ABSTRACT

Advances in mass storage technology are producing devices capable of holding terabytes of data. These new devices, often called *tertiary storage devices*, have dramatically different performance characteristics than magnetic disks. Conventional database systems include explicit dependencies on magnetic disk, and so are unsuited to manage tertiary storage devices. A layer of abstraction has been introduced between the access methods and physical storage devices in POSTGRES. This abstraction layer supports the addition of tertiary storage. An example implementation of a tertiary storage device manager is presented. Finally, the architecture of a novel file system that takes advantage of POSTGRES database system services is discussed. This file system may use any of the storage devices managed by POSTGRES.

1. INTRODUCTION

The research presented here investigates strategies for supporting massive data storage devices in a relational database management system. Such devices, often called *tertiary storage devices* because they are slower and cheaper than magnetic disk (secondary) storage, have recently become widely available commercially.

This thesis describes contributions in three key areas. First, a new internal interface for the POSTGRES database system makes it easy to add new kinds of hardware to the set of storage devices available to users. This abstraction, called the *storage manager switch*, hides characteristics of underlying storage devices, allowing the data manager to treat all devices the same.

Second, as an example of device extensibility, the POSTGRES data storage pool incorporates a 327GByte Sony write-once optical disk jukebox. This device stores relational data under POSTGRES version 4.0 at Berkeley.

Finally, a new file system has been designed and implemented on top of POSTGRES. Since it is built on top of the database system, this file system provides functionality not available in any other file system. Some of the services provided are time travel and transaction protection for user file updates. The file system also takes advantage of the storage manager switch to provide file storage on any device known to POSTGRES.

1.1. An Overview of the POSTGRES Database System

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 tuple is updated or deleted, the original tuple is marked invalid, but remains in place. For updates, a new tuple containing the new values is added to the database. By using transaction start times and a special status file that records 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 tuples must be garbage-collected from the database, and either moved elsewhere or physically deleted. If the tuples are not saved elsewhere, some historical state of the database is lost. If time travel is desired, the tuples must be saved forever somewhere. This process is referred to as *tuple archiving*.

POSTGRES includes a special-purpose process, called the *vacuum cleaner*, that archives tuples. Obsolete tuples are physically removed from the relation in which they originally appeared, and are inserted into a separate relation. This keeps the relation storing current state of the database small, but the database may still grow without bound.

1.2. The Importance of Tertiary Storage

Tertiary storage is generally an order of magnitude or more slower, both in access latency and throughput, than magnetic disk storage. In most cases, physical recording media are managed by robotics, which move them between shelves and players on demand. Moving a platter typically takes thousands of times longer than transferring a data block. However, the use of robotics multiplies the amount of storage available to a single reader, typically by two or three orders of magnitude.

The inclusion of tertiary storage in relational database systems is important for several reasons. As described in [STON87], POSTGRES relies on tertiary storage to hold historical database state. More generally, the relatively low cost per byte of tertiary storage makes it attractive to systems builders, which puts pressure on software vendors to support it. In addition, new data types like voice, video, and still images will require orders of magnitude more space than conventional data types. Finally, there is strong commercial pressure to support ever-larger conventional databases. For example, retailers and stockbrokers would store every financial transaction forever if they had sufficient space [BERN88].

1.3. Thesis Overview

This thesis describes two key extensions to the POSTGRES database system. First, POSTGRES has been made device-extensible, so that new storage technologies may be used to store data. Second, a file system has been implemented on top of the database system, providing important semantic guarantees to file system users by taking advantage of database services.

Other device-extensible database systems include Starburst [HAAS90]. However, Starburst does not provide a file system interface to non-database users. Current file systems research is exploring ways of adding functionality to existing file systems. [CABR88], [CHUT92], and [SELT90] describe transaction-protected file systems, and [ROOM92] supports time travel, but none of these systems offers both services simultaneously.

The remainder of this thesis is organized as follows. Section two gives a brief overview of the new storage manager, and the important differences it exhibits from [STON87]. Section three presents the new storage manager architecture. Section four describes the inclusion of a particular new storage device, namely a 327GByte Sony write-once optical disk jukebox. Section five introduces the Inversion file system, which is built on top of the POSTGRES database system. Section six shows performance measurements for the Inversion file system. Section seven presents conclusions, and section eight lists some directions for future research.

2. OVERVIEW OF THE NEW POSTGRES STORAGE MANAGER

Throughout this paper, the term *storage manager* refers to an interface layer in POSTGRES that accepts requests for operations on the data store. This layer determines which storage device must be used to satisfy the request and calls device-specific code to handle it. Each device has a set of interface routines for use by the storage manager. The collection of these routines for a single device is called a *device manager*.

2.1. An Arbitrary Storage Hierarchy

[STON87] proposed a two-level storage hierarchy consisting of magnetic disk and tertiary storage, as shown in Figure 1.

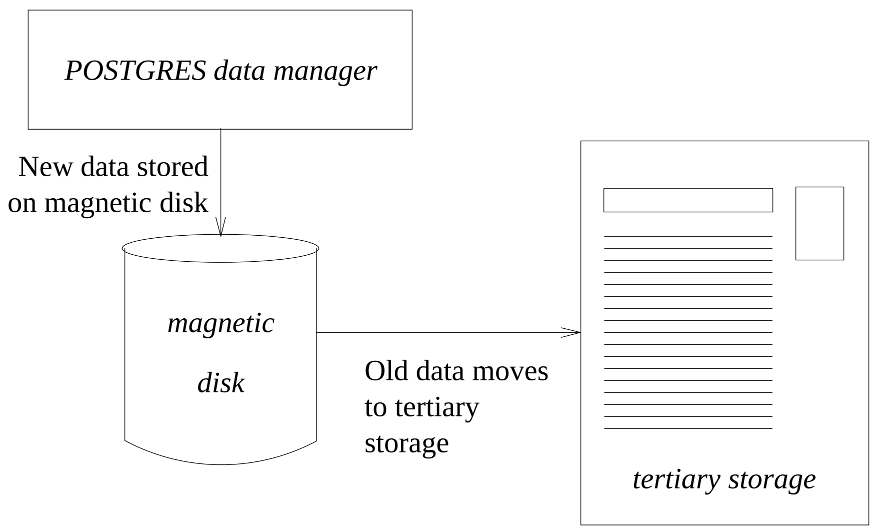

Figure 1: Architecture of the POSTGRES Storage Manager Proposed in [STON87]

The new storage manager, described here, is more general. New devices may be added to the system by writing a set of interface routines. When a user creates a new relation, he assigns it to a particular device, where it resides until it is destroyed. The relation's archive may also be assigned to any device manager. Thus the fixed two-level hierarchy proposed in [STON87] is replaced by a pool of storage, as shown in Figure 2. The POSTGRES data manager no longer communicates directly with devices to read or write data. Instead, a new interface layer handles data placement and retrieval.

Device managers currently exist for magnetic disk, battery-backed main memory, and the Sony optical disk jukebox. Other devices that may be incorporated in the future include rewritable optical disk jukeboxes and helical scan tape jukeboxes. The user may elect to locate a new relation on any of these devices. Once the relation is assigned to a device manager, its location is

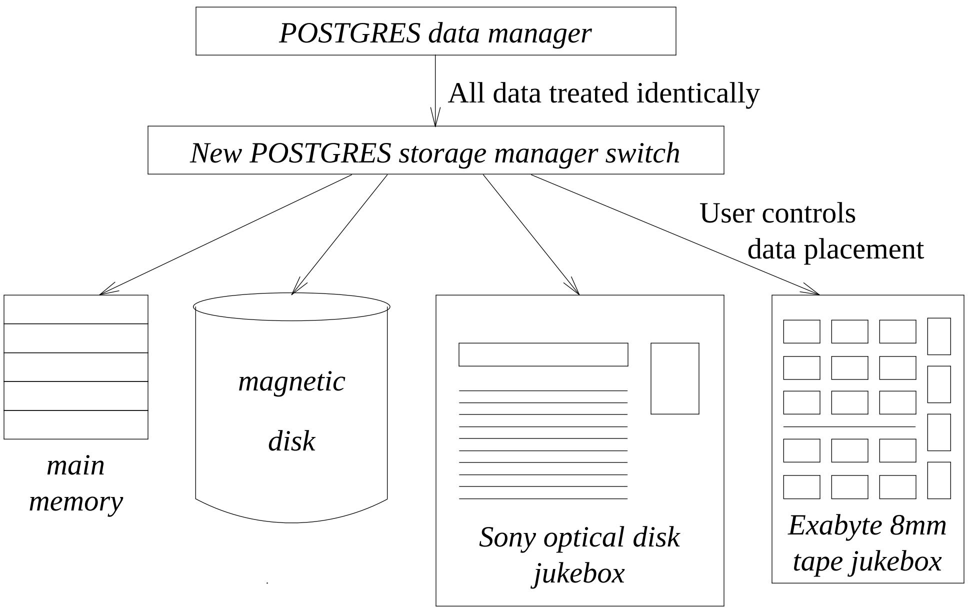

Figure 2: New POSTGRES Storage Manager Architecture

transparent to the user. It is accessed in exactly the same way, regardless of its placement.

2.2. Using Criteria other than Time to Determine Data Placement

Since archiving historical data is an important feature in POSTGRES, the vacuum cleaner uses tuple update times to move data from a ‘‘current’’ relation into a ‘‘historical’’ one. The original storage manager proposal declared that current data would be stored on a fast device (magnetic disk), and historical data on some slower device. The implicit assumption is that current data will be accessed more frequently than historical data. The desired policy is to make the most frequent accesses fastest.

The design proposed in [STON87] has several limitations. First, there exist databases from which no tuples ever get deleted. Consider, for example, a library of books. New volumes are purchased, but old ones remain in the collection forever. In this case, there is no historical data, and the current database state grows without bound.

Second, the assumption that current data will be read more frequently than historical data is not always correct. For example, accountants at the Internal Revenue Service audit tax returns two years after they are filed. In this case, the best policy would be to place all two-year-old tax returns on magnetic disk, and returns that are younger or older on tertiary storage. IRS employees could then more quickly choose two-year-old returns to audit.

Finally, users may want to specify criteria other than time for data placement. For example, consider a credit-card billing database. Tuples for customers who make many purchases should be stored in main memory, to speed up billing updates on their accounts. Tuples for less active customers might be stored on magnetic disk. The least active customers of all could have their tuples migrated to an optical disk store, since the extra cost of fetching their bills would not be incurred very often.

In all of these examples, the user is able to predict his access patterns reliably, and can instruct the database system how data should be laid out in order to minimize access times. Current systems support caches intended to speed up accesses, but these caches are not subject to user control.

The new storage manager supports a more general model than its predecessor. First, the storage manager does away with the notion of a fixed two-level data storage hierarchy. New devices with new performance characteristics may be added to the database system. When new relations are created, they may be located on any storage device, and their archives may be located on any storage device. Thus, a relation's current data may be put on the Sony write-once jukebox and its historical data on magnetic disk, if desired. This would make accesses to old data faster than accesses to new data.

Second, using the POSTGRES rule system [POTA92], the user may create a logical relation that is a view of many physical relations. These physical relations may be located on different storage devices. Further, the user may specify rules that move data from one physical relation to another in response to an update. Thus the user may choose to partition a logical relation based on an arbitrarily complex function of any of its attributes. The same strategy applies to historical data: the user may partition the single logical relation storing history among several physical relations, according to whatever policy he desires.

For example, consider a relational schema for storing tuples on employees. For each employee, the system should record an employee number, a name, a salary, and a scanned photograph for security purposes. A conventional schema would be

```
EMP(id = int, name = text, salary = float,
    picture = image)
```

where `image` is a type known to the database system, and used to store pictures.

One problem with the schema above is that it stores all data in every employee tuple on magnetic disk. Since scanned photographs are likely to be very large, storing them on magnetic disk will be expensive. In addition, it might be preferable to store salaries in memory, so that functions that operate on them can run more quickly. Since the user may have a good idea of the sizes and access frequencies of columns in the `EMP` table, he should be able to partition tuples to reduce storage costs and speed up common queries.

The new storage manager allows the user to declare a more flexible schema. The example above can be rewritten as

```
ENAME (id = int, name = text) store = "magnetic disk"
ESAL (id = int, salary = float) store = "main memory"
EPHOTO (id = int, photo = image) store = "sony jukebox"
```

The original `EMP` relation has been decomposed into three new physical relations. The following POSTGRES rule will reconstruct the logical `EMP` relation on demand:

```
on retrieve to EMP do instead
    retrieve (n.id, n.name, s.salary, p.photo)
        from n in ENAME, s in ESAL, p in EPHOTO
        where n.id = s.id and s.id = p.id
```

In addition, a rule may be declared to handle updates on the logical `EMP` relation:

```
on append to EMP do instead
    append ENAME (id = new.id, name = new.name)
    append ESAL (id = new.id, salary = new.salary)
    append EPHOTO (id = new.id, photo = new.photo)
```

Thus, the new storage manager eliminates the fixed two-level storage hierarchy. Data may be placed on a device in order to conserve space (as in the case of `EMP.photo`) or in order to speed up accesses (as in the case of `EMP.salary`). Using the POSTGRES rules system in conjunction with the new storage manager gives the user considerable freedom to horizontally or vertically partition data tables to achieve good performance while conserving expensive storage space.

3. ARCHITECTURE OF THE NEW POSTGRES STORAGE MANAGER

This section describes the architecture of the new storage manager in detail. It begins with the design changes that were necessary in the POSTGRES data manager, and is followed by the new design.

3.1. Changes to POSTGRES

Before the design and installation of the new storage manager, POSTGRES contained hard-coded dependencies on magnetic disk. These dependencies were manifested in the following ways.

- The query optimizer used a cost function that assumed the performance characteristics of magnetic disk for data stored in the database.
- The code that handled updates to the database assumed that data blocks could be overwritten in place.
- The query planner and executor used system calls on the UNIX file system to compute characteristics like the length and modification times of files storing relations.

Tertiary storage has dramatically different performance characteristics than magnetic disk. For example, throughput and seek times for optical disk are typically an order of magnitude worse than those for magnetic disk. Tape performs poorly under random access patterns. Robotics introduce sharp discontinuities into the performance curves of jukebox devices, because moving a tape or disk from a shelf to a drive takes tens of seconds.

Hard-coded dependencies on magnetic disk made it very hard to add new storage devices to the old system. Even if it were possible to store data on the new devices, the query optimizer and executor would make poor decisions for queries that accessed tertiary storage.

Thus, the most important change to POSTGRES was to extract dependencies on magnetic disk. First, the operations that the data manager carried out on storage devices were determined. From these operations, an abstraction was created that hid device differences. These abstractions are discussed in section 3.3. A set of interface routines was defined that supported this abstraction. These interface routines are listed in detail in section 3.4.

3.2. The Storage Manager Switch

The abstraction that hides device-specific characteristics from the data manager is called the *storage manager switch*. The storage manager switch is similar to the *cdevsw* and *bdevsw* interfaces of UNIX [LEFF89] and the user-supplied backing store interface in Mach [RASH88].

As discussed in the previous section, installing the storage manager switch required that the abstract operations required on the data store be identified. For example, the interface requires routines for opening relations and instantiating particular 8kByte relation blocks.

Once the abstract interface was defined, an appropriate structure (the *storage manager switch table*) was compiled into the data manager. This structure contains an entry for all existing device managers. New device types are added by editing a source file, adding appropriate entries to the storage manager switch table, and recompiling the POSTGRES database system.

Whenever a device operation is required, the data manager calls the storage manager to carry it out. Every relation is tagged in POSTGRES with the device manager on which it appears. The storage manager identifies the appropriate device and routine in the storage manager switch table, and calls the device manager to execute the routine on behalf of the data manager.

Data may be located on a particular device manager at the granularity of a relation. Physical relations may not be horizontally or vertically partitioned, but as mentioned before, the POSTGRES rules system can be used to create a partitioned logical relation.

3.3. Isolation of Device Differences

New device managers are written to manage new types of devices. For example, main memory, magnetic disk, and the Sony optical disk jukebox are all managed by different device managers. To isolate the database system from the underlying devices, device managers are required to accept and return data in the format used internally by the data manager. This means, among other things, that data pages from user relations must be accepted and returned in units of 8kBytes.

Nevertheless, there is no requirement that a device manager use the internal representation when writing a page to persistent storage. Data pages may be compressed, broken into pieces, or concatenated into larger units by a device manager, as long as the original page can be reconstructed on demand.

In addition, device managers may implement caches managed according to an appropriate policy. This means that the new POSTGRES architecture supports a multi-level cache. A main-memory cache of recently-used blocks is maintained in shared memory by the buffer manager. The buffer manager calls the storage manager to migrate pages between shared memory and persistent storage. A device manager may implement a separate cache for its own use, subject to whatever policy it likes. For example, the Sony jukebox device manager maintains a cache of *extents* (contiguous collections of user data blocks) on magnetic disk. If there is locality of reference in accesses to a Sony jukebox relation, then some requests may be satisfied without accessing the jukebox device at all. This cache will be described in more detail in section 4.2.

3.4. The Storage Manager Switch Interface

This section defines the interface presented by the storage manager switch. Although the routines presented here are specific to POSTGRES, the strategy of isolating storage management from data management is general, and similar routines should exist for conventional database management systems.

Figure 3 shows the storage manager switch data structure, `f_smgr`. This structure defines the interface routines that all device managers must define. When a new device type is added to POSTGRES, this structure is filled in with function pointers appropriate to the device, and POSTGRES is recompiled. The storage manager calls these routines in response to requests from the data manager. In some cases, the switch entry for an interface routine may be `NULL`. This means that the corresponding operation is not defined for the device in question. Unless otherwise stated in the sections that follow, all interface routines return an integer status code indicat-

```
typedef struct f_smgr {
    int         (*smgr_init)();         /* may be NULL */
    int         (*smgr_shutdown)();     /* may be NULL */
    int         (*smgr_create)();
    int         (*smgr_unlink)();
    int         (*smgr_extend)();
    int         (*smgr_open)();
    int         (*smgr_close)();
    int         (*smgr_read)();
    int         (*smgr_write)();
    int         (*smgr_flush)();
    int         (*smgr_blindwrt)();
    int         (*smgr_nblocks)();
    int         (*smgr_commit)();       /* may be NULL */
    int         (*smgr_abort)();        /* may be NULL */
    int         (*smgr_cost)();
} f_smgr;
```

Figure 3: The `f_smgr` Storage Manager Switch Structure

ing success or failure.

Table 1 summarizes the responsibilities of these interface routines. The rest of this section gives more detailed descriptions.

3.4.1. smgr_init

The routine

```
int
smgr_init()
```

is called when POSTGRES starts up, and should initialize any private data used by a device manager. In addition, the first call to this routine should initialize any shared data. Typically, this is done by noticing whether shared data exists, and creating and initializing it if it does not. If a particular device manager requires no initialization, it may define this routine to be NULL.

Routine	*Purpose*
smgr_init()	Called at startup to allow initialization.
smgr_shutdown()	Called at shutdown to allow graceful exit.
smgr_create(*r*)	Create relation *r*.
smgr_unlink(*r*)	Destroy relation *r*.
smgr_extend(*r*, *buf*)	Add a new block to the end of relation *r* and fill it with data from *buf*.
smgr_open(*r*)	Open relation *r*. The relation descriptor includes a pointer to the state for the open object.
smgr_close(*r*)	Close relation *r*.
smgr_read(*r*, *block*, *buf*)	Read block *block* of relation *r* into the buffer pointed to by *buf*.
smgr_write(*r*, *block*, *buf*)	Write *buf* as block *block* of relation *r*.
smgr_flush(*r*, *block*, *buf*)	Write *buf* synchronously as block *block* of relation *r*.
smgr_blindwrt(n_d, n_r, i_d, i_r, *blk*, *buf*)	Do a ‘‘blind write’’ of buffer *buf* as block *blk* of the relation named n_r in the database named n_d.
smgr_nblocks(*r*)	Return the number of blocks in relation *r*.
smgr_commit()	Force all dirty data to stable storage.
smgr_abort()	Changes made during this transaction may be discarded.
smgr_cost(*r*, *nblocks*, *ntuples*, *c*)	Return in *c* the estimated cost of accessing *nblock* blocks and *ntuples* tuples of relation *r*.

Table 1: The POSTGRES Storage Manager Switch Interface

3.4.2. smgr_shutdown

The routine

```
int
smgr_shutdown()
```

is analogous to `smgr_init()`, but is called when POSTGRES exits cleanly. Any necessary cleanup may be done here. If no work is required, this routine may be NULL.

3.4.3. smgr_create

The routine to create new relations on a particular device manager is

```
int
smgr_create(reln)
    Relation reln;
```

The `reln` argument points to a POSTGRES relation descriptor for the relation to be created. The device manager should initialize whatever storage structure is required for the new relation. For example, the magnetic disk device manager creates an empty file with the name of the relation.

3.4.4. smgr_unlink

The routine

```
int
smgr_unlink(reln)
    Relation reln;
```

is called by the storage manager to destroy the relation described by `reln`. Device managers may take whatever action is appropriate. For example, the magnetic disk device manager removes the relation's associated file and archive, and reclaims the disk blocks they occupy. Since the Sony write-once drive cannot reclaim blocks once they are used, the Sony jukebox device manager simply marks the relation as destroyed and returns.

3.4.5. smgr_extend

When a relation must be extended with a new data block, the data manager calls

```
int
smgr_extend(reln, buffer)
    Relation reln;
    char *buffer;
```

to do the work. The buffer is an 8kByte buffer in the format used by the data manager internally. The data manager assumes that 8kByte data blocks in relations are numbered starting from one. Thus `smgr_extend()` must logically allocate a new block for the relation pointed to by `reln` and store some representation of `buffer` there.

3.4.6. smgr_open

The routine

```
int
smgr_open(reln)
    Relation reln;
```

should open the relation pointed to by `reln` and return a file descriptor for it. The notion of file descriptors is a holdover from the pre-storage manager switch days, and is no longer necessary. If file descriptors make no sense for a given device, then the associated device manager may return any non-negative value. A negative return value indicates an error.

3.4.7. smgr_close

When the data manager is finished with a relation, it calls

```
int
smgr_close(reln)
    Relation reln;
```

The device manager may release resources associated with the relation pointed to by `reln`.

3.4.8. smgr_read

To instantiate an 8kByte data block from a relation, the data manager calls

```
int
smgr_read(reln, blocknum, buffer)
    Relation reln;
    BlockNumber blocknum;
    char *buffer;
```

As stated above, 8kByte blocks in a relation are logically numbered from one. The storage manager must locate the block of interest and load its contents into `buffer`.

3.4.9. smgr_write

The routine

```
int
smgr_write(reln, blocknum, buffer)
    Relation reln;
    BlockNumber blocknum;
    char *buffer;
```

writes some representation of `buffer` as block number `blocknum` of the relation pointed to by `reln`. This write may be asynchronous; the device manager need not guarantee that the buffer is flushed through to stable storage. Synchronous writes are handled using the `smgr_flush` routine, described below. As long as this routine guarantees that the buffer has been copied somewhere safe for eventual writing, it may return successfully.

The buffer is an 8kByte POSTGRES page in the format used by the data manager. It may be stored in any form, but must be reconstructed exactly when the `smgr_read` routine is called on it.

3.4.10. smgr_flush

This routine synchronously writes a block to stable storage. The POSTGRES data manager almost never needs to do synchronous writes of single blocks. In some cases, however, like the write of the transaction status file that marks a transaction committed, a single block must be written synchronously. In this case, the data manager calls the device manager's `smgr_flush` routine. The interface for this routine is

```
int
smgr_flush(reln, blocknum, buffer)
    Relation reln;
    BlockNumber blocknum;
    char *buffer;
```

The behavior of `smgr_flush` is almost exactly the same as that of `smgr_write`, except that `smgr_flush` must not return until `buffer` is safely written to stable storage.

3.4.11. smgr_blindwrt

Under normal circumstances, POSTGRES moves dirty pages from memory to stable storage by calling the `smgr_write` routine for the appropriate storage manager. In order to call `smgr_write`, the buffer manager must construct and pass a relation descriptor.

In certain cases, POSTGRES is unable to construct the relation descriptor for a dirty buffer that must be evicted from the shared buffer cache. This can happen, for example, when the relation descriptor has just been created in another process, and has not yet been committed to the database. In such a case, POSTGRES must write the buffer without constructing a relation descriptor.

The buffer manager can detect this case, and handles it using a strategy called ‘‘blind writes.’’ When a blind write must be done, the buffer manager determines the name and object id of the database, the name and object id of the relation, and the block number of the buffer to be written. All of this information is stored with the buffer in the buffer cache. The device manager is responsible for determining, from this information alone, where the buffer must be written. This means, in general, that every device manager must use some subset of these values as a unique key for finding the proper location to write a data block.

The interface for this routine is

```
int
smgr_blindwrt(dbname, relname, dbid, relid, blkno,
              buffer)
    Name dbname;
    Name relname;
    ObjectId dbid;
    ObjectId relid;
    BlockNumber blkno;
    char *buffer;
```

It is expected that most device managers will write data much more efficiently using `smgr_write` than `smgr_blindwrt`.

3.4.12. smgr_nblocks

The routine

```
int
smgr_nblocks(reln)
    Relation reln;
```

should return the number of blocks stored for the relation pointed to by `reln`. The number of blocks is used during query planning and execution.

Since this is a potentially expensive operation, the storage manager maintains a shared-memory cache of relation sizes recently computed by device managers.

3.4.13. smgr_commit

When a transaction finishes, the data manager calls the `smgr_commit` routine for every device manager. This routine must flush any pending asynchronous writes for this transaction to disk, so that all relation data is on persistent storage before the transaction is marked committed. In addition, the device manager can release resources or update its state appropriately.

This routine may be NULL, if there is no work to do for a device manager at transaction commit. If it is supplied, its interface is

```
int
smgr_commit()
```

There is one important issue regarding transaction commit that should also be mentioned. Much research has been done in database systems to support *distribution*, or databases that span multiple computers in a network. Although POSTGRES is not a distributed database system, it does support *data distribution* by means of the storage manager switch and device managers.

Data distribution means that different tables, or parts of tables, may reside on storage devices connected to different computers on a network. POSTGRES does all data manipulation at a single site, but the data it works with may be distributed. The data manager will call the device manager, which can operate on a local device, or use the network to satisfy requests on a remote device. In fact, the Sony jukebox device manager works correctly whether it runs on the host computer to which the jukebox is connected, or on some other computer on the Internet.

At transaction commit time, the data manager informs each device manager that all its data must be on stable storage. Each device manager flushes any pending writes synchronously, and returns. Once all device managers have forced data to stable storage, the data manager synchronously records that the transaction has committed by writing a ten-byte record to the *transaction status file* on one of the storage devices. If the system crashes before the commit record is stored in the transaction status file, POSTGRES will consider the transaction to be aborted when the transaction's changes are encountered later.

Thus the `smgr_commit` interface supports data distribution without using a traditional multi-phase commit protocol ([KIM84], [SKEE81]). In particular, device managers are not notified whether a transaction actually commits, once they have forced data to stable storage.

3.4.14. smgr_abort

If a transaction aborts, the data manager calls `smgr_abort` in place of `smgr_commit`. In this case, it does not matter if changes made to user relations during the aborted transaction are reflected on stable storage. Device managers may deschedule pending writes, as long as the writes include no data from other transactions.

This routine may be NULL, if there is no work to do for a device manager at transaction abort.

3.4.15. smgr_cost

Query optimization requires cost estimates for different access paths on relations [SELI79]. Since the cost function depends on the performance characteristics of the device storing the relation, each device manager must provide a cost estimator.

The interface is

```
int
smgr_cost(reln, nblocks, ntuples, costOutP)
    Relation reln;
    int nblocks;
    int ntuples;
    double *costOutP;
```

This function should compute an estimate of the cost of instantiating `nblocks` blocks and `ntuples` tuples from the relation pointed to by `reln`. The `costOutP` parameter is a pointer to a location for the returned cost. An example cost function for a tertiary storage device is given in section 4.

3.5. Architecture Summary

The new storage manager architecture assigns a particular device manager to manage each device, or class of devices, available to the database system. A well-defined set of interface routines makes relation accesses device-transparent.

The new architecture uses a *storage manager switch* to record the interface routines for all the device managers known to the system. An earlier version of this architecture stored these interface routines in a database relation, and made the set of device managers dynamically extensible. All existing device managers, however, use POSTGRES shared memory and spinlocks. Since existing device managers could not be developed outside the POSTGRES data manager, the mechanism for doing so was eliminated.

So far, three device managers exist. The magnetic disk device manager is only 440 lines of C code, including comments. The main memory device manager is 600 lines, most of which is a dynamically-extensible hashing package to make block lookups fast. The most complicated device manager by far — the Sony optical disk write-once jukebox device manager — is less than 3000 lines of code. This complexity is largely due to the fast recovery and aggressive sharing techniques used. The design and implementation of the Sony jukebox device manager are described in the next section.

4. IMPLEMENTATION OF THE SONY JUKEBOX DEVICE MANAGER

The tertiary storage device currently supported by POSTGRES is a jukebox of 50 double-sided write-once platters with a total capacity of 327GByte. A robotic arm moves platters between a bank of shelves and two read/write decks. The device is connected via a SCSI bus to a host computer. A complete description of the hardware can be found in [SONY89a] and [SONY89b].

POSTGRES uses this device via a client/server protocol described in section 4.1. The Sony jukebox device manager caches optical disk blocks on magnetic disk for faster access; the cache management policy, and some important issues in the cache design, are described in section 4.2. Section 4.3 describes the various shared-memory caches used by the jukebox device manager. Sections 4.4, 4.5, and 4.6 respectively discuss issues raised by the write-once properties of the device, how commits and aborts are handled, and the cost function provided in `sj_cost`.

4.1. POSTGRES Access to the Sony Jukebox

The Sony jukebox must be physically connected to a host computer over a SCSI bus. This host computer is not necessarily the computer on which users want to run POSTGRES. In order to make the device accessible from other computers, a server on the host accepts requests to mount platters and to read and write data over the network [MCFA91]. The network connection uses TCP/IP over BSD sockets [LEFF84].

The server manages a queue of requests from multiple clients, so POSTGRES is not necessarily the only user of the jukebox. When requests on more than two platters appear in the request queue, the server provides the illusion that all are mounted simultaneously by moving platters among the two drives and shelves as necessary. The request queue managed by the server is completely external to POSTGRES, so no information about the queue's length or contents can be used when optimizing queries. A future version of the jukebox server will make queue status available to clients. This will allow POSTGRES to estimate query costs more accurately than it does now (see section 4.6 for more details), and will result in better query optimization.

Figure 4 shows the jukebox server architecture. In this figure, there are three clients of the jukebox server; POSTGRES and two others. These others are not clients of POSTGRES; they are processes that have their own data stored on optical platters, which they manage themselves. Clients of the database system connect to POSTGRES, which manages relational data stored on platters for them.

The jukebox server uses platter names and ownership to provide some data protection, and to keep users other than POSTGRES from reading or writing POSTGRES platters. POSTGRES must be the only user of platters that it owns, so that it can handle block allocation cleanly.

4.2. Cache Design

Transfers from the jukebox to the database system are expensive to set up and slow to execute. This is because the device itself is slow, and because there may be a network between the database system and the storage device. In order to mask some of the latency, the Sony jukebox

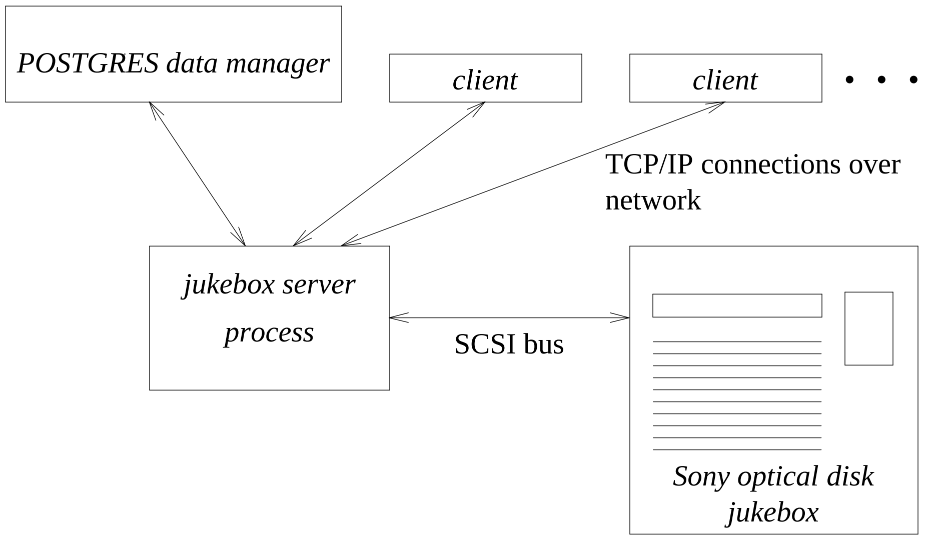

Figure 4: Sony Jukebox Client/Server Process Architecture

device manager caches blocks from optical disk on magnetic disk. This cache is used for both reads and writes; writes are flushed to the Sony jukebox in LRU order when dirty cache entries are reclaimed. This section describes the cache's structure, contents, and replacement policy.

4.2.1. Persistent Magnetic Disk Cache

The magnetic disk cache of optical disk blocks is *persistent*. This means that its contents are valid across executions of POSTGRES. Logically, data stored either in the cache or on an optical platter is considered to be on the Sony jukebox. This is distinct from write-through techniques; writes to the magnetic disk cache must survive system crashes, but need not be flushed immediately to an optical disk platter.

There were two motivations for this decision. First, a persistent cache means that the cache need not be populated when a data manager starts up, and simplifies issues of sharing, since all instantiations of POSTGRES use the same cache. Cache population is extremely expensive, since transfers from the jukebox are so slow. If there is any locality of reference between executions of the database system, then the persistent cache should speed up queries that use the jukebox.

The second motivation for persistence is to allow POSTGRES to delay writes to the jukebox device for as long as possible, to avoid paying unnecessary transfer costs. Data written to the jukebox device manager first hits the magnetic disk cache. If the cache is persistent, then it need not be write-through. Blocks may remain in the magnetic disk cache, since they will be found by subsequent invocations of POSTGRES.

Persistence introduces substantial complexity into cache management. For reasons described in section 4.2.2, user blocks are stored as contiguous chunks of data in the magnetic disk cache, and cache metadata is stored separately. Since individual writes to the cache are very large, atomicity of writes is a problem. Since the metadata and the data it describes are stored separately, the atomicity problem is exacerbated.

In general, the problem is that the cache data and metadata must be written out separately. If one, but not both, of the writes completes before a system crash, then the cache is in an inconsistent state. This state must be detected and repaired before the associated data is used.

To solve this problem, the Sony jukebox device manager uses a *link token* technique like the one described in [SULL92]. The cache data and its associated metadata are tagged with a unique number, which is generated by the POSTGRES object id service. Whenever a cache entry is used, the metadata and data are checked to be sure that they match. If not, then the cache's internal locking scheme guarantees that the inconsistency was caused by a crash, and the correct state must be recovered.

The recovery strategy is to throw away both the cache data and metadata and reinstate it from the optical disk jukebox. The order in which cache operations are performed guarantees that any inconsistent data in the cache is either present on the Sony jukebox, or is new data from a transaction in progress when the system crashed. In either case, the inconsistent cache data may be discarded. If it is on the jukebox, then it may be reinstated later. If it was new data from a transaction in progress at the time of a system crash, then it is invalid and need not be recovered.

This cache management strategy supports fast recovery, an important feature of POSTGRES [STON87], [STON90]. Since cache inconsistencies are discovered and repaired on first use, no special recovery code needs to run when POSTGRES restarts after a crash.

4.2.2. Device Manager Block Allocation and Caching

The Sony jukebox device manager allocates space to relations in units of *extents*. The extent size is configurable when POSTGRES is compiled. The current implementation uses 32 8kByte blocks, or 256kBytes of user data. When a new block is requested for a relation (via `smgr_extend`) and there is no existing extent with free space, a new extent is allocated for it. Subsequent `smgr_extend` calls consume subsequent blocks in the new extent.

Large extents were chosen for several reasons. First, they favor sequential access patterns by guaranteeing contiguity of user data. Second, large extents favor large transfers, and large transfers are more efficient than smaller ones. This is due to the high cost of setting up a transfer on the Sony optical disk jukebox. Finally, the device manager maintains an index to locate extents quickly. A single index entry can point at an entire extent, so large extents require fewer index entries than small ones, saving space.

Whenever possible, the Sony jukebox device manager attempts to locate multiple extents for a single relation on the same side of the same platter. If the platter is full, the next choice is to locate new extents on a new platter. The last choice is to put them on the reverse side of the same platter. This is because reading from one side of a platter, flipping it over, and reading from the other side is more expensive than reading from one side of two different platters. Only one side of a single platter may be mounted at a time. Since there are two read/write decks in the jukebox, two separate platters can be mounted simultaneously.

The current implementation moves user data between the Sony jukebox and the magnetic disk cache in units of full extents. The transfer size is configurable, and can be any size less than or equal to a full extent. Full extents favor locality of reference. The next section describes the replacement policy for cached extents, and addresses some issues raised by the write-once nature of the jukebox.

4.2.3. Replacement Policy for Cached Extents

Extents stored in the magnetic disk cache are flushed to the Sony jukebox in least-recently-used order. When space in the cache is required for a new extent, the oldest unused extent is evicted. If the extent is dirty (that is, some block in it has been modified since the extent entered the cache), it must be written to the Sony jukebox. Clean extents may be evicted without being written.

The important complications in this scheme come about because of the write-once nature of the Sony optical disk jukebox. If only part of an extent has been allocated and written since it appeared in the cache, then only the parts that contain new data should be written to the jukebox. The unused part of the extent must be left free on the jukebox for use later. This requires the device manager to do transfers in units smaller than complete extents under some circumstances.

A worse problem is that the magnetic disk cache may not always contain the most recent version of an extent. This situation can come about because the POSTGRES buffer manager has a cache of relation blocks in shared memory. If some dirty blocks in the shared memory cache have not been written to the magnetic disk cache, then incorrect or incomplete blocks may be written to the jukebox. Once written, these blocks can never be replaced. Thus the jukebox device manager must guarantee that it has the latest copies of all the blocks in the extent before it attempts to write them to the Sony jukebox.

These two problems have distinct solutions. Writing partial extents to the jukebox is straightforward; among the information stored in the magnetic disk cache metadata is a map of the dirty pages in an extent, so only those pages need to be written. Partial writes are less efficient than full-extent writes, but are only necessary if cached extents must be forced out before they are filled. In the long term, extents should fill up with data, so reads will eventually transfer whole extents.

Locating the latest version of every block in an extent required a change to the POSTGRES buffer manager. A new interface routine was added to the code that maintains the shared memory buffer cache.

```
int
DirtyBufferCopy(reln, blocknum, buf)
    Relation reln;
    BlockNumber blocknum;
    char *buf;
```

If block `blocknum` of relation `reln` appears in the buffer pool and is dirty, then it is copied to `buf`, marked clean, and `DirtyBufferCopy` returns `TRUE`. Otherwise, `DirtyBufferCopy` just returns `FALSE`.

Just before writing a dirty segment to optical disk, the Sony jukebox device manager calls `DirtyBufferCopy` once for each buffer that appears in the extent. This assembles the latest copy of the dirty extent. This is guaranteed to behave correctly, since the Sony jukebox device manager acquires an exclusive lock on the extent beforehand. The exclusive lock guarantees that no user can update blocks that appear in the extent until the lock is released.

One final issue in cache replacement is detecting and recovering from crashes during flushes of dirty extents. Consider the case in which a dirty extent is being written to the Sony jukebox, and the system crashes before the write completes. In that case, the cache data and metadata are still consistent, but some of the blocks marked ‘‘dirty’’ in the cache already appear on the jukebox.

Because the jukebox obeys the write ordering specified by POSTGRES, this case is easy to detect and handle. Just before writing a dirty extent, the Sony jukebox storage manager attempts to read the first block it will write. If the read succeeds, then part of the extent was successfully written before a crash. The amount written can be determined by reading the extent from the Sony jukebox, and the remaining dirty blocks can be written safely. Attempts to overwrite blocks on the Sony jukebox actually destroy the data stored there, so the jukebox device manager must be very careful about this.

4.3. Use of Shared Memory

The fact that the magnetic disk block cache is shared by all POSTGRES processes requires that the Sony jukebox device manager coordinate cache operations with its peers. Specifically, only a single POSTGRES process may update a particular cache region at a time. In addition, cache metadata is stored in memory for efficiency. Processes that want to examine cache metadata can do so without accessing the magnetic disk version. Only processes that change the metadata must force their changes to disk.

To support sharing, the Sony jukebox device manager uses the POSTGRES shared memory and semaphore abstractions. Cache metadata is stored at a named location in shared memory, and all concurrent processes can attach this region to their address spaces. Semaphores with well-known names exist to control exclusive access to objects in shared memory. The rest of this section describes what is stored in shared memory and how access to it is controlled.

As mentioned in section 4.2, the magnetic disk block cache is logically divided into *user data* and *metadata*. The user data portion consists of a control block followed by a sequence of data blocks. The control block stores the name and object identifier of the relation owning the data blocks, the link token described in the previous section, and some extra administrative information. The metadata identifies dirty data and unallocated data blocks and stores the link token, among other information. The *extent* allocated by the jukebox storage manager is the user data portion, including the control block. This entire extent is read from and written to the jukebox. The metadata is never stored on the jukebox. It is maintained only on magnetic disk, and is recreated from the control block when an extent moves from an optical disk platter into the magnetic disk cache.

Both the metadata and the user data portions of the cache are persistent, as defined in section 4.2.1. When the first POSTGRES process starts up after a crash, the Sony jukebox device manager detects the fact that its shared memory is not yet initialized and loads the cache metadata into shared memory. This requires an exclusive lock on the cache metadata for the duration of the load. Such a lock is acquired using the POSTGRES semaphore abstraction. As the metadata is loaded, a hash table is constructed in shared memory to make lookups of particular extents (by relation, database, and block numbers) fast. This initialization takes on the order of fifty milliseconds on a DECstation 3100 running Ultrix 4.2.

Once the metadata is loaded into shared memory and the hash table is initialized, the exclusive lock is released. This is the only case in which an exclusive lock is held on a shared data structure during a disk access.

When a POSTGRES process wants access to a particular cached extent, it takes the following steps:

(1) It acquires an exclusive lock on the hash table and looks up the extent. The exclusive lock keeps concurrent processes from changing the hash table during lookup. This lock need not exclude other readers, but does so (for the sake of simplicity) in the current implementation. For the purposes of this discussion, assume that the extent is present in the cache.

(2) It releases the exclusive lock on the hash table.

(3) It acquires an exclusive lock on the particular extent. Every cache entry has an associated semaphore that controls access to it. Since semaphores are a scarce resource, the POSTGRES shared lock table could be used instead.

(4) It verifies that the extent is the one desired. The extent may have changed, for example, if some concurrent process evicted the extent and replaced it with another before the exclusive lock on the extent was acquired. Another possibility is that the cache is inconsistent due to a crash. It is possible to detect and recover from this case as described in section 4.2.1.

(5) POSTGRES then sets the `IO_INPROGRESS` flag on the extent's metadata and releases the exclusive lock. Concurrent processes will notice that the `IO_INPROGRESS` flag is set and defer their accesses until it is cleared. Sleeping and wakeup are coordinated using spinlocks or semaphores, depending on operating system support.

(6) Finally, it releases the exclusive lock on the extent.

At this point, the extent may be read or written freely.

If the extent sought does not yet appear in the cache, POSTGRES must evict an existing extent and load the one desired. The process for doing so is similar to that for reading an extent, except that there is a second round of locking to update the cache metadata and the shared hash table. In addition, the code is careful to detect and handle race conditions, such as the case where two concurrent processes instantiate the same extent in the cache at the same time.

The cache management and locking protocol are loosely based on that used by the POSTGRES buffer manager. The implementation holds exclusive locks on objects for as short a period as possible, and never holds an exclusive lock during an I/O (except during initialization after a crash, as described above). In addition, it guarantees that different processes can update different parts of the cache simultaneously.

The complexity involved in managing the cache for such aggressive sharing is substantial. This, along with cache persistence, are the primary reasons that the Sony jukebox device manager requires 3000 lines of code.

4.4. Managing Write-Once Media

Another source of complexity in the Sony jukebox device manager is the fact that the optical platters are write-once. Although [STON87] claims that the POSTGRES storage manager is no-overwrite, it would be more correct to call it ‘‘seldom-overwrite.’’ There are circumstances in which tuples or blocks are overwritten in place, and these cases cause problems for the Sony jukebox storage manager.

One such case is when a tuple is deleted or replaced. The no-overwrite storage manager in POSTGRES reserves some space in the tuple to mark it with the transaction ID of the updating transaction. Marking a tuple constitutes an overwrite-in-place of the existing tuple, and cannot be supported by the Sony jukebox device manager.

Another case requiring overwrite is when the POSTGRES vacuum cleaner runs. This process fills in the commit times of transactions affecting tuples. Commit times are recorded in fields in the tuples themselves, and so this also constitutes overwrite-in-place. Furthermore, the vacuum cleaner assumes that it can physically remove expired tuples from data pages. This is not true for write-once media.

The present implementation of the Sony jukebox device manager fails to solve either problem. It detects attempts to overwrite existing pages, and returns a failure error code to the caller. This means that relations stored on the Sony jukebox cannot be vacuumed, and that tuples stored on the jukebox can never be deleted. Since vacuuming is only a performance optimization, failure to support it does not lead to incorrect behavior. The fact that jukebox tuples are permanent is a more serious flaw. Essentially, this makes physical device characteristics visible to the user. The design goal of the storage manager switch was to make relations location-transparent, and this property violates location transparency.

A possible solution would be to change the representation of pages stored inside the Sony jukebox device manager. The pages would be separated into overwritable parts (updating transaction ID and transaction commit times for tuples on the page), which would be stored on magnetic disk, and non-overwritable parts (the tuple data), which would be stored on the jukebox. The overwritable parts of a tuple consume thirteen bytes of storage, so this strategy will increase the amount of magnetic disk space required. Under this scheme, conventional POSTGRES data pages could be reassembled by the Sony jukebox device manager on demand.

Another problem raised by write-once storage devices is the fact that data blocks in POSTGRES relations do get overwritten in place. For example, a data block is overwritten every time a new tuple is added to it. As above, this causes problems for the Sony jukebox device manager.

In this case, the solution is straightforward. The current version of POSTGRES does not cluster data in relations, so all insertions to heap relations happen on the relation's highest-numbered block. The Sony jukebox device manager stores the last block of every relation it manages on magnetic disk. When a new block is allocated, the old last block is moved to the optical disk jukebox, and the new one replaces it on magnetic disk. Note that the Sony jukebox device manager does not use the magnetic disk device manager for this; it operates on the magnetic disk directly.

The current system flags write-once device managers in the storage manager switch, and the data manager is careful not to vacuum relations stored on these. In addition, since indexed access methods generally require block overwrite-in-place at random locations in the relation, these cannot be located on write-once device managers. The correct solution is to support logical overwrite-in-place on write-once device managers by writing software to handle overwrites-in-place, but this has not yet been done in the Sony jukebox device manager.

4.5. Commit and Abort Processing

When a transaction commits or aborts, the data manager calls the `smgr_commit` or `smgr_abort` interface routines to guarantee that all data managed by all device managers is on stable storage.

Writes to the physical jukebox device are synchronous and unbuffered, so no special processing needs to be done for them. Writes to magnetic disk may be captured by the file system buffer cache, and must be forced through at transaction commit. In addition, the Sony jukebox device manager must flush changes to stable storage even in case of a transaction abort. This is because the device manager may write a page to disk on behalf of some other transaction. For example, a page in the buffer cache may be dirtied by one transaction, and written by a device manager in another transaction which evicts the page from the cache.

To guarantee that all changes are on stable storage, the Sony jukebox device manager keeps track of all magnetic disk file descriptors on which it has done writes. At transaction commit or abort, the jukebox device manager issues an `fsync` system call on each of these file descriptors. This forces all pending writes to disk. On completion, the device manager returns successfully.

4.6. Query Cost Estimation

The POSTGRES optimizer is based on the design proposed by [SELI79] and is described in detail in [FONG86]. To support multiple devices, the query optimizer must call the `smgr_cost` function for the device managers that manage relations under consideration. The optimizer uses selectivity functions defined by the access methods to estimate the number of blocks and number of tuples in the relation that the query plan will touch. These values are passed to `smgr_cost`. This section describes the cost estimation function provided by the Sony jukebox device manager.

[SELI79] proposes a cost estimation function

$$cost = W_1 \cdot nblocks + W_2 \cdot ntuples$$

to estimate the cost of a particular access path to a relation. W_1 is a coefficient that approximates the cost of instantiating a block from the storage device, and W_2 is a coefficient that approximates the CPU cost of processing a single tuple. These coefficients are set empirically.

The Sony jukebox storage manager adds a term to this equation. It uses

$$cost = W_0 \cdot npswitches + W_1 \cdot nblocks + W_2 \cdot ntuples$$

The new term estimates the cost of platter switches associated with the access path under consideration. W_0 approximates the cost of a single platter switch, and *npswitches* estimates the number of platter switches required.

W_0 is set empirically. A platter switch on the Sony jukebox takes, on the average, 6.7 seconds. This is roughly the amount of time it takes to read 5MByte, or 625 8kByte data pages, from an optical platter. Thus $W_0 = 625 W_1$. W_1, in turn, is set empirically to 40 times the cost of instantiating a magnetic disk block. In general, device managers in POSTGRES normalize their cost function coefficients to the values used by the magnetic disk device manager.

The number of platter switches, *npswitches*, is estimated in the following way. Let $nblocks_r$ be the total number of blocks stored in the relation under consideration. Then the probability that any particular platter must be loaded is assumed to be

$$\frac{nblocks}{nblocks_r}$$

where *nblocks* is the parameter passed in by the query optimizer. This assumes that the probability of touching a given block is uniform over all platters storing data for the relation; that is, that every platter storing data for the relation has an equal chance of being loaded to satisfy the query. In general, this is an invalid assumption, since it ignores the number of blocks stored on a given platter, but it simplifies cost estimation substantially.

The number of platters storing data for the relation, *nplatters*, is computed by examining the system catalogs. Then the number of platter switches is computed as

$$npswitches = \max\left(1 , \frac{nblocks}{nblocks_r} \cdot nplatters \right)$$

This formula assumes that at least one platter must be mounted in order to satisfy a query.

The cost function described in this section favors access plans that touch fewer blocks of relations on the Sony jukebox. This cost function provides only a rough estimate query's actual cost. The contents of the magnetic disk cache are ignored, and no attempt is made to determine whether a desired platter is already mounted in the jukebox. In addition, the actual distribution of data pages across platters is not considered. Section 8 suggests some improvements to the existing cost function.

4.7. Summary of Sony Jukebox Device Manager Implementation

This section has described the design of the Sony optical disk jukebox device manager. The most important features of this device manager are the magnetic disk cache of recently-used optical disk blocks and the write-once nature of the recording medium. These two features make the device manager more complex than the magnetic disk or main memory device managers.

The Sony jukebox device manager demonstrates that the storage manager switch abstraction was successful. New storage devices may be integrated into POSTGRES by a sophisticated programmer with knowledge of the database system's internals.

The next section introduces the Inversion file system, which was implemented on top of POSTGRES and takes advantage of the storage manager switch to provide a file system on any device supported by POSTGRES.

5. DESIGN OF THE INVERSION FILE SYSTEM

Conventional database systems are generally built on top of file systems [STON92]. This section describes the design and implementation of a file system built on top of the database system. This file system, called ‘‘Inversion’’ because the conventional roles of the file system and database system are inverted, runs under POSTGRES version 4.0. It supports file storage on any device managed by POSTGRES, and provides useful services not found in many conventional file systems.

Current file systems researchers are concentrating on providing new services to administrators and users. For example, Episode [CHUT92] embeds transaction protection in the file system directly. However, transactions protect only file system metadata, to speed recovery after crashes, and are not available to users. [SELT90] proposes embedding user-level transaction support directly in existing file systems, and QuickSilver [CABR88] is an early example of such a system. Other researchers are investigating support for time travel. For example, Plan 9 [PIKE90] and 3DFS [ROOM92] allow the user to see past states of the file system. However, most such systems provide coarse temporal granularity. Both Plan 9 and 3DFS take periodic snapshots of the file system, and states that existed between snapshots are not recoverable.

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 is 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. LFS [ROSE91] uses a log to provide similar guarantees about recovery time, but does not support transactions.

In addition, files in the Inversion file system may be located on any device manager in POSTGRES. This means that the Inversion file system can span multiple devices (and device types) transparently.

A less conventional, but nonetheless important, feature of Inversion is that files are strongly typed. Since POSTGRES supports the dynamic loading of user code for execution in the data manager, this means that users can arrange for their applications to run in the file system manager itself, rather than in a separate address space. By exploiting typing, operations may be defined that can be applied to certain groups of files. For example, object files can be treated differently from text files. Locus [WALK83] supports typed files, but the set of file types is small and difficult to extend.

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 systems structure and contents by formulating simple POSTQUEL queries. In addition, indices may be defined to make selected queries run faster, at the user's discretion.

Ad hoc queries are especially powerful because they support attribute-based naming of files [SECH91]. For example, in Inversion, it is possible to store revision control information about a file in a relation. The user can then formulate queries on the database to find files based on revision control information. Since arbitrary schemas are permitted, any attribute of interest may be stored in the database and used to look up files.

In summary, the Inversion file system provides services not available in conventional file systems, including

- transaction-protected file updates,

- fine-grained time travel,
- device independence,
- strong typing of files, and
- *ad hoc* queries on the file system state.

The rest of this section describes the design and implementation of Inversion.

5.1. Decomposing Files into Relations

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 5 shows the schema used to store file data in POSTGRES relations.

For every file, a uniquely-named relation is created. File data is collected into chunks slightly smaller than 8kBytes. The size of the chunk is calculated so that a single tuple 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 page size inherited from the data manager survives.

When a user writes a new data chunk to a file, a tuple is created consisting of the *chunk number*, or index of this chunk into the file, and the data chunk. This tuple is appended to the

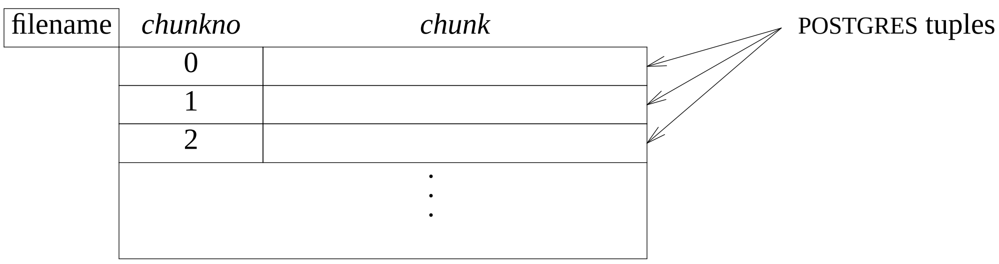

Figure 5: Decomposition of files into relations in Inversion

relation storing the file.

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. In order to be compatible with existing UNIX filesystems, Inversion uses longwords to communicate a file's current read or write position. This limits Inversion files to 2GBytes in size. This constraint is easy to remove, at the cost of compatibility with existing file systems.

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 relation storing the file. Inversion does not know anything about devices.

When a file is modified, the tuples storing changed chunks are replaced in the normal way. In order to speed up seeks on files, Inversion maintains a Btree index on the chunk number attribute.

5.2. Namespace and Metadata Management

Inversion stores the file system namespace in a class

```
pg_largeobject(poid, fname)
```

where `poid` is the `pg_largeobject` object identifier of the parent directory, and `fname` is the file name. The object identifier associated with the `pg_largeobject` tuple for the file is its filesystem-wide unique identifier.

The names of the relations containing file data are computed from the file's `pg_largeobject` object identifier. If the object identifier is 12345, then the relation containing the file data is `inv12345`, and the block number index on the relation is called `inx12345`.

In addition to namespace management, most file systems manage other metadata associated with a file. For example, the file's owner, size, and last access, modification, and creation times are typically stored by the file system. Inversion also maintains this data for all files. The owner and creation time for a file may be determined by examining the `pg_class` tuple for the relation in which the file is stored.

In the current version of POSTGRES, relations are append-at-end only, so frequently changing data consumes space quickly. Since file sizes, last access times, and last modification times typically change frequently, Inversion does not store them in a relation, in order to conserve space. Instead, these attributes are stored in a hash table on magnetic disk, and updated in place. Once POSTGRES can cluster data in relations and reuse free space on data pages, these items should be stored in a relation.

5.3. File System Management Issues

Two important issues in file system management are the policy for caching filesystem blocks for fast access and the policy for laying files out on storage devices. The current implementation of Inversion does not address either of these issues.

POSTGRES supports a multi-level cache, since the buffer manager caches buffers in shared memory and device managers (like the Sony jukebox device manager) can implement caches for their own use. These caches may be distributed among different storage technologies. For example, the Sony jukebox device manager maintains its cache of recently-used extents on magnetic disk. The POSTGRES shared buffer cache resides in main memory. This separation of cache location solves a commonly-cited problem of multi-level caches. Since the caches are on different devices, they do not compete for resources.

Despite the fact that device managers can provide custom caching, there is no special cache of file system data blocks maintained by Inversion. Inversion is built entirely above the storage manager switch layer, and so has no knowledge of a file's location. It would have been possible to create an in-memory cache of file system blocks, but this is precisely the service already supplied by the POSTGRES shared buffer cache.

Since Inversion does not control file location, it cannot control placement of file blocks on storage media. Device managers enforce policies for block layout in the current implementation. At some point, POSTGRES implementors will have to move this policy higher in the system, since support for data clustering in relations requires that the data manager make decisions on block layout. In either case, no support currently exists for enforcement of a layout policy in the Inversion file system code.

6. 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 will use Inversion as a network file server. The system configuration evaluated here is the same as will be used by Sequoia researchers. Inversion is compared to NFS [SAND85] running on identical hardware.

6.1. System Configuration

Inversion was installed on a DECstation 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.

6.2. 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.

6.3. Benchmark Results

Figure 6a 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

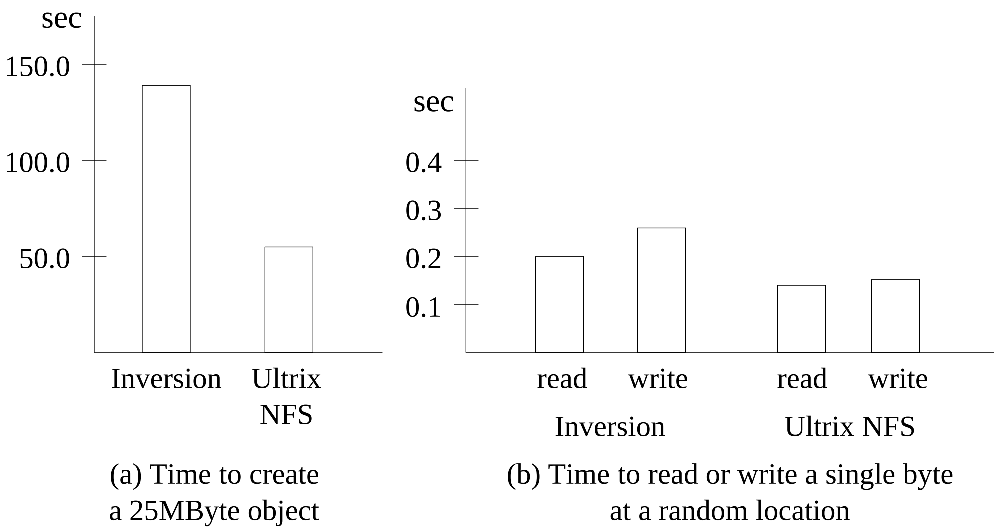

Figure 6: Elapsed time to create 25MByte file, read or write a single byte

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 6b shows the overhead for reading or writing a single byte of 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 7 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

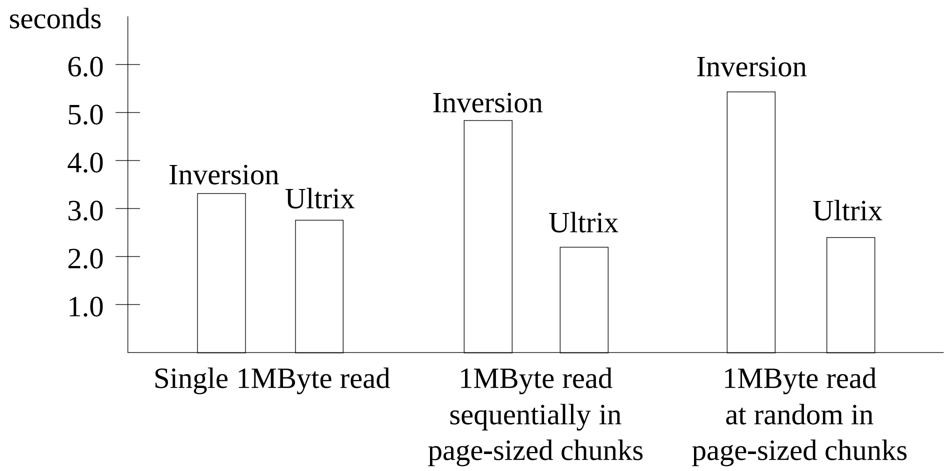

Figure 7: Elapsed read times for Inversion and Ultrix NFS

under the two file systems, there is reason to believe that tuning will improve Inversion.

The final pair of bars in Figure 7 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 8 presents the write performance of Inversion and Ultrix NFS. The tests run were identical to those performed for Figure 7, 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 dramatically 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.

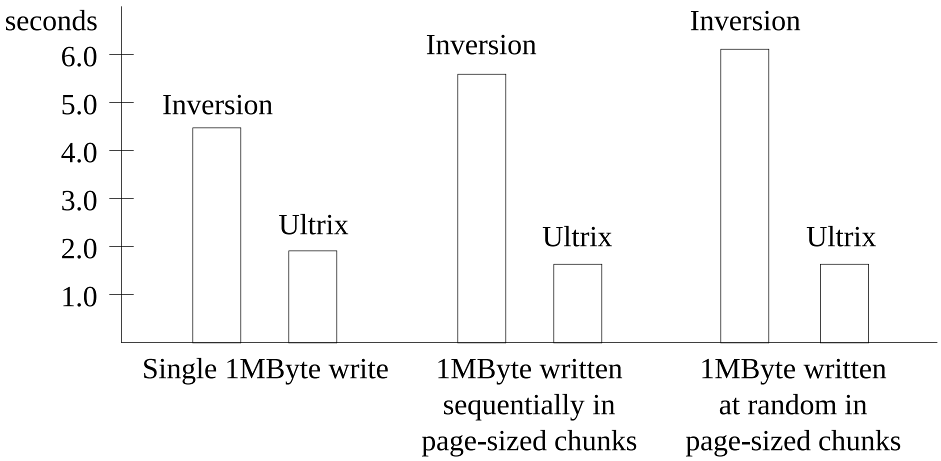

Figure 8: Elapsed write times for Inversion and Ultrix NFS

Figure 8 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.

6.4. Evaluation of Benchmark Results

[STON92] presents results from a benchmark of Inversion run on a symmetric multiprocessor. The results from that study are substantially better than those observed here. On a multiprocessor, Inversion gets between 50 and 95 percent of the throughput of a native file system running on the same hardware.

The most important difference between this study and that conducted for [STON92] is the process structure used to evaluate the file system. In the current benchmark, the client program ran on a remote machine from the file server. In [STON92], both processes ran on the same machine. When the client and server are on the same hardware, more intelligent IPC mechanisms can be used. In particular, buffers shared by the two processes can be marked copy-on-write. This eliminates an extra data copy. Since no traffic crosses the Ethernet, the savings are even more substantial. Finally, the symmetric multiprocessor examined in [STON92] allowed one processor to move blocks between a main memory cache and the file system while other processors were carrying out computations. As a result, the earlier performance study achieved better results than those presented here.

The results presented in [STON92] make it clear that the communication overhead incurred by the current implementation of Inversion is high. In order to determine just how high, another version of the benchmark was run. In this case, the benchmark tests were dynamically loaded into the POSTGRES process being used as the Inversion file server. Since users can dynamically load their routines into the data manager, this is an available option in cases where performance in critical. In this case, the Inversion ‘‘client’’ and ‘‘server’’ run in the same address space on the DECsystem 5900.

In this configuration, data is never copied across address space boundaries, and protocol overhead is eliminated. The performance presented here is the best that a user could achieve with the current implementation of Inversion. Note that the same file system can be used simultaneously by dynamically-loaded code and by the more conventional client/server architecture.

Table 2 shows the performance of the single-process benchmark. 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 2 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

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 2: Elapsed time in seconds for benchmark tests in three configurations

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. Given this optimization, it is reasonable to expect performance within fifty percent of Ultrix NFS and PRESTOserve from Inversion.

7. CONCLUSIONS

The research presented here makes contributions in three areas. These contributions are

- introducing a storage manager switch abstraction into a database system, making it easy to add new device types to a data manager;

- the design and implementation of a device manager for a Sony write-once optical disk jukebox; and
- the design and implementation of a novel filesystem, Inversion, which provides important semantic guarantees.

As tertiary storage devices are more widely deployed, the importance of the storage manager switch abstraction will increase. Current commercial relational databases include hard-coded dependencies on magnetic disk. Experience with POSTGRES indicates that these dependencies can be subtle and hard to identify. Eliminating these dependencies is critical, since tertiary storage has dramatically different characteristics from magnetic disks.

The issues raised in designing the Sony jukebox device manager include cache management policies, allocation strategies, and transfer size optimizations. The Sony jukebox device manager achieves aggressive sharing and supports fast recovery by implementing a persistent cache. The expense of setting up transfers from optical platters is amortized by doing extent-based allocation, and making extent sizes sufficiently large. The cache size, extent size, and transfer size are all configurable.

The Inversion file system provides transaction protection, fine-grained time travel, fast recovery, device independence, strong typing, and support for *ad hoc* queries on file system metadata and data. As shown in the performance evaluation section, Inversion gets between thirty and eighty percent of the throughput of an optimized version of Ultrix NFS, which uses special-purpose hardware to accelerate file system accesses. Tuning should improve Inversion's performance further.

8. FUTURE WORK

The work presented here raises several new research problems. This section lists some of the ways in which the storage manager, the Sony jukebox device manager, the Inversion file system, or POSTGRES as a whole could be improved.

8.1. Better Integration of the Storage Manager into POSTGRES

There are several ways in which the integration of the new storage manager into POSTGRES could be improved. Most important of these is making relations truly location-transparent. Doing so requires hiding the fact that some devices are actually write-once. As mentioned above, this requires writing software to support logical overwrite-in-place on relation data blocks. Overwrite should work on all device managers.

Another area for future work is improving the cost estimation functions for tertiary storage. This may require more extensive changes to the query optimizer, so that device managers can, for example, estimate locality of particular access paths. The current cost estimation function for the Sony jukebox device manager provides only a coarse guide in choosing an optimal execution plan. One way to improve the ability of device managers to correctly estimate plan cost would be to allow them to collect device-specific statistics on the relations that they store.

There are also interesting research issues in designing new access paths to relations on tertiary storage. For example, it may be desirable to invent new join strategies or indexing structures that are optimized for high-latency, low-bandwidth devices. Such strategies would make tertiary storage much more attractive.

Finally, when POSTGRES supports clustering of relation data, some design decisions in (for example) the Sony jukebox device manager will need to be rethought. If blocks at arbitrary locations in relations may be overwritten, then the current strategy of keeping the highest-numbered block on magnetic disk will not work.

8.2. Tuning the Sony Jukebox Device Manager

More experiments should be done on the Sony jukebox device manager, to quantify the costs of cache maintenance, transfer setup, and extent-based allocation. The behavior of the device manager could certainly be tuned with some analysis of its current behavior. In addition, cache replacement policies other than LRU should be investigated.

Another strategy would be to reimplement the caching code completely. The current implementation assumes a cache size on the order of tens of megabytes. If this were scaled to, say, a gigabyte, different cache layout strategies would make sense. For example, extents could be laid out using a hash function, rather than contiguously on disk.

8.3. Work on the Inversion File System

The Inversion file system, similarly, should be profiled and tuned to improve its performance. It may make sense to use other indices, or other indexed access methods altogether, on the blocks and metadata of files stored in Inversion. The protocol used for network file access should be redesigned to speed up transfers.

One issue that should be addressed is the programmatic interface to Inversion files. Since Inversion provides a strict superset of the services provided by conventional UNIX systems, the interface to Inversion will be different from the interface to the native file system. For example, time travel requires the specification of timestamps when opening a file. This interface is not yet clearly defined.

Two other important issues are management of a file system cache and the policy used to lay out file data blocks on storage devices. Unfortunately, both of these require violating the storage manager abstraction. It would be interesting to see if a clean interface could be devised that would continue to support device extensibility, and still allow Inversion to handle layout and caching.

8.4. Addition of New Device Managers

Finally, new device managers should be added to POSTGRES. Hardware that will soon become available to the project includes a Metrum videocassette jukebox, an Exabyte 8500 8mm helical scan jukebox, and a Hewlitt-Packard magneto-optical disk jukebox. New device managers should be written for these.

Currently, a version of POSTGRES exists that optimizes and executes parallel query plans [HONG92]. Parallel POSTGRES uses striping, or horizontal partitioning of relations, in order to achieve data parallelism. A striping device manager for magnetic disk would be easy to write, and would eliminate a good deal of special-case code in the parallel version of the system.

9. ACKNOWLEDGEMENTS

Joey Hellerstein and Wei Hong reviewed early versions of this thesis and suggested many improvements. Margo Seltzer and Mark Sullivan, Sin City's senior citizens, offered useful advice on the design and implementation of the storage manager switch and the Inversion file system. Dr. Randy Katz and Dr. Ray Larson served as members of my thesis committee. Dr. Michael Stonebraker's guidance and encouragement have been invaluable throughout the project. Finally, I want especially to thank my wife Teresa and my son Matthew for their patience and support over the course of an apparently endless academic career.

10. REFERENCES

[BERN88] Bernstein, P., *et al.*, ''Future Directions in DBMS Research'', *The Laguna Beach Report*, Laguna Beach, CA, February 1988.

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

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

[FONG86] Fong, Z., ''The Design and Implementation of the POSTGRES Query Optimizer'', M.Sc. Report, University of California at Berkeley, Berkeley, CA, Aug. 1986.

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

[HONG92] Hong, W., ''Exploiting Inter-Operation Parallelism in XPRS'', *Proc. 1992 ACM-SIGMOD International Conference on Management of Data*, San Diego, CA, June, 1992 (to appear).

[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.

[KIM84] Kim, W., ‘‘Highly Available Systems for Database Applications’’, *ACM Computing Surveys* 16(1), 1984.

[LEFF84] Leffler, S., *et al.*, ‘‘An Advanced 4.3BSD Interprocess Communication Tutorial’’, *UNIX Programmer’s Manual Supplementary Documents* vol. 1, Berkeley, CA, 1984.

[LEFF89] Leffler, S., *et al.*, *The Design and Implementation of the 4.3 BSD Operating System*, Prentice-Hall, 1989.

[MCFA91] McFadden, A., ‘‘C Library Interface for the Sony Optical Disk Changer’’, CS199 Project Report, UC Berkeley, May 1991.

[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.

[PIKE90] Pike, R., *et al.*, ‘‘Plan 9 From Bell Labs’’, *Proc. Summer 1990 UKUUG Conference*, London, England, July 1990.

[POTA91] Potamianos, J., ‘‘Semantics and Performance of Integrated DBMS Rule Systems’’, Ph.D. Dissertation, University of California at Berkeley, Berkeley, CA, January 1992.

[RASH88] Rashid, R., *et al.*, ‘‘Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures’’, *IEEE Transactions on Computers* Vol. 37 No. 8, August 1988.

[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*, 1991.

[SAND85] Sandberg, R., *et al.*, ‘‘Design and Implementation of the Sun Network File System’’, *Proc. Usenix Summer 1985 Technical Conference*, June 1985.

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

[SELI79] Selinger, P., *et al.*, ‘‘Access Path Selection in a Relational Database Management System’’, *Proc. 1979 ACM-SIGMOD Conference on Management of Data*, Boston, MA, June 1979.

[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.

[SKEE81] Skeen, D., ‘‘Nonblocking Commit Protocols’’, *Proc. 1981 ACM SIGMOD Conference*, 1981.

[SONY89a] Sony Corporation, ‘‘Writable Disk Auto Changer WDA-610 Specifications and Operating Instructions’’, 3-751-106-21 (1), Japan, 1989.

[SONY89b] Sony Corporation, ‘‘Writable Disk Drive WDD-600 and Writable Disk WDM-6DL0 Operating Instructions’’, 3-751-047-21 (1), Japan, 1989.

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

[STON90] Sonebraker, M., *et al.*, ‘‘The Implementation of POSTGRES’’, *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[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.

[STON92] Stonebraker, M., and Olson, M., ‘‘Large Object Support in POSTGRES’’, submitted to *Proc. 1992 VLDB Conference*, Vancouver, BC, Canada.

[SULL92] Sullivan, M. and Olson, M., ‘‘An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System’’, *Proc. 8th Annual International Conference on Data Engineering*, Tempe, AZ, February 1992.

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

Extending the POSTGRES Database System to Manage Tertiary Storage

Michael Allen Olson
Department of Electrical Engineering and Computer Science
University of California at Berkeley

Master's Thesis

Version 2.4 (1992/05/12)
FINAL
Printed May 12, 1992

Extending the POSTGRES Database System to Manage Tertiary Storage

by

Michael Allen Olson

Table of Contents