Large Object Support in POSTGRES[1]

Michael Stonebraker
Michael Olson

Department of Electrical Engineering and Computer Science
University of California at Berkeley

Abstract

This paper presents four implementations for support of large objects in POSTGRES. The four implementations offer varying levels of support for security, transactions, compression, and time travel. All are implemented using the POSTGRES abstract data type paradigm, support user-defined operators and functions, and allow file-oriented access to large objects in the database. The support for user-defined storage managers available in POSTGRES is also detailed. The performance of all four large object implementations on two different storage devices is presented.

1. Introduction

There have been numerous implementations supporting large objects in database systems [BILI92]. Typically, these implementations concentrate on low-level issues such as space management and layout of objects on storage media. Support for higher-level services is less uniform among existing systems.

Commercial relational systems normally support **BLOBs** (binary large objects), and provide the capability to store and retrieve them through the query language. Commercial systems, however, in general do not allow the query language to be extended with new functions and operators that manage typed BLOBs. Such query language and typing support would make BLOBs much more useful.

An important issue in managing large objects is the interface provided to users for accessing them. The Starburst long field manager [LEHM89] supports file-oriented access using a functional interface. [HASK82] proposes extending the SQL cursor mechanism to support cursor operations on large object data.

Other important services include versioning and compression. Some research systems, such as Exodus [CARE86], support versioning of large objects, but previous proposals have been not supported other essential features, such as user defined functions, operators, and compression.

Finally, there has been considerable discussion about **extendability** in relational database management systems [BATO86, HAAS90, STON90]. Proposals have addressed the inclusion of new types, new access methods, new optimizer strategies, and rule systems.

In this paper, we present the implementation of large objects in POSTGRES, and describe the services provided by the database system for managing large object data. We consider the notions of files, large objects, file systems, and user-defined storage managers in a new light. Specifically, we propose that large objects be considered **large** abstract data types (ADTs), that large ADTs have a file-oriented rather than cursor-oriented interface, that several different implementations for large ADTs be constructed, and that files be supported on top of data base large ADTs. The constructs we discuss are all implemented in POSTGRES, Version 4, and performance results from the running system will be presented.

The rest of this paper is organized as follows. Section 2 describes related work on the management of large object data. In Section 3 we consider support for large objects as ADTs, and indicate the advantages of this approach as well as three major problems. Then in Section 4 we turn to the user interface desired for large ADTs. Section 5

[1] This research was sponsored by NSF Grant IRI-9107455, ARO Grant DAAL03-91-G-0183, and DARPA Contract DABT63-92-C-0007. Additional support was provided by the University of California and Digital Equipment Corporation under Research Grant #1243.

Other industrial and government partners include the State of California Department of Water Resources, United States Geological Survey, Construction Engineering Research Laboratory (CERL) of the U.S. Army Corps of Engineers, the National Aeronautics and Space Administration (NASA), Epoch Systems, Inc., Hewlett-Packard Corp., Hughes Aircraft Company, MCI, Metrum Corporation, PictureTel Corporation, Research Systems Inc., Science Applications International Corporation, Siemens Inc., and TRW Space and Electronics.

describes the techniques used to manage temporary large objects. We continue in Section 6 with four implementations for POSTGRES large ADTs. Section 7 introduces POSTGRES support for user-defined storage managers and discusses three that we have implemented. Section 8 continues with a discussion of the **Inversion** file system, which supports conventional user files on top of large ADTs. Finally, Section 9 presents a performance study of the various implementations available for files and large objects.

2. Related work

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

The EXODUS storage manager [CARE86] provides a set of low-level abstractions for managing large data objects. It supports efficient versioning of these objects. Users can extend the system to support new object types and operations on them. EXODUS is the storage management component of a DBMS toolkit, and does not provide (for example) query-language access to the data that it stores.

Storage management in Starburst [LEHM89] is more closely integrated with conventional relational database management. Starburst manages large objects as files, with strategies for laying out data efficiently and creating new versions of existing objects. Users can extend the storage manager with new storage strategies, and can define new large data types. Large objects can be accessed via an *ad hoc* query language using a functional interface.

Orion [KIM89] is an object-oriented system that supports efficient versioning of objects. A special buffering strategy for large object data allows Orion to instantiate only pages of interest, so entire objects need not be present in physical memory. #{roman "O"} sub 2# [DEUX90] uses WiSS to manage object data, and so provides page-oriented access to large object data. Both Orion and #{roman "O"} sub 2# provide good support for abstraction and encapsulation over large object data. Users may define methods and accessors on large objects and invoke them from within their programs.

The work described here differs from these systems in several ways. Most importantly, POSTGRES provides a file-oriented interface to large objects, both for the client and for the server programs. The user may choose among several supported implementations, trading off speed against security and durability guarantees. POSTGRES supports fast random access to compressed data, fine-grained time travel over versions of large objects, and the definition of operators and functions on large objects. Operators and functions are dynamically loaded, and may be invoked from the query language. A file system built on top of the large object implementation supports transaction-protected access to conventional file data. Finally, POSTGRES provides this large object support in a relational database system.

3. Large objects as abstract data types

Commercial systems support only rudimentary access to untyped BLOBs, allowing users to read or write them, but not to pass them to functions inside the database system. This approach has some serious drawbacks. First, it requires BLOBs to be moved from the database system to the user's application program in order to examine their contents. Second, it precludes indexing BLOB values, or the results of functions invoked on BLOBs.

A much better alternative is to support an **extensible** collection of **data types** in the DBMS with **user-defined** functions. In this way, the data type ''image'' could be added, requiring a variable and perhaps large amount of space. In addition, functions that operate on the large type could be registered with the database system, and could then be run directly by the data manager. Indexing BLOBs can also be supported.

Clearly, Abstract Data Types (ADTs) offer many advantages over BLOBs. Previous work in this areas is reported in [OSBO86, STON86, STON87]. These proposals support the definition of new types, along with appropriate operators and functions for them. It is straightforward to apply this construct to large objects, and several different storage representations can be quickly identified.

However, there are three major problems with the specific ADT proposal suggested in [STON86, STON87] when it is applied to large objects.

First, when small types are passed as arguments to a function, they reside completely in memory. For large objects, which may be gigabytes in size, this is clearly impractical. Functions using large objects must be able to locate them, and to request small chunks for individual operations.

Second, the specification of ADTs in [STON86, STON87] allows **conversion** functions to be invoked when ADT values move between the database system and the client. These conversion functions transform data from an **external** to an **internal** format.

Conversion routines allow the ADT mechanism to support compression. The input conversion routine compresses the large object, and the output routine uncompresses it. Large objects will include photographs, satellite images, audio streams, video streams, and documents, which will require tailored compression strategies. The ADT proposal supports this capability by allowing an arbitrary number of data types for large objects, and by supporting type-specific conversion routines.

Using conversion routines to manage compression has some problems. Conversion routines only compress or uncompress complete data values, which can be very inefficient if only a small part of a large object must be examined. The second problem arises in a client-server environment. The database server must support access from geographically remote clients over long-haul networks. Using the ADT proposal of [STON86, STON87], data conversion routines will execute on the server, converting the data before it is sent to the client. If these routines are performing large object compression, this is the wrong place to do the conversion. Whenever possible, only compressed large objects should be shipped over the network — the system should support ‘‘just-in-time’’ uncompression. This saves network bandwidth, and will be crucial to good performance in wide-area networks. The original ADT proposal supported compression only on the server side of a client-sever interface, and cannot support this just-in-time requirement.

In Section 6 of this paper, we suggest a collection of proposals with increasing functionality that address these disadvantages. First, however, Sections 4 and 5 discusses the user interface that all our proposals share.

4. Interface to large objects

Others (*e.g.* [HASK82]) have suggested that the cursor mechanism used in SQL be extended to support large objects. We believe that a better model for large object support is a file-oriented interface. A function can be written and debugged using files, and then moved into the database where it can manage large objects without being rewritten. Also, since programmers are already familiar with file access, large objects are easy to use.

Version 4 of POSTGRES provides such a file-oriented interface. In version 4, an application can execute a query to fetch a large object, *e.g.*:

```
retrieve (EMP.picture)
    where EMP.name = "Joe"
```

POSTGRES will return a **large object name** for the ‘‘picture’’ field. The application can then open the large object, seek to any byte location, and read any number of bytes. The application need not buffer the entire object; it can manage only the bytes it actually needs at one time.

To define a large ADT, the normal abstract data type syntax in POSTGRES must be extended to:

```
create large type type-name (
      input = procedure-name-1,
      output = procedure-name-2,
      storage = storage type)
```

Here, the first two fields are from the normal ADT definition and indicate the input and output conversion routines used to convert between external and internal representation. The last field has been added to specify which of the implementations of large ADTs to be described in the next section should be used.

5. Temporary objects

Consider the following query:

```
retrieve (clip(EMP.picture,
               "0,0,20,20"::rect))
    where EMP.name = ‘‘Mike’’
```

Here clip is an ADT function which accepts two arguments, an image and a rectangle, and clips the image to the dimensions of the rectangle. The result of the function is another image which is passed back to the executor, for subsequent forwarding to the user. Functions which return small objects allocate space on the stack for the return value. The stack is not an appropriate place for storage allocation for the return of large objects, and temporary large objects in the data base must be created for this purpose.

As a result, a function returning a large object must create a new large object and then fill in the bytes using a collection of write operations. A pointer to the large object can be returned by the function. Temporary large objects must be garbage-collected in the same way as temporary classes after the query has completed.

6. Large abstract data types

We expect there to be several implementations of large ADTs offering a variety of services at varying performance. This same point about multiple instantiations of an interface is made in a file system context in [MULL91].

In the next four subsections, we discuss four different implementations, namely user file (u-file), POSTGRES file (p-file), records of fixed-length chunks of user data (f-chunk), and variable-length segments of user data (v-segment).

6.1. User file as an ADT

The simplest way to support large ADTs is with user files. With this mechanism, the user could insert a new employee as follows:

```
append EMP (name = "Joe",
            picture = "/usr/Joe")

open ("/usr/Joe")
```

```
write (...)
```

Here, a new record for Joe is added to the EMP class, and the name of a user file is used as a large object designator and stored in the appropriate field in the data base. The user then opens the large object designator and executes a collections of write operations to supply the appropriate bytes.

This implementation has the advantage of being simple, and gives the user complete control over object placement. However, it has several serious drawbacks. Access controls are difficult to manage, since both the user and the database system must be able to read and write the file. If the file system does not support transactions, then the database cannot guarantee transaction semantics for any query using a large object. Finally, this implementation provides no support for automatic management of versions of large objects.

6.2. POSTGRES file as an ADT

The second ADT mechanism for large objects is to utilize a user file, as in the implementation above. However, in this case, the DBMS owns the file in question. An insert is programmed as:

```
retrieve (result = newfilename())
append EMP (name = "Joe",
            picture = result)

open(result)
write ( ... )
```

Here, there is an extra step relative to the previous case. Because POSTGRES is allocating the file in which the bytes are stored, the user must call the function `newfilename` in order to have POSTGRES perform the allocation. After this extra step, the insert proceeds as before.

The only advantage of this implementation over the previous one is that it allows the UNIX file to be updatable by a single user.

6.3. Fixed-length data chunks

In order to support transactions on large objects, POSTGRES breaks them into ‘‘chunks’’ and stores the chunks as records in the database. In the third large object implementation, these chunks are of fixed size, so this implementation is referred to as **f-chunk**.

For each large object, P, a POSTGRES class is constructed of the form:

```
create P (sequence-number = int4,
          data = byte[8000])
```

Here, the user’s large object would be broken into a collection of 8K sub-objects, each with a sequence number. The size of the data array is chosen to ensure a single record neatly fills a POSTGRES 8K page; a small amount of space is reserved for the tuple and page headers.

Since large objects are managed directly by POSTGRES, they are protected. Also, large objects are stored in POSTGRES classes for which transaction support is automatically provided. Lastly, since POSTGRES does not overwrite data, time travel is automatically available.

If a conversion routine is present, each 8K record is passed to the input conversion routine on input, and the variable length string returned by the compression routine is stored in the database. Before a given byte range is examined, the required variable length strings are uncompressed by the output conversion routine. Just-in-time conversion is supported.

The problems with this scheme are that compression is performed on fixed length blocks of size 8K, and that no space savings is achieved unless the compression routine reduces the size of a chunk by one half. Because POSTGRES does not break tuples across pages, the only way that two compressed values will be stored on the same page is if they are half the size of the original block or smaller.

6.4. Variable-length segments

In contrast to the f-chunk proposal, the fourth implementation stores a large object as a collection of variable length segments, or **v-segments**. A segment index is constructed for each large object with the following composition:

```
segment_ndx (locn, compressed_len,
             byte_pointer)
```

The contents of a segment are constructed by compressing the variable length data generated by the user into a variable length object. These variable length objects are concatenated end-to-end and stored as a large ADT, chunked into 8K blocks using the fixed-block storage scheme f-chunk described above. Each time the large object is extended, a new segment is created, and a record is added to the appropriate segment index indicating the location of the segment, its compressed length and a pointer to its position in the compressed f-chunk.

Using this implementation, the unit of compression is a variable length segment, rather than an 8K block. Also, because the segment index obeys the no-overwrite POSTGRES philosophy, time travel is supported for the index. Because the actual segment contents are not overwritten, time travel for the contents of each large ADT is supported. Finally, any reduction in size by the compression routine is reflected in the size of the POSTGRES large object.

7. Storage managers

POSTGRES allows large object data to be stored on any of several different storage devices by supporting **user-defined storage managers**. This construct is supported in Starburst through the use of **extensions** [HAAS90]. Our approach is similar, but defines a specific abstraction that a storage manager must follow. Our abstraction is modelled after the UNIX **file system switch**, and any user can define a new storage manager by writing and registering a small set of interface routines. Typically, a storage manager must be written to manage new device types. A single POSTGRES storage manager can manage all of the magnetic disks available to the system, but a different storage manager was written to manage an optical disk jukebox. A complete description of the device extensibility in POSTGRES appears in [OLSO91].

Version 4 of POSTGRES contains three storage managers. The first supports storage of classes on local magnetic disk, and is a thin veneer on top of the UNIX file system. The second allows relational data to be stored in non-volatile random-access memory. The third supports data on a local or remote optical disk WORM jukebox. When a POSTGRES class is created, it is allocated to any of these storage managers, using a parameter in the **create** command.

8. The Inversion file system

POSTGRES exports a file system interface to conventional application programs. Large objects stored in the database are simultaneously accessible to both database clients, using the query language and database front-end library, and to non-database clients, which treat the large objects as conventional files. Because the file system is supported on top of the DBMS, we have called it the **Inversion** file system.

Inversion stores the directory tree in two database classes:

```
STORAGE  (file-id, large-object)
DIRECTORY (file-name, file-id,
           parent-file-id)
```

The first class maps files to large ADTs in POSTGRES, while the second class stores the directory structure. The standard file system calls (*e.g.* read and write) are supported by turning them into large object reads and writes. Other file system calls are executed by performing the appropriate data base operations on these two classes. A separate class, FILESTAT, stores file access and modification times, the owner's user id, and similar information.

This implementation has several advantages over the conventional UNIX fast file system. First, files are database large ADTs, so security, transactions, time travel and compression are readily available. Second, a DBMS-centric storage manager can be optimized for large object support, so higher performance on very large files is possible. Lastly, because the file system meta-data is stored in DBMS classes, a user can use the query language to perform searches on the DIRECTORY class.

9. Performance

The various large object and file implementations described above provide different levels of security and atomicity, and allow users to trade off performance for reliability. All tests were run on a 12-processor 80386-based Sequent Symmetry computer running Dynix 3.0.17. In this section, we analyze the performance of the following implementations:

(1) user file as a ADT or user file as a file

(2) POSTGRES file as an ADT

(3) f-chunk ADTs or Inversion files using the disk storage manager

(4) v-segment ADTs or Inversion files using the disk storage manager

(5) f-chunk ADTs or Inversion files using the WORM storage manager

(6) v-segment ADTs or Inversion files using the WORM storage manager

9.1. The benchmark

The benchmark measures read and write throughput for large transfers which are either sequential or random. Specifically, a 51.2MB large object was created and then logically considered a group of 12,500 **frames,** each of size 4096 bytes. The following operations constitute the benchmark:

- Read 2,500 frames (10MB) sequentially.
- Replace 2,500 frames sequentially. This operation replaced existing frames in the object with new ones.
- Read 250 frames (1MB) randomly distributed among the 12,500 frames in the object.
- Replace 250 randomly distributed frames throughout the object.
- Read 250 frames from the large object, distributed with 80/20 locality, i.e. the next frame was read sequentially 80% of the time and a new random frame was read 20% of the time.
- Replace 250 frames from the large object, according to the distribution rule above.

In Figure 1, we indicate the size of the 51.2 Mbyte object in the 6 implementations that we tested. User file and POSTGRES file as ADTs show no storage overhead.

User file	51,200,000
POSTGRES file	51,200,000
f-chunk data	51,838,976
f-chunk B-tree index	270,336
f-chunk data (30% compression)	51,838,976
f-chunk B-tree index	270,336
v-segmentdata (50% compression)	36,290,560
v-segment 2-level map	507,904
v-segment B-tree index	188,416
f-chunk data (50% compression)	25,919,488
f-chunk B-tree index	270,336

Storage Used by the Various
Large Object Implementations
Figure 1

This is actually incorrect; the Dynix file system associates inodes and indirect blocks with these files, but the inodes and indirect blocks are owned by the directory containing the file, and not the file itself. As a result they are not included in the space computation. We also measured the f-chunk implementation when no compression was present, and Figure 1 indicates that the storage overhead is 1.8%. Lastly, we tested f-chunk with 30% and 50% compression and v-segment with 30% compression. The f-chunk with 30% compression saves no space because the data values stored by f-chunk are about 5.67K bytes in size. As a result only one data value fits on a POSTGRES page, and the remainder of the space is wasted. On the other hand, the 50% f-chunk and 30% v-segment achieve the desired compression characteristics.

The next two subsections indicate benchmark performance respectively on disk data and on WORM data.

9.2. Disk performance

Figure 2 shows the elapsed time spent in each of the benchmark operations described above. The implementations measured are user file as an ADT, POSTGRES file as an ADT, f-chunk ADTs and v-segment ADTs. Furthermore, f-chunk is measured with 30% and 50% compression as well as with no compression. Lastly, v-segment is measured with 30% compression. Elapsed times are in seconds.

Consider initially the first three columns of Figure 2 where uncompressed data is measured. For sequential accesses, f-chunk is within seven percent of the performance the native file system implementations. Random accesses are more expensive, since the f-chunk implementation maintains a secondary btree index on the data blocks, and so must traverse the index any time a seek is done. Nevertheless, random throughput in f-chunk is half to three-quarters that of the native systems. Although the extra cost of the btree traversal will remain a factor, it is likely that further tuning can improve performance even more.

Now consider the rightmost three columns, in which the effects of compression are shown. We evaluated two compression algorithms; one achieved 30% compression on 4096-byte frames, at an average cost of eight instructions per byte. A second algorithm achieved 50% compression, consuming 20 instructions per byte.

When f-chunk is used with the 30% compression algorithm, no space is saved, because only one compressed user block fits on a POSTGRES page. Elapsed time increases, since an extra eight instructions are executed per byte transferred. The f-chunk implementation with 30% compression is about 13% slower than without compression.

The v-segment implementation does save space, but increases elapsed time even more. This increase is due to extra disk reads that must be done to fetch a user data block. A btree index is scanned to locate the correct entry in the segment-index table. The segment-index record contains a pointer to the compressed segment, which is retrieved. This increases the average number of disk accesses required per data block read. V-segment is about 25% slower than uncompressed f-chunk, but does require less storage.

Finally, using fixed-sized chunks with 50% compression, two user data blocks fit on a single POSTGRES page. In this case, the Inversion file system actually beats the native file system, since fewer I/Os are required to satisfy requests for user blocks. The extra 20 instructions per byte are more than compensated for by the reduced disk traffic. This clearly demonstrates the importance of support for ‘‘chunking’’ compression. No commercially-available file system we know of could begin to approach the performance of the Inversion system on random access to compressed data.

9.3. Optical disk jukebox performance

An earlier section of this paper described the WORM storage manager. In Figure 3 we present the result of our benchmark for this storage system. Because there is no file system for the WORM, we have used in its place a special purpose program which reads and writes the raw device. This program provides an upper bound on how well an operating system WORM jukebox file system could expect to do. Also, this special program cannot update frames, so we have restricted our attention to the read portion of the benchmark.

Operation	*user file*	*POSTGRES file*	*f-chunk 0%*	*f-chunk 30%*	*v-segment 30%*	*f-chunk 50%*
10MB sequential read	13.11	13.77	14.05	16.26	17.35	11.03
10MB sequential write	33.92	34.02	35.31	39.53	41.90	23.48
1MB random read	5.63	5.77	7.36	8.37	9.48	5.52
1MB random write	6.73	6.91	8.81	10.01	11.46	6.76
1MB read, 80/20 locality	2.20	2.23	3.26	3.73	4.12	2.33
1MB write, 80/20 locality	4.25	4.26	5.30	6.06	6.43	3.72

Disk Performance on the Benchmark
Figure 2

Operation	*special program*	*f-chunk 0%*	*f-chunk 30%*	*v-segment 30%*	*f-chunk 50%*
10MB sequential read	123.01	148.92	157.08	100.37	69.87
1MB random read	141.32	142.16	143.51	104.38	67.19
1MB read, 80/20 locality	149.77	42.11	41.21	30.94	16.20

WORM Performance on the Benchmark
Figure 3

For large sequential transfers, the special purpose program outperforms f-chunk by about 20%. This is because it has no overhead for cache management, and makes no guarantees about atomicity or recoverability. For random transfers, however, f-chunk is dramatically superior, because the WORM storage manager in POSTGRES maintains a magnetic disk cache of optical disk blocks. For the 1MB random read test, the cache satisfies some of the block requests. For the 1MB test with locality, most of the requests are satisfied from the cache.

In Figure 3, compression begins to pay off in terms of elapsed times. The 50% f-chunk and 30% v-segment strategies reduce the amount of data that must be moved from the optical disk jukebox into the user program. These transfers are very slow, so eliminating some of them speeds things up substantially.

Overall, the WORM performance numbers are disappointing. Due to a probable bug in the device driver that manages the jukebox, we get only one-quarter the raw throughput claimed by the hardware specification. We are investigating this problem now, and expect to have much better performance soon.

10. Conclusions

In this section we summarize the novel features of our proposal. Others have suggested large object managers, *e.g.* [CARE86, LEHM89]. Our contribution is to propose a range of alternatives with various performance and functionality characteristics. Moreover, two of our proposals support user defined compression, which previous proposals have not considered. Lastly, two of our proposals also have efficient deltaing of versions of large objects to support time travel, which some of the previous proposals have not supported. As was seen in the previous section, our implementations offered varying performance, depending on the underlying device and the nature of the user's operations.

In addition, our architecture supports a standard file system interface to large objects. As such, there is little distinction between files and large objects, and this allows the definition of a novel **Inversion** file system. This system was shown to offer very high functionality at surprisingly good performance. Inversion can use either the f-chunk or v-segment large object implementations for file storage. As our measurements demonstrate, the Inversion approach is within 1/3 of the performance of the native file system. This is especially attractive because time-travel, transactions and compression are automatically available. Another study determined that transaction support alone costs about 15% [SELT92].

Lastly, we suggested a clean table-driven interface to **user-defined storage managers**. This allows the convenient definition of new storage managers, a capability also present in Starburst. However, our proposal has the advantage that any new storage manager automatically supports Inversion files. Consequently, non-data base programs can automatically use the new storage manager.

11. References

[BATO86] Batory, D., ''GENESIS: A Project to Develop an Extendible Database Management System,'' Proc. 1986 International Workshop on Object-oriented Database Systems, Asilomar, Ca., Sept. 1986.

[BILI92] Biliris, A., ''The Performance of Three Database Storage Structures for Managing Large Objects,'' Proc. 1992 ACM SIGMOD Conference, San Diego, CA, June 1992.

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

[DEUX90] Deux, O. *et al.*, ''The Story of O2,'' IEEE Transactions on Knowledge and Data Engineering, March 1990.

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

[HASK82] Haskins, R. and Lorie, R., ''On Extending the Function of a Relational Database System,'' Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, Fl., June 1982.

[KIM89] Kim, W., *et al.*, ''Features of the ORION Object-Oriented Database System,'' *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley/ACM Press, May 1989, pp. 251–282.

[LEHM89] Lehman, T., and Lindsay, B., ''The Starburst Long Field Manager,'' Proc. 1989 VLDB Conference, Amsterdam, Netherlands, Sept. 1989.

[MOSH92] Mosher, C. (ed), ''The POSTGRES Reference Manual, Version 4,'' Electronics Research Laboratory, University of California, Berkeley, Ca., Report M92/14, March 1992.

[MULL91] Muller, K. and Pasquale, J., ''A High-performance Multi-structured File System Design,'' Proceedings of the 1991 ACM Symposium on Operating System Principles, Asilomar, CA, October 1991.

[OLSO91] Olson, M., ''Extending the POSTGRES Database System to Manage Tertiary Storage,'' M.Sc. thesis, University of California, Berkeley, CA, May 1991.

[OSBO86] Osborne, S. and Heaven, T., ''The Design of a Relational System with Abstract Data Types as Domains,'' ACM TODS, Sept. 1986.

[SELT92] Seltzer, M. and Olson, M., ''LIBTP: Portable, Modular Transactions for Unix,'' Proc. 1992 Winter Usenix, San Francisco, CA, Feb. 1992.

[STON80] Stonebraker, M., ''Operating System Support for Database Management,'' CACM, April 1980.

[STON84] Stonebraker, M. and Rowe, L., ''Database Portals: A New Application Program Interface,'' Proc. 1984 VLDB Conference, Singapore, Sept. 1984.

[STON85] Stonebraker, M., *et al.*, ''Problems in Supporting Data Base Transactions in an Operating System Transaction Manager,'' Operating System Review, January, 1985.

[STON86] Stonebraker, M., ''Inclusion of New Types in Relational Data Base Systems,'' Proc. 1986 IEEE Data Engineering Conference, Los Angeles, Ca., Feb. 1986.

[STON87] Stonebraker, M. *et al.*, ''Extensibility in POSTGRES,'' IEEE Database Engineering, Sept. 1987.

[STON87B] Stonebraker, M., ''The POSTGRES Storage System,'' Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

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

[TRAI82] Traiger, I., ''Virtual Memory Management for Data Base Systems,'' Operating Systems Review, Vol 16, No 4, October 1982.