THE DESIGN OF THE POSTGRES STORAGE SYSTEM

Michael Stonebraker

EECS Department
University of California
Berkeley, Ca., 94720

Abstract

This paper presents the design of the storage system for the POSTGRES data base system under construction at Berkeley. It is novel in several ways. First, the storage manager supports transaction management but does so without using a conventional write ahead log (WAL). In fact, there is no code to run at recovery time, and consequently recovery from crashes is essentially instantaneous. Second, the storage manager allows a user to optionally keep the entire past history of data base objects by closely integrating an archival storage system to which historical records are spooled. Lastly, the storage manager is consciously constructed as a collection of asynchronous processes. Hence, a large monolithic body of code is avoided and opportunities for parallelism can be exploited. The paper concludes with a analysis of the storage system which suggests that it is performance competitive with WAL systems in many situations.

1. INTRODUCTION

The POSTGRES storage manager is the collection of modules that provide transaction management and access to data base objects. The design of these modules was guided by three goals which are discussed in turn below. The first goal was to provide transaction management without the necessity of writing a large amount of specialized crash recovery code. Such code is hard to debug, hard to write and must be error free. If it fails on an important client of the data manager, front page news is often the result because the client cannot access his data base and his business will be adversely affected. To achieve this goal, POSTGRES has adopted a novel storage system in which no data is ever overwritten; rather all updates are turned into insertions.

The second goal of the storage manager is to accomodate the historical state of the data base on a write-once-read-many (WORM) optical disk (or other archival medium) in addition to the current state on an ordinary magnetic disk. Consequently, we have designed an asynchronous process, called the **vacuum cleaner** which moves archival records off magnetic disk and onto an archival storage system.

The third goal of the storage system is to take advantage of specialized hardware. In particular, we assume the existence of non-volatile main memory in some reasonable quantity. Such memory can be provide through error correction techniques and a battery-back-up scheme or from some other hardware means. In addition, we expect to have a few low level machine instructions available for specialized uses to be presently explained. We also assume that architectures with several processors will become increasingly popular. In such an environment, there

---

This research was sponsored by the Navy Electronics Systems Command under contract N00039-84-C-0039.

is an opportunity to apply multiple processors to running the DBMS where currently only one is utilized. This requires the POSTGRES DBMS to be changed from the monolithic single-flow-of-control architectures that are prevalent today to one where there are many asynchronous processes concurrently performing DBMS functions. Processors with this flavor include the Sequent Balance System [SEQU85], the FIREFLY, and SPUR [HILL85].

The remainder of this paper is organized as follows. In the next section we present the design of our magnetic disk storage system. Then, in Section 3 we present the structure and concepts behind our archival system. Section 4 continues with some thoughts on efficient indexes for archival storage. Lastly, Section 5 presents a performance comparison between our system and that of a conventional storage system with a write-ahead log (WAL) [GRAY78].

2. THE MAGNETIC DISK SYSTEM

2.1. The Transaction System

Disk records are changed by data base **transactions,** each of which is given a unique **transaction identifier** (XID). XIDs are 40 bit unsigned integers that are sequentially assigned starting at 1. At 100 transactions per second (TPS), POSTGRES has sufficient XIDs for about 320 years of operation. In addition, the remaining 8 bits of a composite 48 bit interaction identifier (IID) is a command identifier (CID) for each command within a transaction. Consequently, a transaction is limited to executing at most 256 commands.

In addition there is a **transaction log** which contains 2 bits per transaction indicating its status as:

committed
aborted
in progress

A transaction is **started** by advancing a counter containing the first unassigned XID and using the current contents as a XID. The coding of the log has a default value for a transaction as ‘‘in progress’’ so no specific change to the log need be made at the start of a transaction. A transaction is **committed** by changing its status in the log from ‘‘in progress’’ to ‘‘committed’’ and placing the appropriate disk block of the log in stable storage. Moreover, any data pages that were changed on behalf of the transaction must also be placed in stable storage. These pages can either be forced to disk or moved to stable main memory if any is available. Similarly, a transaction is aborted by changing its status from ‘‘in progress’’ to ‘‘aborted’’.

The **tail** of the log is that portion of the log from the oldest active transaction up to the present. The **body** of the log is the remainder of the log and transactions in this portion cannot be ‘‘in progress’’ so only 1 bit need be allocated. The body of the log occupies a POSTGRES relation for which a special access method has been built. This access method places the status of 65536 transactions on each POSTGRES 8K disk block. At 1 transaction per second, the body increases in size at a rate of 4 Mbytes per year. Consequently, for light applications, the log for the entire history of operation is not a large object and can fit in a sizeable buffer pool. Under normal circumstances several megabytes of memory will be used for this purpose and the status of all historical transactions can be readily found without requiring a disk read.

In heavier applications where the body of the log will not fit in main memory, POSTGRES applies an optional compression technique. Since most transactions commit, the body of the log contains almost all ‘‘commit’’ bits. Hence, POSTGRES has an optional bloom filter [SEVR76] for the aborted transactions. This tactic compresses the buffer space needed for the log by about a factor of 10. Hence, the bloom filter for heavy applications should be accomodatable in main memory. Again the run-time system need not read a disk block to ascertain the status of any transaction. The details of the bloom filter design are presented in [STON86].

The tail of the log is a small data structure. If the oldest transaction started one day ago, then there are about 86,400 transactions in the tail for each 1 transaction per second processed. At 2 bits per entry, the tail requires 21,600 bytes per transaction per second. Hence, it is reasonable to put the tail of the log in stable main memory since this will save the pages containing the tail of the log from being forced to disk many times in quick succession as transactions with similar transaction identifiers commit.

2.2. Relation Storage

When a relation is created, a file is allocated to hold the records of that relation. Such records have no prescribed maximum length, so the storage manager is prepared to process records which cross disk block boundaries. It does so by allocating continuation records and chaining them together with a linked list. Moreover, the order of writing of the disk blocks of extra long records must be carefully controlled. The details of this support for multiblock records are straightforward, and we do not discuss them further in this paper. Initially, POSTGRES is using conventional files provided by the UNIX operating system; however, we may reassess this decision when the entire system is operational. If space in a file is exhausted, POSTGRES extends the file by some multiple of the 8K page size.

If a user wishes the records in a relation to be approximately clustered on the value of a designated field, he must declare his intention by indicating the appropriate field in the following command

cluster rel-name on {(field-name using operator)}

POSTGRES will attempt to keep the records approximately in sort order on the field name(s) indicated using the specified operator(s) to define the linear ordering. This will allow clustering secondary indexes to be created as in [ASTR76].

Each disk record has a bit mask indicating which fields are non-null, and only these fields are actually stored. In addition, because the magnetic disk storage system is fundamentally a versioning system, each record contains an additional 8 fields:

OID	a system-assigned unique record identifier
Xmin	the transaction identifier of the interaction inserting the record
Tmin	the commit time of Xmin (the time at which the record became valid)
Cmin	the command identifier of the interaction inserting the record
Xmax	the transaction identifier of the interaction deleting the record
Tmax	the commit time of Xmax (the time at which the record stopped being valid)
Cmax	the command identifier of the interaction deleting the record
PTR	a forward pointer

When a record is inserted it is assigned a unique OID, and Xmin and Cmin are set to the identity of the current interaction. the remaining five fields are left blank. When a record is updated, two operations take place. First, Xmax and Cmax are set to the identity of the current interaction in the record being replaced to indicate that it is no longer valid. Second, a new record is inserted into the data base with the proposed replacement values for the data fields. Moreover, OID is set to the OID of the record being replaced, and Xmin and Cmin are set to the identity of the current interaction. When a record is deleted, Xmax and Cmax are set to the identity of the current interaction in the record to be deleted.

When a record is updated, the new version usually differs from the old version in only a few fields. In order to avoid the space cost of a complete new record, the following compression technique has been adopted. The initial record is stored uncompressed and called the **anchor point.** Then, the updated record is differenced against the anchor point and only the actual changes are stored. Moreover, PTR is altered on the anchor point to point to the updated record, which is called a **delta record.** Successive updates generate a one-way linked list of delta records off an initial anchor point. Hopefully most delta record are on the same operating system page as the anchor point since they will typically be small objects.

It is the expectation that POSTGRES would be used as a local data manager in a distributed data base system. Such a distributed system would be expected to maintain multiple copies of all important POSTGRES objects. Recovery from hard crashes, i.e. one for which the disk cannot be read, would occur by switching to some other copy of the object. In a non-distributed system POSTGRES will allow a user to specify that he wishes a second copy of specific objects with the command:

    mirror rel-name

Some operating systems (e.g. VMS [DEC86] and Tandem [BART81]) already support mirrored files, so special DBMS code will not be necessary in these environments. Hopefully, mirrored files will become a standard operating systems service in most environments in the future.

2.3. Time Management

The POSTGRES query language, POSTQUEL allows a user to request the salary of Mike using the following syntax.

    retrieve (EMP.salary) where EMP.name = ‘‘Mike’’

To support access to historical tuples, the query language is extended as follows:

    retrieve (EMP.salary) using EMP[T] where EMP.name = ‘‘Mike’’

The scope of this command is the EMP relation as of a specific time, T, and Mike’s salary will be found as of that time. A variety of formats for T will be allowed, and a conversion routine will be called to convert times to the 32 bit unsigned integers used internally. POSTGRES constructs a query plan to find qualifying records in the normal fashion. However, each accessed tuple must be additionally checked for validity at the time desired in the user’s query. In general, a record is **valid at time T** if the following is true:

```
Tmin < T and Xmin is a committed transaction and either:
  Xmax is not a committed transaction or
  Xmax is null  or
  Tmax > T
```

In fact, to allow a user to read uncommitted records that were written by a different command within his transaction, the actual test for validity is the following more complex condition.

```
Xmin = my-transaction and Cmin != my-command and T = ‘‘now’’
     or
Tmin < T and Xmin is a committed transaction and either:
  (Xmax is not a committed transaction and Xmax != my-transaction) or
  (Xmax = my-transaction and Cmax = my-command) or
   Xmax is null  or
   Tmax > T or
```

If T is not specified, then T = ‘‘now’’ is the default value, and a record is valid at time, ‘‘now’’ if

```
Xmin = my-transaction and Cmin != my-command
     or
Xmin is a committed transaction and either
  (Xmax is not a committed transaction and Xmax != my-transaction) or
  (Xmax = my-transaction and Cmax = my-command) or
  Xmax is null
```

More generally, Mike's salary history over a range of times can be retrieved by:

retrieve (EMP.Tmin, EMP.Tmax, EMP.salary)
using EMP[T1,T2] where EMP.name = ''Mike''

This command will find all salaries for Mike along with their starting and ending times as long as the salary is valid at some point in the interval, [T1, T2]. In general, a record is **valid in the interval [T1,T2]** if:

Xmin = my-transaction and Cmin != my-command and T2 >= ''now''
or
Tmin < T2 and Xmin is a committed transaction and either:
(Xmax is not a committed transaction and Xmax != my-transaction) or
(Xmax = my-transaction and Cmax = my-command) or
Xmax is null or
Tmax > T1

Either T1 or T2 can be omitted and the defaults are respectively T1 = 0 and T2 = +infinity

Special programs (such as debuggers) may want to be able to access uncommitted records. To facilitate such access, we define a second specification for each relation, for example:

retrieve (EMP.salary) using all-EMP[T] where EMP.name = ''Mike''

An EMP record is in all-EMP at time T if

Tmin < T and (Tmax > T or Tmax = null)

Intuitively, all-EMP[T] is the set of all tuples committed, aborted or in-progress at time T.

Each accessed magnetic disk record must have one of the above tests performed. Although each test is potentially CPU and I/O intensive, we are not overly concerned with CPU resources because we do not expect the CPU to be a significant bottleneck in next generation systems. This point is discussed further in Section 5. Moreover, the CPU portion of these tests can be easily committed to custom logic or microcode or even a co-processor if it becomes a bottleneck.

There will be little or no I/O associated with accessing the status of any transaction, since we expect the transaction log (or its associated bloom filter) to be in main memory. We turn in the next subsection to avoiding I/O when evaluating the remainder of the above predicates.

2.4. Concurrency Control and Timestamp Management

It would be natural to assign a timestamp to a transaction at the time it is started and then fill in the timestamp field of each record as it is updated by the transaction. Unfortunately, this would require POSTGRES to process transactions logically in timestamp order to avoid anomolous behavior. This is equivalent to requiring POSTGRES to use a concurrency control scheme based on timestamp ordering (e.g. [BERN80]. Since simulation results have shown the superiority of conventional locking [AGRA85], POSTGRES uses instead a standard two-phase locking policy which is implemented by a conventional main memory lock table.

Therefore, Tmin and Tmax must be set to the commit time of each transaction (which is the time at which updates logically take place) in order to avoid anomolous behavior. Since the commit time of a transaction is not known in advance, Tmin and Tmax cannot be assigned values at the time that a record is written.

We use the following technique to fill in these fields asynchronously. POSTGRES contains a TIME relation in which the commit time of each transaction is stored. Since timestamps are 32 bit unsigned integers, byte positions 4*j through 4*j + 3 are reserved for the commit time of transaction j. At the time a transaction commits, it reads the current clock time and stores it in the appropriate slot of TIME. The tail of the TIME relation can be stored in stable main memory to avoid the I/O that this update would otherwise entail.

Moreover, each relation in a POSTGRES data base is tagged at the time it is created with one of the following three designations:

no archive: This indicates that no historical access to relations is required.

light archive: This indicates that an archive is desired but little access to it is expected.

heavy archive: This indicates that heavy use will be made of the archive.

For relations with ‘‘no archive’’ status, Tmin and Tmax are never filled in, since access to historical tuples is never required. For such relations, only POSTQUEL commands specified for T = ‘‘now’’ can be processed. The validity check for T = ‘‘now’’ requires access only to the POSTGRES LOG relation which should be contained in the buffer pool. Hence, the test consumes no I/O resources.

If ‘‘light archive’’ is specified, then access to historical tuples is allowed. Whenever Tmin or Tmax must be compared to some specific value, the commit time of the appropriate transaction is retrieved from the TIME relation to make the comparison. Access to historical records will be slowed in the ‘‘light archive’’ situation by this requirement to perform an I/O to the TIME relation for each timestamp value required. This overhead will only be tolerable if archival records are accessed a very small number of times in their lifetime (about 2-3).

In the ‘‘heavy archive’’ condition, the run time system must look up the commit time of a transaction as in the ‘‘light archive’’ case. However, it then writes the value found into Tmin or Tmax, thereby turning the read of a historical record into a write. Any subsequent accesses to the record will then be validatable without the extra access to the TIME relation. Hence, the first access to an archive record will be costly in the ‘‘heavy archive’’ case, but subsequent ones will will incur no extra overhead.

In addition, we expect to explore the utility of running another system demon in background to asynchronously fill in timestamps for ‘‘heavy archive’’ relations.

2.5. Record Access

Records can be accessed by a sequential scan of a relation. In this case, pages of the appropriate file are read in a POSTGRES determined order. Each page contains a pointer to the next and the previous logical page; hence POSTGRES can scan a relation by following the forward linked list. The reverse pointers are required because POSTGRES can execute query plans either forward or backward. Additionally, on each page there is a line table as in [STON76] containing pointers to the starting byte of each anchor point record on that page.

Once an anchor point is located, the delta records linked to it can be constructed by following PTR and decompressing the data fields. Although decompression is a CPU intensive task, we feel that CPU resources will not be a bottleneck in future computers as noted earlier. Also, compression and decompression of records is a task easily committed to microcode or a separate co-processor.

An arbitrary number of secondary indexes can be constructed for any base relation. Each index is maintained by an **access method.** and provides keyed access on a field or a collection of fields. Each access method must provide all the procedures for the POSTGRES defined abstraction for access methods. These include get-record-by-key, insert-record, delete-record, etc. The POSTGRES run time system will call the various routines of the appropriate access method when needed during query processing.

Each access method supports efficient access for a collection of operators as noted in [STON86a]. For example, B-trees can provide fast access for any of the operators:

{=, <=, <, >, >=}

Since each access method may be required to work for various data types, the collection of operators that an access methods will use for a specific data type must be **registered** as an **operator class.** Consequently, the syntax for index creation is:

index on rel-name is index-name ({key-i with operator-class-i})
using access-method-name and performance-parameters

The performance-parameters specify the fill-factor to be used when loading the pages of the index, and the minimum and maximum number of pages to allocate. The following example specifies a B-tree index on a combined key consisting of an integer and a floating point number.

index on EMP is EMP-INDEX (age with integer-ops, salary with float-ops)
using B-tree and fill-factor = .8

The run-time system handles secondary indexes in a somewhat unusual way. When a record is inserted, an anchor point is constructed for the record along with index entries for each secondary index. Each index record contains a key(s) plus a pointer to an entry in the line table on the page where the indexed record resides. This line table entry in turn points to the byte-offset of the actual record. This single level of indirection allows anchor points to be moved on a data page without requiring maintenance of secondary indexes.

When an existing record is updated, a delta record is constructed and chained onto the appropriate anchor record. If no indexed field has been modified, then no maintenance of secondary indexes is required. If an indexed field changed, then an entry is added to the appropriate index containing the new key(s) and a pointer to the anchor record. There are no pointers in secondary indexes directly to delta records. Consequently, a delta record can only be accessed by obtaining its corresponding anchor point and chaining forward.

The POSTGRES query optimizer constructs plans which may specify scanning portions of various secondary indexes. The run time code to support this function is relatively conventional except for the fact that each secondary index entry points to an anchor point and a chain of delta records, all of which must be inspected. Valid records that actually match the key in the index are then returned to higher level software.

Use of this technique guarantees that record updates only generate I/O activity in those secondary indexes whose keys change. Since updates to keyed fields are relatively uncommon, this ensures that few insertions must be performed in the secondary indexes.

Some secondary indexes which are hierarchical in nature require disk pages to be placed in stable storage in a particular order (e.g. from leaf to root for page splits in B+-trees). POSTGRES will provide a low level command

order block-1 block-2

to support such required orderings. This command is in addition to the required **pin** and **unpin** commands to the buffer manager.

3. THE ARCHIVAL SYSTEM

3.1. Vacuuming the Disk

An asynchronous demon is responsible for sweeping records which are no longer valid to the archive. This demon, called the **vacuum cleaner,** is given instructions using the following command:

vacuum rel-name after T

Here T is a time relative to ‘‘now’’. For example, the following vacuum command specifies vacuuming records over 30 days old:

vacuum EMP after ‘‘30 days’’

The vacuum cleaner finds candidate records for archiving which satisfy one of the following conditions:

Xmax is non empty and is a committed transaction and ‘‘now’’ - Tmax >= T
Xmax is non empty and is an aborted transaction
Xmin is non empty and is an aborted transaction

In the second and third cases, the vacuum cleaner simply reclaims the space occupied by such records. In the first case, a record must be copied to the archive unless ‘‘no-archive’’ status is set for this relation. Additionally, if ‘‘heavy-archive’’ is specified, Tmin and Tmax must be filled in by the vacuum cleaner during archiving if they have not already been given values during a previous access. Moreover, if an anchor point and several delta records can be swept together, the vacuuming process will be more efficient. Hence, the vacuum cleaner will generally sweep a chain of several records to the archive at one time.

This sweeping must be done very carefully so that no data is irrecoverably lost. First we discuss the format of the archival medium, then we turn to the sweeping algorithm and a discussion of its cost.

3.2. The Archival Medium

The archival storage system is compatible with WORM devices, but is not restricted to such systems. We are building a conventional extent-based file system on the archive, and each relation is allocated to a single file. Space is allocated in large extents and the next one is allocated when the current one is exhausted. The space allocation map for the archive is kept in a magnetic disk relation. Hence, it is possible, albeit very costly, to sequentially scan the historical version of a relation.

Moreover, there are an arbitrary number of secondary indexes for each relation in the archive. Since historical accessing patterns may be different than accessing patterns for current data, we do not restrict the archive indexes to be the same as those for the magnetic disk data base. Hence, archive indexes must be explicitly created using the following extension of the indexing command:

index on {archive} rel-name is index-name ({key-i with operator-class-i})
using access-method-name and performance-parameters

Indexes for archive relations are normally stored on magnetic disk. However, since they may become very large, we will discuss mechanisms in the next section to support archive indexes that are partly on the archive medium.

The anchor point and a collection of delta records are concatenated and written to the archive as a single variable length record. Again secondary index records must be inserted for any indexes defined for the archive relation. An index record is generated for the anchor point for each archive secondary index. Moreover, an index record must be constructed for each delta record in which a secondary key has been changed.

Since the access paths to the portion of a relation on the archive may be different than the access paths to the portion on magnetic disk, the query optimizer must generate two plans for any query that requests historical data. Of course, these plans can be executed in parallel if multiple processors are available. In addition, we are studying the decomposition of each of these two query plans into additional parallel pieces. A report on this subject is in preparation [BHID87].

3.3. The Vacuum Process

Vacuuming is done in three phases, namely:

phase 1: write an archive record and its associated index records
phase 2: write a new anchor point in the current data base
phase 3: reclaim the space occupied by the old anchor point and its delta records

If a crash occurs while the vacuum cleaner is writing the historical record in phase 1, then the data still exists in the magnetic disk data base and will be revacuumed again at some later time. If the historical record has been written but not the associated indexes, then the archive will have a record which is reachable only through a sequential scan. If a crash occurs after some index records have been written, then it will be possible for the same record to be accessed in a magnetic disk relation and in an archive relation. In either case, the duplicate record will consume system resources; however, there are no other adverse consequences because POSTGRES is a relational system and removes duplicate records during processing.

When the record is safely stored on the archive and indexed appropriately, the second phase of vacuuming can occur. This phase entails computing a new anchor point for the magnetic disk relation and adding new index records for it. This anchor point is found by starting at the old anchor point and calculating the value of the last delta that satisfies

‘‘now’’ - Tmax >= T

by moving forward through the linked list. The appropriate values are inserted into the magnetic disk relation, and index records are inserted into all appropriate index. When this phase is complete, the new anchor point record is accessible directly from secondary indexes as well as by chaining forward from the old anchor point. Again, if there is a crash during this phase a record may be accessible twice in some future queries, resulting in additional overhead but no other consequences.

The last phase of the vacuum process is to remove the original anchor point followed by all delta records and then to delete all index records that pointed to this deleted anchor point. If there is a crash during this phase, index records may exist that do not point to a correct data record. Since the run-time system must already check that data records are valid and have the key that the appropriate index record expects them to have, this situation can be checked using the same mechanism.

Whenever there is a failure, the vacuum cleaner is simply restarted after the failure is repaired. It will re-vacuum any record that was in progress at some later time. If the crash occurred during phase 3, the vacuum cleaner could be smart enough to realize that the record was already safely vacuumed. However, the cost of this checking is probably not worthwhile. Consequently, failures will result in a slow accumulation of extra records in the archive. We are depending on crashes to be infrequent enough that this is not a serious concern.

We now turn to the cost of the vacuum cleaner.

3.4. Vacuuming Cost

We examine two different vacuuming situations. In the first case we assume that a record is inserted, updated K times and then deleted. The whole chain of records from insertion to deletion is vacuumed at once. In the second case, we assume that the vacuum is run after K updates, and a new anchor record must be inserted. In both cases, we assume that there are Z secondary indexes for both the archive and magnetic disk relation, that no key changes are made during these K updates, and that an anchor point and all its delta records reside on the same page. Table 1 indicates the vacuum cost for each case. Notice that vacuuming consumes a constant cost. This

	whole chain	K updates
archive-writes	1+Z	1+Z
disk-reads	1	1
disk-writes	1+Z	1+Z

I/O Counts for Vacuuming
Table 1

rather surprising conclusion reflects the fact that a new anchor record can be inserted on the same page from which the old anchor point is being deleted without requiring the page to be forced to stable memory in between the operations. Moreover, the new index records can be inserted on the same page from which the previous entries are deleted without an intervening I/O. Hence, the cost PER RECORD of the vacuum cleaner decreases as the length of the chain, K, increases. As long as an anchor point and several delta records are vacuumed together, the cost should be marginal.

4. INDEXING THE ARCHIVE

4.1. Magnetic Disk Indexes

The archive can be indexed by conventional magnetic disk indexes. For example, one could construct a salary index on the archive which would be helpful in answering queries of the form:

retrieve (EMP.name) using EMP [,] where EMP.salary = 10000

However, to provide fast access for queries which restrict the historical scope of interest, e.g:

retrieve (EMP.name) using EMP [1/1/87,] where EMP.salary = 10000

a standard salary index will not be of much use because the index will return all historical salaries of the correct size whereas the query only requested a small subset. Consequently, in addition to conventional indexes, we expect time-oriented indexes to be especially useful for archive relations. Hence, the two fields, Tmin and Tmax, are stored in the archive as a single field, I, of type **interval.** An R-tree access method [GUTM84] can be constructed to provide an index on this interval field. The operators for which an R-tree can provide fast access include "overlaps" and "contained-in". Hence, if these operators are written for the interval data type, an R-tree can be constructed for the EMP relation as follows:

index on archive EMP is EMP-INDEX (I with interval-ops)
using R-tree and fill-factor = .8

This index can support fast access to the historical state of the EMP relation at any point in time or during a particular period.

To utilize such indexes, the POSTGRES query planner needs to be slightly modified. Note that POSTGRES need only run a query on an archive relation if the scope of the relation includes some historical records, Hence, the query for an archive relation must be of the form:

...using EMP[T]

or

...using EMP[T1,T2]

The planner converts the first construct into:

...where T contained-in EMP.I

and the second into:

...where interval(T1,T2) overlaps EMP.I

Since all records in the archive are guaranteed to be valid, these two qualifications can replace all the low level code that checks for record validity on the magnetic disk described in Section 2.3. With this modification, the query optimizer can use the added qualification to provide a fast access path through an interval index if one exists.

Moreover, we expect combined indexes on the interval field along with some data value to be very attractive, e.g:

```
index on archive EMP is EMP-INDEX
(I with interval-ops, salary with float-ops)
using R-tree and fill-factor = .8
```

Since an R-tree is a multidimensional index, the above index supports intervals which exist in a two dimensional space of time and salaries. A query such as:

```
retrieve (EMP.name) using EMP[T1,T2] where EMP.salary = 10000
```

will be turned into:

```
retrieve (EMP.name) where EMP.salary = 10000
                 and interval(T1,T2) overlaps EMP.I
```

The two clauses of the qualification define another interval in two dimensions and conventional R-tree processing of the interval can be performed to use both qualifications to advantage.

Although data records will be added to the archive at the convenience of the vacuum cleaner, records will be generally inserted in ascending time order. Hence, the poor performance reported in [ROUS85] for R-trees should be averted by the nearly sorted order in which the records will be inserted. Performance tests to ascertain this speculation are planned. We now turn to a discussion of R-tree indexes that are partly on both magnetic and archival mediums.

4.2. Combined Media Indexes

We begin with a small space calculation to illustrate the need for indexes that use both media. Suppose a relation exists with 10**6 tuples and each tuple is modified 30 times during the lifetime of the application. Suppose there are two secondary indexes for both the archive and the disk relation and updates never change the values of key fields. Moreover, suppose vacuuming occurs after the 5th delta record is written, so there are an average of 3 delta records for each anchor point. Assume that anchor points consume 200 bytes, delta records consume 40 bytes, and index keys are 10 bytes long.

With these assumptions, the sizes in bytes of each kind of object are indicated in Table 2. Clearly, 10**6 records will consume 200 mbytes while 3 x 10**6 delta records will require 120 mbytes. Each index record is assumed to require a four byte pointer in addition to the 10 byte key; hence each of the two indexes will take up 14 mbytes. There are 6 anchor point records on the archive for each of the 10**6 records each concatenated with 4 delta records. Hence, archive records will be 360 bytes long, and require 2160 mbytes. Lastly, there is an index record for each of the archive anchor points; hence the archive indexes are 6 times as large as the magnetic disk indexes.

object	mbytes
disk relation anchor points	200
deltas	120
secondary indexes	28
archive	2160
archive indexes	168

Sizes of the Various Objects
Table 2

Two points are evident from Table 2. First, the archive can become rather large. Hence, one should vacuum infrequently to cut down on the number of anchor points that occur in the archive. Moreover, it might be desirable to differentially code the anchor points to save space. The second point to notice is that the archive indexes consume a large amount of space on magnetic disk. if the target relation had three indexes instead of two, the archive indexes would consume a greater amount of space than the magnetic disk relation. Hence, we explore in this section data structures that allow part of the index to migrate to the archive. Although we could alternatively consider index structures that are entirely on the archive, such as those proposed in [VITT85], we believe that combined media structures will substantially outperform structures restricted to the archive. We plan performance comparisons to demonstrate the validity of this hypothesis.

Consider an R-tree storage structure in which each pointer in a non-leaf node of the R-tree is distinguished to be either a magnetic disk page pointer or an archive page pointer. If pointers are 32 bits, then we can use the high-order bit for this purpose thereby allowing the remaining 31 bits to specify 2**31 pages on magnetic disk or archive storage. If pages are 8K bytes, then the maximum size of an archive index is 2**44 bytes (about 1.75 x 10**13 bytes), clearly adequate for almost any application. Moreover, the leaf level pages of the R-tree contain key values and pointers to associated data records. These data pointers can be 48 bytes long, thereby allowing the data file corresponding to a single historical relation to be 2**48 bytes long (about 3.0 x 10**14 bytes), again adequate for most applications.

We assume that the archive may be a write-once-read-many (WORM) device that allows pages to be initially written but then does not allow any overwrites of the page. With this assumption, records can only be dynamically added to pages that reside on magnetic disk. Table 3 suggests two sensible strategies for the placement of new records when they are not entirely contained inside some R-tree index region corresponding to a magnetic disk page.

Moreover, we assume that any page that resides on the archive contains pointers that in turn point only to pages on the archive. This avoids having to contend with updating an archive page which contains a pointer to a magnetic disk page that splits.

Pages in an R-tree can be moved from magnetic disk to the archive as long as they contain only archive page pointers. Once a page moves to the archive, it becomes read only. A page can be moved from the archive to the magnetic disk if its parent page resides on magnetic disk. In this case, the archive page previously inhabited by this page becomes unusable. The utility of this reverse migration seems limited, so we will not consider it further.

We turn now to several page movement policies for migrating pages from magnetic disk to the archive and use the parameters indicated in Table 4 in the discussion to follow. The simplist policy would be to construct a system demon to ‘‘vacuum’’ the index by moving the leaf page to the archive that has the smallest value for Tmax, the left-hand end of its interval. This vacuuming would occur whenever the R-tree structure reached a threshold near its maximum size of F disk pages. A second policy would be to choose a worthy page to archive based both on its value of Tmax and on percentage fullness of the page. In either case, insertions would be made into the R-tree index at the lower left-hand part of the index while the demon would be archiving pages in the lower right hand part of the index. Whenever an intermediate R-tree node had descendents all on the archive, it could in turn be archived by the demon.

For example, if B is 8192 bytes, L is 50 bytes and there is a five year archive of updates at a frequency, U of 1 update per second, then 1.4 x 10**6 index blocks will be required resulting in a four level R-tree. F of these blocks will reside on magnetic disk and the remainder will be on the

P1	allocate to the region which has to be expanded the least
P2	allocate to the region whose maimum time has to be expanded the least

Record Insertion Strategies
Table 3

F	number of magnetic disk blocks usable for the index
U	update frequency of the relation being indexed
L	record size in the index being constructed
B	block size of magnetic disk pages

Parameters Controlling Page Movement
Table 4

archive. Any insertion or search will require at least 4 accesses to one or the other storage medium.

A third movement policy with somewhat different performance characteristics would be to perform ‘‘batch movement’’. In this case one would build a magnetic disk R-tree until its size was F blocks. Then, one would copy the all pages of the R-tree except the root to the archive and allocate a special ‘‘top node’’ on magnetic disk for this root node. Then, one would proceed to fill up a second complete R-tree of F-1 pages. While the second R-tree was being built, both this new R-tree and the one on the archive would be searched during any retrieval request. All inserts would, of course, be directed to the magnetic disk R-tree. When this second R-tree was full, it would be copied to the archive as before and its root node added to the existing top node. The combination might cause the top node to overflow, and a conventional R-tree split would be accomplished. Consequently, the top node would become a conventional R-tree of three nodes. The filling process would start again on a 3rd R-tree of F-3 nodes. When this was full, it would be archived and its root added to the lower left hand page of the 3 node R-tree.

Over time, there would continue to be two R-trees. The first would be completely on magnetic disk and periodically archived. As long as the height of this R-tree at the time it is archived is a constant, H, then the second R-tree of height, H1, will have the bottom H-1 levels on the archive. Moreover, insertions into the magnetic disk portion of this R-tree are always on the left-most page. Hence, the pages along the left-side of the tree are the only ones which will be modified; other pages can be archived if they point entirely to pages on the archive. Hence, some subcollection of the pages on the top H1-H+1 levels remain on the magnetic disk. Insertions go always to the first R-tree while searches go to both R-trees. Of course, there are no deletions to be concerned with.

Again if B is 8192 bytes, L is 50 bytes and F is 6000 blocks, then H will be 3 and each insert will require 3 magnetic disk accesses. Moreover, at 1 update per second, a five year archive will require a four level R-tree whose bottom two levels will be on the archive and a subcollection of the top 2 levels of 100-161 blocks will be on magnetic disk. Hence, searches will require descending two R-trees with a total depth of 7 levels and will be about 40 percent slower than either of the single R-tree structures proposed. On the other hand, the very common operation of insertions will be approximately 25 percent faster.

5. PERFORMANCE COMPARISON

5.1. Assumptions

In order to compare our storage system with a conventional one based on write-ahead logging (WAL), we make the following assumptions.

1) Portions of the buffer pool may reside in non-volatile main memory

2) CPU instructions are not a critical resource, and thereby only I/O operations are counted.

The second assumption requires some explanation. Current CPU technology is driving down the cost of a MIP at a rate of a factor of two every couple of years. Hence, current low-end workstations have a few MIPs of processing power. On the other hand, disk technology is getting denser and cheaper. However, disks are not getting faster at a significant rate. Hence, one can still only expect to read about 30 blocks per second off of a standard disk drive. Current implementations of data base systems require several thousand instructions to fetch a page from the disk followed by 1000-3000 instructions per data record examined on that page. As a simple figure of merit, assume 30000 instructions are required to process a disk block. Hence, a 1 MIP CPU will approximately balance a single disk. Currently, workstations with 3-5 MIPs are available but are unlikely to be configured with 3-5 disks. Moreover, future workstations (such as SPUR and FIREFLY) will have 10-30 MIPs. Clearly, they will not have 10-30 disks unless disk systems shift to large numbers of SCSI oriented single platter disks and away from current SMD disks.

Put differently, a SUN 3/280 costs about $5000 per MIP, while an SMD disk and controller costs about $12,000. Hence, the CPU cost to support a disk is much smaller than the cost of the disk, and the major cost of data base hardware can be expected to be in the disk system. As such, if an installation is found to be CPU bound, then additional CPU resources can be cheaply added until the system becomes balanced.

We analyze three possible situations:

large-SM: an ample amount of stable main memory is available
small-SM: a modest amount of stable main memory is available
no-SM: no stable main memory is availble

In the first case we assume that enough stable main memory is available for POSTGRES and a WAL system to use so that neither system is required to **force** disk pages to secondary storage at the time that they are updated. Hence, each system will execute a certain number of I/O operations that can be buffered in stable memory and written out to disk at some convenient time. We count the number of such **non-forced** I/O operations that each system will execute, assuming all writes cost the same amount. For both systems we assume that records do not cross page boundaries, so each update results in a single page write. Moreover, we assume that each POSTGRES delta record can be put on the same page as its anchor point. Next, we assume that transactions are a single record insertion, update, deletion or an aborted update. Moreover, we assume there are two secondary indexes on the relation affected and that updates fail to alter either key field. Lastly, we assume that a write ahead log will require 3 log records (begin transaction, the data modification, and end transaction), with a total length of 400 bytes. Moreover, secondary index operations are not logged and thereby the log records for 10 transactions will fit on a conventional 4K log page.

In the second situation we assume that a modest amount of stable main memory is available. We assume that the quantity is sufficient to hold only the tail of the POSTGRES log and the tail of the TIME relation. In a WAL system, we assume that stable memory can buffer a conventional log turning each log write into one that need not be synchronously forced out to disk. This situation (small-SM) should be contrasted with the third case where no stable memory at all is available (no-SM). In this latter cases, some writes must be forced to disk by both types of storage systems.

In the results to follow we ignore the cost that either kind of system would incur to mirror the data for high availability. Moreover, we are also ignoring the WAL cost associated with checkpoints. In addition, we assume that a WAL system never requires a disk read to access the appropriate un-do log record. We are also ignoring the cost of vacuuming the disk in the POSTGRES architecture.

5.2. Performance Results

Table 5 indicates the number of I/O operations each of the four types of transactions must execute for the assumed large-SM configuration. Since there is ample stable main memory, neither system must force any data pages to disk and only non-forced I/Os must be done. An insert requires that a data record and two index records be written by either system. Moreover, 1/10th of a log page will be filled by the conventional system, so every 10 transactions there will be another log page which must be eventually written to disk. In POSTGRES the insertions to the LOG relation and the TIME relation generate an I/O every 65536 and 2048 transactions respectively, and we have ignored this small number in Table 5. Consequently, one requires 3 non-forced I/Os in POSTGRES and 3.1 in a conventional system. The next two columns in Table 1 can be similarly computed. The last column summarizes the I/Os for an aborted transaction. In POSTGRES the updated page need not be rewritten to disk. Hence, no I/Os are strictly necessary; however, in all liklihood, this optimization will not be implemented. A WAL system will update the data and construct a log record. Then the log record must be read and the data page returned to its original value. Again, a very clever system could avoid writing the page out to disk, since it is identical to the disk copy. Hence, for both systems we indicate both the optimized number of writes and the non-optimized number. Notice in Table 5 that POSTGRES is marginally better than a WAL system except for deletes where it is dramatically better because it does not delete the 2 index records. We now turn to cases where POSTGRES is less attractive.

Table 6 repeats the I/O counts for the small-SM configuration. The WAL configuration performs exactly as in Table 5 while the the POSTGRES data pages must now be forced to disk since insufficient stable main memory is assumed to hold them. Notice that POSTGRES is still better in the total number of I/O operations; however the requirement to do them synchronously will be a major disadvantage.

	Insert	Update	Delete	Abort
WAL-force	0	0	0	0
WAL-no-force	3.1	1.1	3.1	0.1 or 1.1
POSTGRES-force	0	0	0	0
POSTGRES-non-force	3	1	1	0 or 1

I/O Counts for the Primitive Operations
large-SM Configuration
Table 5

	Insert	Update	Delete	Abort
WAL-force	0	0	0	0
WAL-no-force	3.1	1.1	3.1	0.1 or 1.1
POSTGRES-force	3	1	1	0 or 1
POSTGRES-non-force	0	0	0	0

I/O Counts for the Primitive Operations
small-SM Configuration
Table 6

Table 7 then indicates the I/O counts under the condition that NO stable main memory is available. Here the log record for a conventional WAL system must be forced to disk at commit time. The other writes can remain in the buffer pool and be written at a later time. In POSTGRES the LOG bit must be forced out to disk along with the insert to the TIME relation. Moreover, the data pages must be forced as in Table 6. In this case POSTGRES is marginally poorer in the total number of operations; and again the synchronous nature of these updates will be a significant disadvantage.

In summary, the POSTGRES solution is preferred in the large-SM configuration since all operations require less I/Os. In Table 6 the total number of I/Os is less for POSTGRES; however, synchronous I/O is required. Table 7 shows a situation where POSTGRES is typically more expensive. However, group commits [DEWI84] could be used to effectively convert the results for either type of system into the ones in Table 6. Consequently, POSTGRES should be thought of as fairly competitive with current storage architectures. Moreover, it has a considerable advantage over WAL systems in that recovery time will be instantaneous while requiring a substantial amount of time in a WAL architecture.

6. CONCLUSIONS

This paper has described the storage manager that is being constructed for POSTGRES. The main points guiding the design of the system were:

1) instantaneous recovery from crashes

2) ability to keep archival records on an archival medium

3) housekeeping chores should be done asynchronously

4) concurrency control based on conventional locking

The first point should be contrasted with the standard write-ahead log (WAL) storage managers in widespread use today.

	Insert	Update	Delete	Abort
WAL-force	1	1	1	1
WAL-no-force	3	1	3	0 or 1
POSTGRES-force	5	3	3	1
POSTGRES-non-force	0	0	0	0 or 1

I/O Counts for the Primitive Operations
no-SM Configuration
Table 7

In engineering application one often requires the past history of the data base. Moreover, even in business applications this feature is sometimes needed, and the now famous TP1 benchmark assumes that the application will maintain an archive. It makes more sense for the data manager to do this task internally for applications that require the service.

The third design point has been motivated by the desire to run multiple concurrent processes if there happen to be extra processors. Hence storage management functions can occur in parallel on multiple processors. Alternatively, some functions can be saved for idle time on a single processor. Lastly, it allows POSTGRES code to be a collection of asynchronous processes and not a single large monolithic body of code.

The final design point reflects our intuitive belief, confirmed by simulations, that standard locking is the most desirable concurrency control strategy. Moreover, it should be noted that read-only transactions can be optionally coded to run as of some point in the recent past. Since historical commands set no locks, then read-only transactions will never interfere with transactions performing updates or be required to wait. Consequently, the level of contention in a POSTGRES data base may be a great deal lower than that found in conventional storage managers.

The design of the POSTGRES storage manager has been sketched and a brief analysis of its expected performance relative to a conventional one has been performed. If the analysis is confirmed in practice, then POSTGRES will give similar performance compared to other storage managers while providing the extra service of historical access to the data base. This should prove attractive in some environments.

At the moment, the magnetic disk storage manager is operational, and work is proceeding on the vacuum cleaner and the layout of the archive. POSTGRES is designed to support extendible access methods, and we have implemented the B-tree code and will provide R-trees in the near future. Additional access methods can be constructed by other parties to suit their special needs. When the remaining pieces of the storage manager are complete, we plan a performance ‘‘bakeoff’’ both against conventional storage managers as well as against other storage managers (such as [CARE86, COPE84]) with interesting properties.

REFERENCES

[AGRA85] Agrawal, R. et. al., ‘‘Models for Studying Concurrency Control Performance Alternatives and Implications,’’ Proc. 1985 ACM-SIGMOD Conference on Management of Data, Austin, Tx., May 1985.

[ASTR76] Astrahan, M. et. al., ‘‘System R: A Relational Approach to Data,’’ ACM-TODS, June 1976.

[BART81] Bartlett, J., ‘‘A Non-STOP Kernel,’’ Proc. Eighth Symposium on Operating System Principles,’’ Pacific Grove, Ca., Dec. 1981.

[BERN80] Bernstein, P. at. al., ‘‘Concurrency Control in a System for Distributed Databases (SDD-1),’’ ACM-TODS, March 1980.

[BHID87] Bhide, A., ‘‘Query Processing in Shared Memory Multiprocessor Systems,’’ (in preparation).

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

[COPE84] Copeland, G. and D. Maier, ‘‘Making Smalltalk a Database System,’’ Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.

[DEC86] Digital Equipment Corp., ‘‘VAX/VMS V4.0 Reference Manual,’’ Digital Equipment Corp., Maynard, Mass., June 1986.

[DEWI84] Dewitt, D. et. al., ‘‘Implementation Techniques for Main Memory Database Systems,’’ Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass., June 1984.

[GRAY78] Gray, J., ‘‘Notes on Data Base Operating Systems,’’ IBM Research, San Jose, Ca., RJ1879, June 1978.

[GUTM84] Gutman, A., ‘‘R-trees: A Dynamic Index Structure for Spatial Searching,’’ Proc. 1984 ACM-SIGMOD Conference on Management of Data, Boston, Mass. June 1984.

[HILL86] Hill, M., et al. ‘‘Design Decisions in SPUR,’’ Computer Magazine, vol.19, no.11, November 1986.

[ROUS85] Roussoupoulis, N. and Leifker, D., ‘‘Direct Spatial Search on Pictorial Databases Using Packed R-trees,’’ Proc. 1985 ACM-SIGMOD Conference on Management of Data, Austin, Tx., May 1985.

[SEQU85] Sequent Computer Co., ‘‘The SEQUENT Balance Reference Manual,’’ Sequent Computers, Portland, Ore., 1985.

[SEVR76] Severence, D., and Lohman, G., ‘‘Differential Files: Their Application to the Maintenance of large Databases,’’ ACM-TODS, June 1976.

[STON76] Stonebraker, M., et. al. ‘‘The Design and Implementation of INGRES,’’ ACM-TODS, September 1976.

[STON86] Stonebraker, M. and Rowe, L., ‘‘The Design of POSTGRES,’’ Proc. 1986ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[STON86a] Stonebraker, M., ‘‘Inclusion of New Types in Relational Data Base Systems,’’ Proc. Second International Conference on Data Base Engineering, Los Angeles, Ca., Feb. 1986.

[VITT85] Vitter, J., ‘‘An Efficient I/O Interface for Optical Disks,’’ ACM-TODS, June 1985.