Fast File System for UNIX’’, *ACM Transactions on Computer Systems* **2**(3), August 1984.

[MILL93] Miller, E., Katz, R., and Strange, S., ‘‘An Analysis of File Migration in a Unix Supercomputing Environment’’, *Proc. Winter 1993 USENIX*, San Diego, CA, January 1993.

[MOSH92] Mosher, C., *ed.*, ‘‘The POSTGRES Reference Manual, Version 4’’, UCB Technical Report M92/14, Electronics Research Laboratory, University of California at Berkeley, Berkeley, CA, March 1992.

[OLSO92] Olson, M., ‘‘Extending the POSTGRES Database System to Manage Tertiary Storage’’, Master’s thesis, University of California at Berkeley, May 1992.

[QUIN91] Quinlan, S., ‘‘A Cached WORM File System’’, *Software – Practice and Experience*, **21**(12), December 1991.

[ROOM92] Roome, W.D., ‘‘3DFS: A Time-Oriented File Server’’, *Proc. USENIX Winter 1992 Technical Conference*, San Francisco, CA, January 1992.

[ROSE91] Rosenblum, M. and Ousterhout, J., ‘‘The Design and Implementation of a Log-Structured File System’’, *Proc. 13th Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991.

[SECH91] Sechrest, S., ‘‘Attribute-Based Naming of Files’’, University of Michigan Technical Report CSE-TR-78-91, January 1991.

[SELT90] Seltzer, M., and Stonebraker, M., ‘‘Transaction Support in Read Optimized and Write

Optimized File Systems,'' *Proc. 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[SELT92] Seltzer, M., and Olson, M., ''LIBTP: Portable, Modular Transactions for UNIX'', *Proc. USENIX Winter 1992 Technical Conference*, San Francisco, January 1992.

[SELT93] Seltzer, M., Bostic, K., McKusick, M, and Staelin, C., ''An Implementation of a Log-Structured File System for UNIX'', *Proc. Winter 1993 Usenix*, San Diego, CA, January 1993.

[SIMM91] Simmel, S., and Godard, I., ''The Kala Basket – A Semantic Primitive Unifying Object Transactions, Access Control, Versions, and Configurations'', *Proc. 1991 Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1991.

[STON87] Stonebraker, M., ''The POSTGRES Storage System'', *Proc. 1987 VLDB Conference*, Brighton, England, Sept. 1987.

[STON91] Stonebraker, M., and Dozier, J., ''Sequoia 2000: Large Capacity Object Servers to Support Global Change Research,'' Sequoia 2000 Technical Report 91/1, UC Berkeley, July 1991.

[STON93] Stonebraker, M., and Olson, M., ''Large Object Support in POSTGRES'', *Proc. 9th Int'l Conf. on Data Engineering*, Vienna, Austria, April 1993 (to appear).

[WALK83] Walker, B., *et al.*, ''The LOCUS Distributed Operating System'', *Operating Systems Review* v. 17 no. 5, November 1983.

Author Information

Michael Olson is a graduate student in Computer Science at the University of California at Berkeley, where he has attracted much notoriety by wearing clothes. His research interests include managing very large data stores and a categorical ranking scheme for all the local breweries. The first of these may someday lead to a Ph.D. Reach him via US Mail at:

Mike Olson
571 Evans Hall
UC Berkeley
Berkeley, CA 94720

His electronic mail address is mao@cs.Berkeley.EDU.