The Design and Implementation of the Database File System

Nick Murphy, Mark Tonkelowitz, Mike Vernal
{nmurphy, mtonk, vernal}@eecs.harvard.edu

November 11, 2001

Abstract

This paper describes the design and implementation of the Database File System (DBFS). DBFS is a block-structured embeddable file system developed on top of the Berkeley DB (BDB) embeddable database. The DBFS interface is largely consistent with the normal POSIX file system interface, allowing programs to access the file system with minimal code changes. For demonstrative and evaluative purposes, we implement a file server running the Sun Network File System (NFS) protocol that uses DBFS as its backing store, and we compare it with a traditional user-level NFS server implementation.

1 Introduction

In 1981, Michael Stonebraker lamented that operating system support for databases was suboptimal [7]. Stonebraker argued that kernel scheduling and allocation policies were often pessimal for common database workloads, and that the file system abstraction was often superfluous. Despite these deficiencies, users continue to run database systems on top of normal operating systems. Though it was predicted that the gap between the operating system and database communities would continue to widen over time, much recent research has focused on merging ideas from both communities. In this paper, we continue attempts to unify the database and file system communities by implementing and evaluating a new file system, the Database File System (DBFS), which we have implemented on top of the Berkeley DB (BDB), an embedded database management system (DBMS).

With the advent of the Internet, the number of databases in the world has grown rapidly. In order to support dynamic, customizable content, most commercial web sites are database-driven, at least in part. While larger web sites can afford to maintain separate web servers and database servers, many smaller web sites cannot afford the costs associated with this differentiation. As such, we find that many machines must be able to concurrently run both web and database applications.

While there is a high degree of idea cross-pollination between the file system and database communities, there have been few attempts at large-scale architectural unification. In 1993, Michael Olson implemented a file system on top of the POSTGRES DBMS called the Inversion File System (IvFS) [4]. Olson claimed that he could achieve 30-80% of the throughput of an ordinary Ultrix NFS backed by a non-volatile RAM cache [4].

This paper describes the design and implementation of the Database File System (DBFS). DBFS uses the Berkeley DB, an embedded database system originally developed at the University of California at Berkeley and currently maintained by Sleepycat Software [5]. While the performance of IvFS is encouraging, we believe that IvFS has a number of shortcomings. Olson's system used the POSTGRES query language to perform file system operations. We believe that the performance of IvFS was hampered by the parsing and interpretation costs associated with POSTQUEL, the POSTGRES query language.

In contrast, BDB neither requires nor supports high-level query languages, allowing us to bypass the overhead imposed by POSTQUEL. Similarly, the only interface to IvFS was via a special library that supported POSIX file system operations. While we, too, implement a user-level library for DBFS, we also implement an NFS server through which the DBFS can be transparently mounted on any system offering NFS support.

A database backing store helps us avoid the most complex code in the file system, allowing us to concentrate on some of the more interesting aspects of file system design. There is no need to maintain a file system buffer cache in the DBFS code, as the BDB buffer cache should perform well enough. File system consistency can be transparently guaranteed through the use of transactions, rather than the painstaking ordering of file system operations. Similarly, we are not required to flush all metadata changes to disk after each operation, as we can rely upon BDB's support for group commit. There is no need to write and test complex file system recovery code, as we can rely upon the recovery functionality of BDB. Most of our development efforts can instead focus on file system design and performance.

Additionally, we believe that the DBFS offers a number of opportunities for novel file system features. By exposing a transparent transaction interface for the file system, we could theoretically offer multi-file transactions. We can maintain indices on frequently accessed metadata, providing richer support for applications such as `find`. Indices could also be generated for entire directories or files, providing better performance for certain types of search queries.

In our implementation, the file system runs outside of the kernel. This is mainly due to the fact that the BDB is a user-level database. While the DBFS would be optimally implemented on top of a database that used the raw disk, our current version of DBFS runs on top of the Berkeley Fast File System (FFS). As such, our implementation has a number of inherent performance penalties. Our goal is to place a reasonable and demonstrable bound on these penalties by comparing the performance of our DBFS/NFS server with a similar NFS server running directly on top of FFS.

We proceed with a brief examination of related work in section 2. In section 3, we describe the general design of DBFS, and in section 4, we discuss the actual implementation of DBFS on top of BDB. We describe our measurement methodology and results in section 5. We suggest some future work in section 6 and we conclude in section 7.

2 Related Work

Stonebraker noted a number of problems that DBMSs had with typical file systems [7], including:

1. Block allocation is not always physically contiguous, which defeats attempts by databases to localize data.
2. The overlapping of the tree structure of most block-oriented file systems and the tree structure of a database index adds unnecessary overhead.
3. Buffer allocation policies of the operating system may be at odds with those of the database. The LRU replacement policy used by most operating systems can often be the worst policy to use for a database where cyclic access is common.

One of the motivations of this paper was the observation that if a file system could be built on top of a database with reasonable performance, the file system could exploit the rich feature set of the database while leaving the database to benefit from raw disk control. We felt that systems that required database functionality but also wanted a traditional file system interface could take advantage of this scheme. Unfortunately, since raw disk access is not currently supported by BDB, this paper does not directly address the issues that Stonebraker raises. Instead, we believe that we can reasonable infer the behavior of such a system by analyzing the DBFS.

In 1992, Margo Seltzer and Michael Olson developed LIBTP, a UNIX library that provides basic transactioning functions without the overhead of a full DBMS [6]. They suggest in their introduction several useful applications of transactions, including instances where multiple files or parts of files need to be updated atomically and instances where applications need to perform concurrent updates to a shared file with some guarantee of logical consistency. They conclude that their implementation successfully introduces transaction protection in a more sophisticated way than simple `fsync` and `flock` calls and in a more efficient and flexible manner than a full DBMS. The authors' work on LIBTP eventually grew into the Berkeley DB used in the DBFS.

In an effort to provide a cleaner interface to an object-oriented database, Gehani *et al* introduced the OdeFS, a file system interface to an object data store [2]. The authors used NFS to allows access to the OdeFS. As a rationale, they argued that NFS can be run as a user-level program that allows for easy multi-user access. Similarly, it requires no changes to existing clients. Rather than using NFS to retrieve files, Gehani's group used NFS to store and retrieve objects. The OdeFS interface was similar to a file system – basic file system tools such as `ls`, `cd`, `cat`, and `rm` could be used to interact with objects in OdeFS. The server differentiates between actual files (*ufiles*) and *object files*, handling each accordingly. Where OdeFS stores files in the underlying file system, we choose to implement a file system entirely on top of the database. Unfortunately, Gehani provided very little performance analysis.

Oracle Corporation, currently the largest database manufacturer in the world, has developed a commercial product called the Internet File System (iFS). iFS is very similar to the DBFS in spirit – it runs as a user-level application on top of Oracle's flagship database product [1]. iFS offers a suite of interfaces that present the contents of the database as a file system, including NFS, AppleTalk, Samba, HTTP, and FTP. iFS has been largely ignored by the research community, and Oracle has resisted attempts by outsiders to quantify its performance.[1] Oracle claims that the performance of iFS is within a factor of 2 of other unspecified network file systems for standard file system operations, though certain database-friendly operations (e.g., text searches) are thousands of times faster. Unfortunately, performance data is not publicly available to use in comparison with our DBFS.

Perhaps the most comparable system to DBFS, however, is the Inversion File System, which is built on top of the POSTGRES DBMS and supports a comprehensive set of Unix file manipulation commands [4]. All interactions with file data occur through POSTQUEL, the POSTGRES query language. Inversion makes use of the uniqueness of the POSTGRES storage system to allow users to "time travel" to the state of a file at any time in its history [8]. It also allows users to tag files as particular types and define functions that operate on them. While Inversion provides some powerful data manipulation features, it requires users to alter and recompile their programs to use the IvFS API. By providing an NFS interface to the DBFS, we can avoid this problem.

Our decision to use NFS as the interface to our DBFS was initially motivated by a David Mazières paper describing a user-level file system toolkit [3]. We originally explored the possibility of implementing DBFS using the Mazières toolkit, but we found that altering BDB to use Mazières's asynchronous I/O routines would be non-trivial. Instead, we intend to port an existing Open Source NFS implementation to use DBFS.

[1]The ZDNet Article exploring iFS was no longer publically-available as of publication time.

3 Design

3.1 Design Goals

As with any meritorious research project, we were faced with a number of incompatible design goals. First and foremost, we needed to demonstrate that a file system could be implemented on top of a database. Our second goal was for adequate performance. It is clear that a file system can be naïvely built on top of a database by storing every file as a Binary Large Object (BLOB) and having a separate table for every directory. This method, however, would clearly offer abysmal performance, and, therefore, we must reconcile simplicity with efficiency.

Many of our design goals were less fixed. We wanted to offer the full functionality of a standard POSIX file system. We also wanted to be able to test our file system using real-world applications without modification. The straightforward solution – writing a new kernel-level file system – was complicated by the difficulty involved in moving BDB into the kernel. A second solution involved writing DBFS as a strict NFS backing store. While this simplifies our task slightly, we found that binding our code to NFS failed to demonstrate that we could natively offer the full functionality of a POSIX system. A third solution involved building a user-level API that allowed application programs to embed DBFS, as in IvFS [4]. Again, this allowed us to support a POSIX interface, but it limited the range of applications we could use with the DBFS. We settled on a hybrid solution – we chose to develop an embeddable file system API and then develop an NFS server on top of that API.

We spent much of the planning phase vacillating between using a query language interface such as SQL and using an embedded database such as BDB. We found that implementation would be easier and more general in SQL. Designing our file system would be reduced to designing a SQL schema and a set of pre-defined queries, plus a translation/abstraction API. This type of system has a number of benefits, including ease-of-implementation and cross-database portability. Unfortunately, we believed that using such a high-level interface to the database would violate our goal of adequate performance. On the other hand, by tightly bundling DBFS with an embedded database, we could bypass the overhead of SQL while optimizing our system for the embedded database. To this end, we chose the Berkeley DB.

3.2 Berkeley DB

We have already referred in some depth to the Berkeley DB. We present some of the key features of BDB here, with an emphasis on those features we will use. An educated reader should feel free to skip to section 3.3.

Michael Olson, coincidentally the author of IvFS, describes the Berkeley DB in [5] as:

> . . . an Open Source embedded database system with a number of key advantages over comparable systems. It is simple to use, supports concurrent access by multiple users, and provides industrial-strength transaction support, including surviving system and disk crashes.

BDB is currently maintained by Sleepycat Software, which supports and licenses three incarnations of the BDB for commercial uses. The DBFS utilizes the most sophisticated version of BDB, the Berkeley DB Transactional Data Store, which supports logging, fine-grained locking, group commit/rollback, and disaster recovery. All of our interactions with BDB occur via the provided C API.

BDB supports three main types of data access methods – a B^+ tree, a Hash access method, a Queue access method and a unique record number interface. We implement our system using the B^+ tree for simplicity, speed, and generality. We make special note of the fact that BDB's B^+ tree implementation is re-balancing, so insertions in sequential key order (e.g., a block number or a file number) will yield a balanced B^+ tree.

3.3 Schema

We designed the schema for DBFS to be simple and relatively general, so that similar implementations could be undertaken on top of similar systems. We took a block-based approach in our file system design. Rather than storing each file as one Binary Large Object (BLOB), we broke each file into fixed-size blocks, and stored each of the blocks separately in the database.

Our system is organized around three main tables: the `blocks` table, the `metadata` table, and the `direntries` table. These tables store all of the metadata and data in the system. Each file in the system has a unique identifier, which we refer to as the *inumber* for historical reasons. Each table consists of a key component and a data component, both of which can point to arbitrary structs.

The `blocks` table is keyed by a unique block identifier, which consists of the file's inumber and the relative block number. The data portion of the `blocks` table is the actual data page. Thus, a query of the form (i, b) will return a pointer to the bth block of file i.

(File, Block)	Data
⋮	⋮
(28, 0)	"`<html>` ..."
(28, 1)	"...`</html>`"
⋮	⋮

Table 1: An example `blocks` table.

The `metadata` table is keyed by a file's inumber. It returns a `c`-style structure with the file metadata, which includes, but is not limited to, such attributes as size, file type, parent inumber, write time, access time, create time, etc.

Finally, the `direntries` table is is keyed by a directory's inumber and contains information related to the directory structure of the file system. The `direntries` table allows multiple data values for the same key value. It returns a set of tuples of the form $(v, \overline{v})$, where v is a child's inumber and $\overline{v}$ is the associated name of v.

File	Size	Type	Parent	...
⋮	⋮	⋮	⋮	...
28	29782	FILE	15	...
53	0	DIR	15	...
87	1089	FILE	53	...
⋮	⋮	⋮	⋮	⋱

Table 2: An example `metadata` table. We use an ellipsis at the end because we cannot represent all of the metadata in table form.

More intuitively, the `direntries` table represents the set E of all directed edges in the file system tree. That is, if we suppose u is a directory, then $Out(u) = \{v : (u, v) \in E\}$ is the set of all nodes coming from u. For $\forall u$, `direntries` $= \{(u, \{(v, \overline{v}) : v \in Out(u)\})\}$. Thus, by querying on a directory's inumber, we are returned a set of records each of which represents a link out of the directory.

There may be other global state which becomes necessary based upon our implementation, but we save implementation details for section 4.

Parent	Child	Child Name
⋮	⋮	⋮
0	2	"local"
0	3	"bin"
0	4	"etc"
⋮	⋮	⋮
2	8	"bin"
⋮	⋮	⋮

Table 3: An example `direntries` table.

4 Implementation

Our implementation of the high-level DBFS design presented in section 3 is fairly straightforward, so we will not dwell upon it. Instead, we will discuss some of the implementation questions we have been faced with up until this point, and we will discuss how we have addressed them.

As previously discussed, we were torn between implementing an NFS server, a user-level file system library, or some combination of both. A POSIX-compliant file system requires a high-degree of state keeping, while NFS is stateless. We eventually decided porting NFS to a stateful file system would be easier and less error-prone then writing a new NFS server on top of a stateless file system.

We currently have support for a number of pseudo-NFS commands, such as `getattr`, `setattr`, `lookup`, `read`, `write`, `create`, `mkdir`, `remove`, `rmdir`, `rename`, `readdir`. Noticeably absent pseudo-NFS commands currently include `readlink`, `symlink`, `link`, and `fsstat`.

We encountered a number of interesting problems in our implementation attempts. Where possible, we implemented a naïve solution and have designed new, more efficient and/or robust solutions. For instance, it is fairly common that an application must "walk" a directory – that is, iterate over the entries one by one. Generally each iteration in this case requires a return to the user program. We are using what are known as BDB cursors in our `direntries` table to be able to walk all of the entries in a particular directory. BDB cursors have no seek ability – they can only iterate through a set of values. As such, the naïve solution for `readdir` is that a slot number i is passed into the function, the cursor walks forward and returns the ith record. For an n-sized directory, we see that the number of entries we must walk is $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$. Instead, our solution will be to instantiate a cursor on every `opendir` and destroy it on the following `close`.

Another problem we have encountered is devising a method for estimating the amount of space used by the file system and the amount of space remaining. Unfortunately, we cannot give completely accurate measurements because of the fluid nature of data and metadata in DBFS. Instead, this will probably involve deriving a formula that estimates the amount of space remaining a volume.

Similarly, we have not finalized an allocation scheme for inumbers yet. Each inumber is a 64-bit value, so it's theoretically possible to use each inumber once without repetition, as it would take decades to exhaust the inumber space. This allocation policy seems particularly hideous and short-sighted, though. We decided to implement the "naïve" solution in the interim – we chose a new inumber uniformly at random from the inumber space and checked to see if that inumber existed. As it turns out, this method is not entirely unreasonable. Specifically, if we assume a page size of 8 kilobytes and a file system with 8 terabytes, less than 0.0000000001% of the valid inumbers will be used. Thus, we will be virtually assured of choosing an unused inode on each iteration. This has the added nice property of providing a balanced distribution for the B$^+$ tree indices.

5 Measurements

We are currently evaluating the performance of DBFS on a 1 GHz Pentium III running FreeBSD 4.4. Our test machine is equipped with 256 MB of RAM and three 10,000 RPM 18 GB SCSI drives. One drive is configured as both the system and swap drive. A second drive is used solely for storing BDB databases. The third drive is used only for storing the write-ahead logs that BDB generates.

All tests are being conducted locally because of the preliminary nature of this research. As such, we will only explore the results of some early micro-benchmarks on our system. We will then proceed to describe the experimental methodology we will employ on our completed system.

5.1 Read/Write Benchmarks

We were interested in evaluating the performance of the first DBFS implementation before performing any performance modifications. Our results were surprising and somewhat encouraging.

We ran both a read and a write microbenchmark with similar structure. Both benchmarks tested access for file sizes ranging from a single byte to 16 Megabytes. We ran each test 51 times, the first time to warm the cache, and the next fifty times to collect data. We ran each benchmark using both normal system calls and the DBFS interface.

Our first benchmark involved writing files of various sizes. Our results can be seen in figure 5.1. We found that our unoptimized write performance was with a factor of 2 of FFS with SoftUpdates. There are a number of clear factors which contribute to the overheard seen. FFS with SoftUpdates does not require any synchronous writes to disc, while our current configuration of BDB requires synchronous write-ahead logging. Similarly, there are a number of functions that must be called in user-land before a write actually makes it into the kernel, while the FFS write call goes directly into the kernel in our test program.

There is clearly some unavoidable overhead due to the user-level implementation of DBFS and the fact that BDB is running on top of FFS. Interestingly, a DBFS write currently costs a maximum of 5 BDB operations / page. We believe that we can decrease that number by doing some DBFS-level caching and and restructuring the DBFS schema slightly. Thus, once optimized, we should clearly be well-within a factor of 2 performance of FFS with SoftUpdates. We consider that performance encouraging.

Our second benchmark involved reading files of various sizes from the file system. Our results can be seen in 5.1. We found that the read performance of our unoptimized system was within a factor of 4 of FFS. Many of the factors that hurt our write performance also hurt our read performance – BDB on top of FFS, user-level DBFS calls, etc. There is more to the read benchmark than meets the eye, however.

First, we note that we perform 4 BDB operations / page worst case. We believe we can eliminate at least one of those operations with a schema and code reorganization, speeding our read performance up greatly.

We believe our real performance hit stems from the efficient file system cache of FFS. We made no attempt to perform any kind of caching in the DBFS layer itself, and we have not yet customized the performance of BDB's buffer cache. More experiments should lend further insight into our read performance problem.

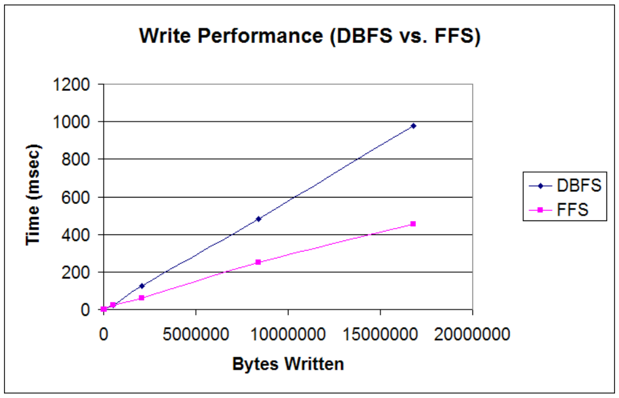

Figure 1: A comparison of the write performance of DBFS vs. FFS with SoftUpdates as a function of the amount of data written.

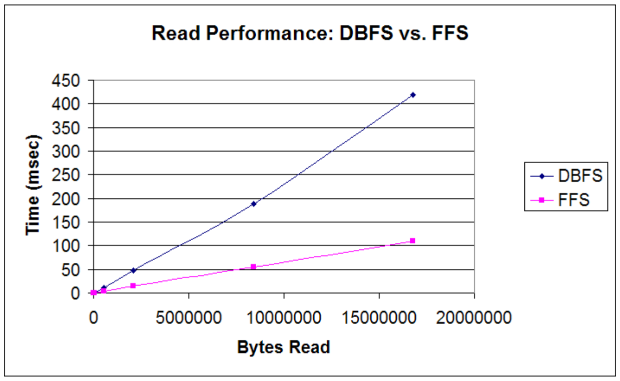

Figure 2: A comparison of the read performance of DBFS vs. FFS with SoftUpdates as a function of the amount of data read.

5.2 Future Methodology

In our first implementation, we stressed correctness over performance, and our system reflects those priorities. Once we have a functional DBFS, we will be primarily interested in its feasibility vis-a-vis a traditional file system. In order to establish its feasibility, we will need to demonstrate that its performance is within some low constant factor of a control file system.

We have already designed and run some simple micro-benchmark utilities, but we will re-adapt these benchmarks and create new ones to measure the performance of specific file system operations. We will test the performance of the system under read-heavy workloads with many small files and with a few large files. We will vary the size of the files being read and the level of concurrency and examine how our system responds to these workloads. We will repeat this methodology with similarly-spirited write-heavy tests.

In order to test how the differing systems handle concurrent access, we may try a set of file-locking read/write workloads. By varying the concurrency, we'd like to see how the DBFS performs under stress.

We're very interested in how the DBFS performs under a metadata-heavy workload, like rapid file creation. By varying the number of files and the size of files created, we imagine that the DBFS will pull quickly ahead of its competition.

After implementing the DBFS, we will have a ripe source of trace data – the write-ahead logs that the DBFS will store by default. It should be possible to run certain real-world applications using the DBFS, store the trace results, and then re-run the derived simulations on both DBFS and another file system and trace the results. We may pursue this experiment if the complexity of parsing the write-ahead log is not prohibitive. Workloads could include web traffic, compilation, and any number of other canonical applications.

Benchmarking simulated and real-world workloads will allow us to evaluate the performance of our system, but will not necessarily help us understand the performance. To truly understand our system, we will want to profile our system and see where and when our code executes. Using the benchmarked workloads as a starting point, we will use a profiling utility such as `gprof` to look at execution times to understand why our system performs the way it does. We would hope to use this data to tune our code and re-run these experiments.

In addition to standard file system benchmarks and measurements, we would like to measure some of the theoretical DBFS operations we posed in this paper, such as metadata-based and text-based searches. We would like to categorize the performance improvement and the impact of moving that support into the file system (e.g., is maintaining indices in the database more efficient in terms of time and space than just running `grep`).

6 Future Work

The design and implementation of DBFS v1 is not yet complete, so it is somewhat premature to speculate about future versions of DBFS. For DBFS v1, the most pressing future work involves the completion of the embeddable DBFS API and the implementation of an NFS server using DBFS.

For the future, there are a number of possible extensions to this work. We hope to eventually be able to port the DBFS to an embeddable database that supports raw disk access for increased performance. In this manner, we should finally be able to natively compare DBFS and FFS. A second possible extension involves the implementation of DBFS on top of a SQL bottom-end. By using SQL, we may be able to facilitate true cross-platform compatibility. Lastly, as previously described, we would like to expose the indexing and transaction back-end of DBFS for application developers to exploit.

7 Conclusion

While the idea of building a file system on top of a database has been mentioned in the literature, it has not necessarily been well-motivated or explored to fruition. Our goal in implementing DBFS is to better understand the nature of database-based file systems.

Our initial implementation shows promising results. While `read()` bandwidth seems sub-optimal, `write()` bandwidth is within a factor of 2 of FFS. With a better understanding of the problem and our implementation, we hope to achieve performance with a factor of 2 of FFS on most workloads.

With a better understanding of what DBMS functionality is exportable through the file system interface, we hope to be able to motivate certain further areas of research and engineering. For instance, if we can exploit the underlying transaction support or replication abilities of the database, we believe we can add functionality to the file system that would have previously been prohibitively complex.

Time-permitting, we would hope to profile DBFS and an analogous NFS server, perhaps using a utility such as `gprof`, and compare and contrast the results. Using this information, we would like to categorize what kind of file system workloads would perform well on the DBFS.

8 Acknowledgments

We stress that this is currently a work-in-progress. We ask that any suggestions be sent to the authors at the addresses listed on the title page.

References

[1] Oracle Corporation. Oracle internet file system: Frequently asked questions, 2001.

[2] N. H. Gehani, H. V. Jagadish, and W. D. Roome. OdeFS: A File System Interface to an Object-Oriented Database. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 249–260, 1994.

[3] David Mazieres. A toolkit for user-level file systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2001.

[4] Michael A. Olson. The design and implementation of the inversion file system. In *Proceedings of the Winter 1993 USENIX Conference*, Berkeley, CA, 1993. USENIX Association.

[5] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track (FREENIX-99)*, pages 183–192, Berkeley, CA, June 6–11 1999. USENIX Association.

[6] Margo Seltzer and Michael Olson. LIBTP: Portable, modular transactions for UNIX. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference: January 20 — January 24, 1992, San Francisco, California*, pages 9–26, Berkeley, CA, USA, Winter 1992. USENIX.

[7] Michael Stonebraker. Operating system support for database management. In *Communications of the ACM, v.24, no. 7*, pages 412–418. Association for Computing Machinery, 1981.

[8] Michael Stonebraker. The design of the POSTGRES storage system. In *Proceedings of the 13th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Brighton UK*, 1987.