# TABLEFS: Embedding a NoSQL Database Inside the Local File System

Kai Ren, Garth Gibson

CMU-PDL-12-103
May 2012

**Parallel Data Laboratory**
Carnegie Mellon University
Pittsburgh, PA 15213-3890

## Abstract

*Conventional file systems are optimzed for large file transfers instead of workloads that are dominated by metadata and small file accesses. This paper examines using techniques adopated from NoSQL databases to manage file system metadata and small files, which feature high rate of changes and efficient out-of-core data representation. A FUSE file system prototype was built by storing file system metadata and small files into a modern key-value store LevelDB. We demonstrate that such techniques can improve the performance of modern local file systems in Linux as much as an order of magnitude for workloads dominated by metadata and tiny files.*

**Acknowledgements:** This research is supported in part by The Gordon and Betty Moore Foundation, NSF under award, SCI-0430781 and CCF-1019104, Qatar National Research Foundation 09-1116-1-172, DOE/Los Alamos National Laboratory, under contract number DE-AC52-06NA25396/161465-1, by Intel as part of the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by gifts from Yahoo!, APC, EMC, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, IBM, Intel, Microsoft, NEC, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, and VMware. We thank the member companies of the PDL Consortium for their interest, insights, feedback, and support.

**Keywords:** TableFS, File System, File System Metadata, NoSQL Database, LSM Tree

## Introduction

While parallel and Internet service file systems have demonstrated effective scaling for high bandwidth, large file transfers in the last decade [26, 33, 17, 25, 8, 10], the same is not true for workloads that are dominated by metadata and tiny file access [23, 34]. Instead there has emerged a large class of scalable small-data storage systems, commonly called key-value stores, that emphasize simple (NoSQL) interfaces and large in-memory caches [2, 21, 16].

Some of these key-value stores feature high rates of change and efficient out-of-memory log-structured merge (LSM) tree structures [22, 5, 29]. We assert that file systems should adopt techniques from modern key-value stores for metadata and tiny files, because these systems are "thin" enough to provide the performance levels required by file systems. We are not attempting to improve semantics (e.g. transactions [27, 12]). To motivate our assertion, in this paper we present experiments in the most mature and restrictive of environments: a local file system managing one magnetic hard disk. Our results show that for workloads dominated by metadata and tiny files, it is possible to improve the performance of the most modern local file systems in Linux by as much as an order of magnitude by adding an interposed file system layer [1] that represents metadata and tiny files in a LevelDB key-value store [15] that stores its LSM tree and write-ahead log segments in these same local file systems. Perhaps it is finally time to accept the old refrain that file systems should at their core use more database management representations and techniques [30], now that database management techniques have been sufficiently decoupled from monolithic database management system (DBMS) bundles [31].

## 1 Background

Even in the era of big data, most things in a file system are small [19, 6]. Inevitably, scalable systems should expect the numbers of small files to soon achieve and exceed billions, a known problem for both the largest [23] and most local file systems [34].

**Embedded Databases** File system metadata is structured data, a natural fit for relational database techniques. However, because of large size, complexity and slow speed, file system developers have long been reluctant to incorporate traditional databases into the lower levels of file systems [30, 20]. Modern stacked file systems often expand on the limited structure in file systems, hiding structures inside directories meant to represent files [9, 13, 4], although this may increase the number of small files in the file system. In this paper, we return to the basic premise: file system metadata is natural for table-based representation, and show that today's lightweight data stores may be up to the task. We are concerned with an efficient representation of huge numbers of small files, not strengthening transactional semantics [27, 12].

**Local File System Techniques** Early file systems stored directory entries in a linear array in a file and inodes in simple on-disk tables. Modern file systems such as Ext4 uses hash tables, and XFS, ZFS, and Btrfs use B-Trees, for indexing directories [14, 18, 32]. Moreover, LFS, WAFL, ZFS and Btrfs [24, 11, 3] use non-overwrite or log structured methods to batch metadata changes and write them sequentially. Such techniques may group all the metadata needed to access a file together on-disk to exploit temporal locality. C-FFS [7], however, explicitly groups the inodes of files with their directory entries, and small files from the same directory in adjacent data blocks. And hFS [35] uses log structuring to manage metadata and update-in-place to manage large files.

**LevelDB and LSM Trees** LevelDB [15] is an open-source key-value database library that features Log-Structured Merge (LSM) Trees [22]. It provides simple APIs such as GET, PUT, DELETE and SCAN.

Unlike BigTable, single row transactions are not supported in LevelDB. Because TABLEFS uses LevelDB, we will review its design in greater detail in the next section.

The basic technique used by LSM-Trees and LevelDB is to manage multiple large sorted arrays of on-disk data (called SSTables) in a log-structured way. When inserting or updating, elements are write-back buffered in memory, and then sorted and written to disk as an SSTable. When the memory buffer grows and exceeds a threshold (4MB by default), the buffer is dumped into disk in SSTables. When querying an element, it requires searching for the element in a list of SSTables and returning the most up-to-date. To reduce the number of SSTables it searches, LevelDB maintains a memory index that records the key range of each SSTable, and uses bloom-filters to reduce false positive lookups. To improve read query speed and remove deleted data, it periodically merge-sorts a list of SSTables. This process is called "compaction", and is similar to *online defragmentation* in file systems, and cleaning in log-structured file system [24].

## 2 TABLEFS

As shown in Figure 1(a), TABLEFS exploits the FUSE user level file system infrastructure to interpose on top of the local file system, represent directories, inodes and small files in one all encompassing table, and only write to the local disk large objects such as write-ahead logs, SSTables containing changes to the metadata table, and files whose size is large.

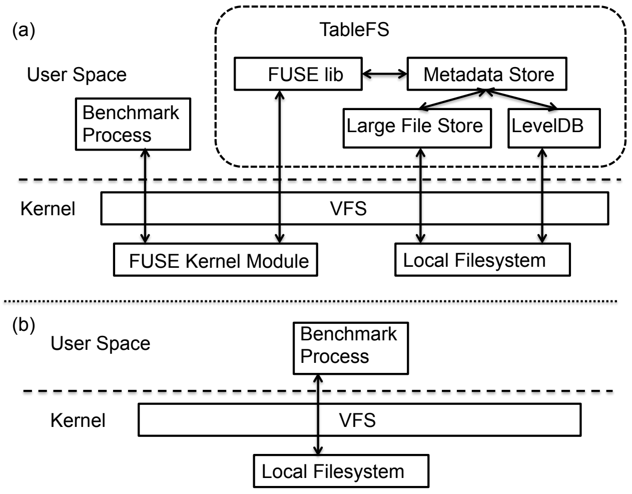
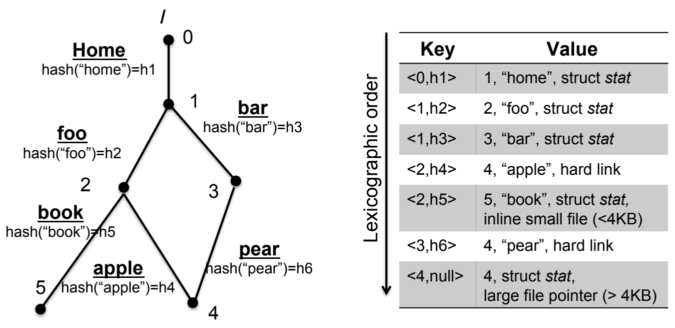


Figure 1: (a) shows the architecture of TABLEFS. A FUSE kernel module redirects file system calls from a benchmark process to TABLEFS, and TABLEFS stores objects into either LevelDB and a large file store. (b) shows the case architecture an experiment compare against in Section 3. These figures suggest the large overhead TABLEFS experiences relative to the traditional local file systems.

**Local File System as Object Store** There is no explicit space management in TABLEFS, instead it uses the local file systems for allocation and storage of objects. Because TABLEFS packs directories, inodes and small files into a LevelDB table, and LevelDB stores sorted logs of about 2MB each, the local file system sees many fewer, larger objects.

**Large File "Blob" Store** Files larger than $T$ bytes are stored directly in the object store according to their inode number. Today's object store uses a two-level directory tree in the local file system, storing a file with inode number $I$ as "/LargeFileStore/$J$/$I$" where $J = I \div 10000$. In TABLEFS today, $T$, the threshold for blobbing a file is 4KB, which is the median size of files in desktop workloads [19], although others have suggested it be 256KB to 1MB [28].

**Table Schema** TableFS's metadata store aggregates directory entries, inode attributes and small files into one LevelDB table. Each file is given an inode number, to represent the hierarchical structure of the user's namespace, each row in the table is ordered by a 128-bit key consisting of the 64-bit inode number of its parent directory and a 64-bit hashing value of its filename string (final component of its pathname). The value of a row contains the full name and inode attributes such as inode number, ownership, access mode, file size, timestamps (*struct stat* in Linux). For small files, the row value also contains the file's data. Figure 2 shows an example of storing a sample file system's metadata into one LevelDB table.



| Key | Value |
|---|---|
| <0,h1> | 1, "home", struct *stat* |
| <1,h2> | 2, "foo", struct *stat* |
| <1,h3> | 3, "bar", struct *stat* |
| <2,h4> | 4, "apple", hard link |
| <2,h5> | 5, "book", struct *stat*, inline small file (<4KB) |
| <3,h6> | 4, "pear", hard link |
| <4,null> | 4, struct *stat*, large file pointer (> 4KB) |

Figure 2: An example illustrates table schema used by TableFS's metadata store. The file with inode number 4 has two hard links, one called "apple" from directory *foo* and the other called "orange" from directory *bar*.

All the entries in the same directory share the same first 64 bits of the table's key. For *readdir* operations, once the inode number of the target directory has been retrieved, a scan sequentially lists all entries having the directory's inode number as the first 64 bits of the table's key. To resolve a single pathname, TableFS starts searching from the root inode, which has a well-known inode number (0). Traversing the user's directory tree involves constructing a search key by concatenating the inode number of current directory with the hash of next component name in the pathname.

**Hard Link** The above schema results special case hard links because two ore more rows must have the same inode attributes and data. As shown in Figure 2, TableFS marks some rows as hard links, and maintain only one row, whose key is its inode number with no name hash value, for the attributes and data.

**Inode Number Allocation** TableFS uses a global counter for allocating inode numbers. The counter increments when creating a new file or a new directory. Since we use 64-bit inode numbers, it is not soon necessary to recycle the inode number of delete entries. Coping with for 32-bit Linux operating systems requiring inode number recycling, is beyond the scope of this paper.

**Locking** LevelDB provides atomic row writes but does not support atomic row read-modify-write operations. Since all inode attributes are stored in one key-value pair, TableFS must often read-modify-write attributes. We implemented a light-weight locking mechanism in the TableFS core layer to ensure correctness under concurrent accesses.

**Journaling** TABLEFS relies on LevelDB and the local file system to achieve journaling. LevelDB has its own write-ahead log that journals all updates to the table. LevelDB can be set to commit the log to disk synchronously or asynchronously. To achieve a consistency guarantee similar to "ordered mode" in Ext4, TABLEFS forces LevelDB to commit the write-ahead log to disk synchronously every 5 seconds.

## 3 Evaluation

**Evaluation System** We evaluate our TABLEFS prototype a Linux desktop equipped with:

| | |
|---|---|
| Linux | Ubuntu 10.04, Kernel 2.6.32-33 |
| CPU | Intel Core2 Quad Q9550 @ 2.83GHz |
| DRAM | DDR SDRAM 4GB, using only 512 MB |
| Hard Disk | Seagate ST31000340NS |
| | SATA, 7200rpm, 1TB |
| | Using only a 5GB partition |
| | Random Seeks 145 seeks/sec peak |
| | Sequential Reads 121.6 MB/sec peak |
| | Sequential Writes 106.4 MB/sec peak |

We limit the machine's available memory to only 512 MB (setting boot parameters of Linux), to prohibit any cache in a user process or kernel cache from using much more memory than it competition, because we cannot easily control all cache sizes.

We compare TABLEFS with Linux's most sophisticated local file systems: Ext4, XFS, and Btrfs, whose versions are 1.41.11, 3.1.0, and 0.19 respectively. Ext4 is mounted with "ordered" journaling to force all data to be flushed out to disk before its metadata is committed to the journal. We believe this is the same fault semantics we achieve in TABLEFS. By default, the journal of Ext4 is synchronously committed to disks every 5 seconds. XFS and Btrfs uses similar policies to synchronously update journals. Btrfs, by default, duplicates metadata and also calculates checksums for data and metadata. We disable both features (unavailable in the other file systems) when benchmarking Btrfs. TABLEFS always uses Btrfs as the underlying file system. Since the tested filesystems have different inode sizes (Ext4 and XFS use 256 bytes and Btrfs uses 136 bytes), we pessimistically punish TABLEFS by padding its inode attributes to 256 bytes. This slows down TABLEFS quite a bit, but it still performs quite well.

All benchmarks are simple "create and query" micro-benchmarks intended only to show that even with the overhead of FUSE, LevelDB, LevelDB compaction, and padded inode structures, TABLEFS can improve performance on the local file system. All benchmarks were run of three times with very little variation.

**Benchmark with Metadata Only** We first micro-benchmark the efficiency of pure metadata operations. The micro-benchmark consists of two phases. The first phase ("creation") generates a file system of one million files, all zero length. This file system has the same namespace as one author's personal Ubuntu desktop, trimmed back to one million files. The benchmark creates this test namespace in the tested file systems in depth first order. The second phase ("query") issues 2 million random read or write queries to random (uniform) files or directories. A read query calls *stat* on the file, a write query randomly does either a *chmod* or *utime* to update the mode or the timestamp fields. Between the two phases, we force local filesystems to drop their cache, so that the second phases starts with a cold cache.

Figure 3 shows the performance in operations per second, averaged over the query phase, for three different ratios of read and write queries: (1) read-only queries, (2) 50% read and 50% write queries, and (3) write-only queries. TABLEFS *is 1.5X to almost 10X faster than the other tested file systems in all three workloads.* Figure 4 shows the total disk traffic (sectors and requests) during the query phase in the 50%

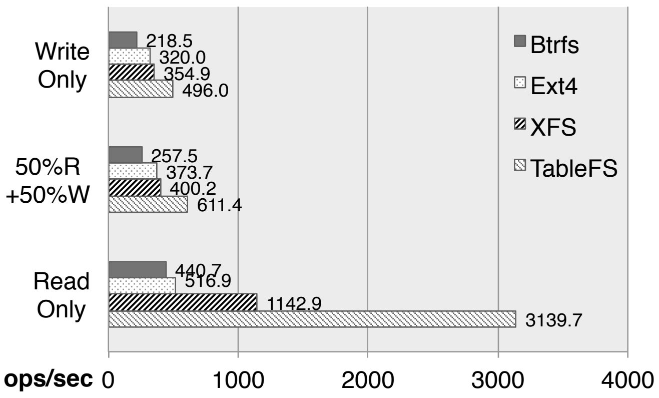
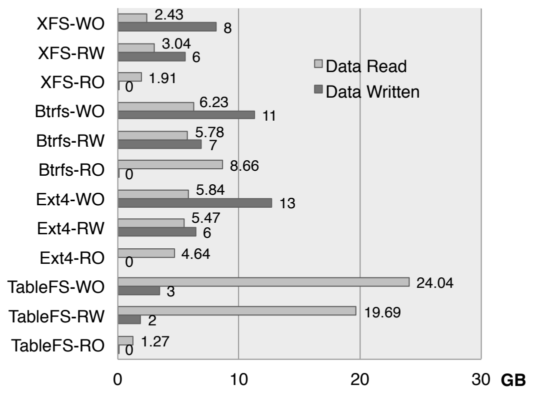
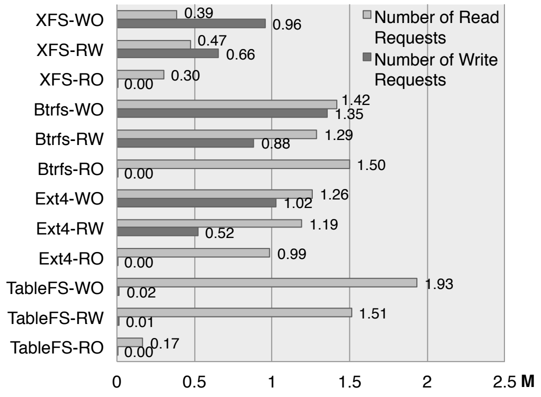


Figure 3: Performance of each file system in the query phase of the metadata-only benchmark.

read and 50% write workload. These numbers are extracted from Linux proc file system (*/proc/diskstats*). Compared to other file systems, TABLEFS reduces write disk traffic a lot, and has only about 10127 disk write requests. This shows that using LevelDB effectively batches small random writes into large sequential writes. TABLEFS, however, incurs more read traffic, and more total number of requests which still running faster.



(a) Total data read from / written to disk



(b) Total number of disk read / write requests

Figure 4: Total disk traffic during the query phase of metadata-only benchmark, when 50% of queries are reads and 50% are writes.

**Benchmark with Small Files** The second micro-benchmark is similar to first except that we create one million 1KB files in 1000 directories, each directory containing 1000 files. In the query phase, read queries retrieves the content of a file, and write queries overwrite the whole file. Files in the query phase are still randomly picked, and distributed uniformly in the namespace. Figure 5 shows the results with a 50%read-50%write workload of one million queries. In creation phase, TABLEFS is much slower than Ext4 and Btrfs, probably because the FUSE overhead is more significant with non-zero file size. In the query phase, however, TABLEFS outperforms all other file systems by 2X.

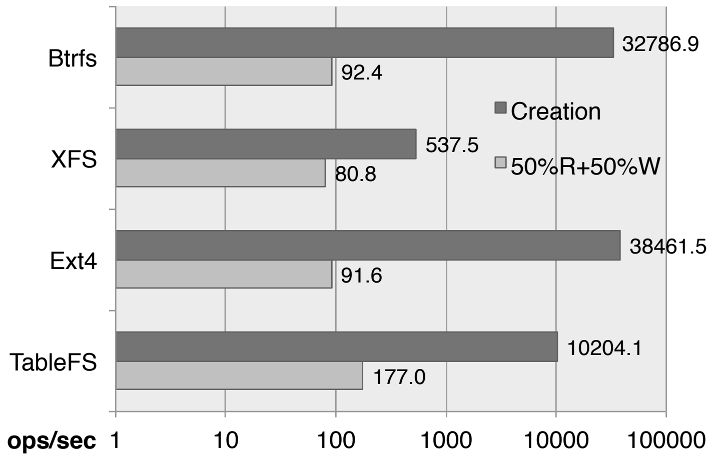


Figure 5: Performance during the query phase of a small files benchmark. The workload is 1 million 50%Read-50%Write queries on 1 million 1KB files.

## 4 Conclusion

File systems have long suffered low performance when accessing huge collections of small files because caches cannot hide all disk seeks. TableFS uses modern key-value store techniques to pack small things (directory entries, inode attributes, small file data) into large on-disk files with the goal of suffering fewer seeks when seeks are unavoidable. Our implementation, even hampered by FUSE overhead, LevelDB code overhead, LevelDB compaction overhead, and pessimistically padded inode attributes, performs much better than state-of-the-art local file systems when the workload is pure metadata and much better during the query phase for small file workloads.

## References

[1] FUSE. http://fuse.sourceforge.net/.

[2] Memcached. http://memcached.org/.

[3] ZFS. http://www.opensolaris.org/os/community/zfs.

[4] John Bent and et al. Plfs: a checkpoint filesystem for parallel applications. In *SC*, 2009.

[5] Fay Chang and et al. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.

[6] Shobhit Dayal. Characterizing HEC storage systems at rest. Technical report, Carnegie Mellon University, 2008.

[7] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX ATC*, 1997.

[8] Sanjay Ghemawat and et al. The Google file system. In *SOSP*, 2003.

[9] Tyler Harter and et al. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *SOSP*, 2011.

[10] HDFS. Hadoop file system. http://hadoop.apache.org/.

[11] Dave Hitz and et al. File system design for an NFS file server appliance. In *USENIX Winter*, 1994.

[12] Aditya Kashyap and et al. File system extensibility and reliability using an in-kernel database. *Master Thesis, Computer Science Department, Stony Brook University*, 2004.

[13] Hyojun Kim and et al. Revisiting storage for smartphones. In *FAST*, 2012.

[14] Jan Kra. Ext4, btrfs, and the others. In *Proceeding of Linux-Kongress and OpenSolaris Developer Conference*, 2009.

[15] LevelDB. A fast and lightweight key/value database library. `http://code.google.com/p/leveldb/`.

[16] Hyeontaek Lim and et al. SILT: a memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[17] Lustre. Lustre file system. `http://www.lustre.org/`.

[18] Avantika Mathur and et al. The new ext4 lesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.

[19] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *FAST*, 2011.

[20] Michael A. Olson. The design and implementation of the inversion file system. In *USENIX Winter*, 1993.

[21] Diego Ongaro and et al. Fast crash recovery in ramcloud. In *SOSP*, 2011.

[22] Patrick ONeil and et al. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 1996.

[23] Swapnil Patil and Garth A. Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *FAST*, 2011.

[24] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, 1991.

[25] Robert B. Ross and et al. PVFS: a parallel file system. In *SC*, 2006.

[26] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[27] Russell Sears and Eric A. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[28] Russell Sears and et al. To blob or not to blob: Large object storage in a database or a filesystem? *Microsoft Technique Report*, 2007.

[29] Jan Stender and et al. BabuDB: Fast and efficient file system metadata storage. In *SNAPI '10*, 2010.

[30] Michael Stonebraker. Operating system support for database management. *Commun. ACM*, 1981.

[31] Michael Stonebraker and Ugur Çetintemel. "one size fits all": An idea whose time has come and gone. In *ICDE*, 2005.

[32] Adam Sweeney. Scalability in the xfs file system. In *USENIX ATC*, 1996.

[33] Brent Welch and et al. Scalable performance of the panasas parallel file system. In *FAST*, 2008.

[34] Ric Wheeler. One billions files: pushing scalability limits of linux filesystem. In *Linux Foudation Events*, 2010.

[35] Zhihui Zhang and et al. hFS: A hybrid file system prototype for improving small file and metadata performance. In *EuroSys*, 2007.