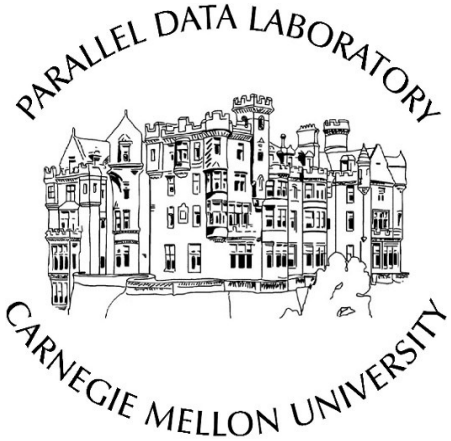


TABLEFS: Enhancing Metadata Efficiency in the Local File System

Kai Ren, Garth Gibson

CMU-PDL-12-110
September 2012

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

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

Abstract

File systems that manage magnetic disks have long recognized the importance of sequential allocation and large transfer sizes for file data. Fast random access has dominated metadata lookup data structures with increasingly use of B-trees on-disk. For updates, on-disk data structures are increasingly non-overwrite, copy-on-write, log-like and deferred. Yet our experiments with workloads dominated by metadata and small file access indicate that even sophisticated local disk file systems like Ext4, XFS and BTRFS leaves a lot of opportunity for performance improvement in workloads dominated by metadata and small files.

In this paper we present a simple stacked file system, TableFS, which uses another local file system as an object store and organizes all metadata into a single sparse table backed on-disk using a Log-Structured Merge (LSM) tree, LevelDB in our experiments. By stacking, TableFS asks only for efficient large file allocation and access from the local file system. By using an LSM tree, TableFS ensures metadata is written to disk in large, non-overwrite, sorted and indexed logs, and inherits a compaction algorithm. Even an inefficient FUSE based user level implementation of TableFS can perform comparably to Ext4, XFS and BTRFS on simple data-intensive benchmarks, and can outperform them by 50% to as much as 1000% for a metadata-intensive query/update workload on data-free files. Such promising performance results from TableFS suggest that local disk file systems can be significantly improved by much more aggressive aggregation and batching of metadata updates.

1 Introduction

In the last decade parallel and internet service file systems have demonstrated effective scaling for high bandwidth, large file transfers [12, 16, 25, 38, 39, 49]. The same, however, is not true of workloads that are dominated by metadata and tiny file access [34, 50]. Instead there has emerged a class of scalable small-data storage systems, commonly called key-value stores, that emphasize simple (NoSQL) interfaces and large in-memory caches [2, 23, 32].

Some of these key-value stores feature high rates of change and efficient out-of-memory log-structured merge (LSM) tree structures [7, 33, 45]. We assert that file systems should adopt techniques from modern key-value stores for metadata and tiny files, because these systems aggressively aggregate metadata and are "thin" enough to provide the performance levels required by file systems. We are not attempting to improve semantics (e.g. provide applications with transactions [40, 51]).

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. Our demonstration is more compelling because it begins disadvantaged: we use an interposed file system layer [1, 52] that represents metadata and tiny files in a LevelDB key-value store [22] 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 [46], now that database management techniques have been sufficiently decoupled from monolithic database management system (DBMS) bundles [47].

2 Background

Even in the era of big data, most things in a file system are small [9, 28]. Inevitably, scalable systems should expect the numbers of small files to soon achieve and exceed billions, a known challenge for both the largest [34] and most local file systems [50]. In this section we review implementation details of the systems employed in our experiments: Ext4, XFS, BTRFS and LevelDB.

2.1 Local File System Structures

Ext4[26] is the fourth generation of Linux ext file systems, and, of the three we study, the most like traditional UNIX file systems. Ext4 divides the disk into block groups, similar to traditional UNIX's cylinder groups, and stores in each block group a copy of the superblock, a block group descriptor, a bitmap describing free data blocks, a table of inodes and bitmap describing free inodes, in addition to the actual data blocks. Inodes contain a file's attributes (such as the file's inode number, ownership, access mode, file size, timestamps) and four extent pointers for data extents or a tree of data extents. The inode of a directory contains links to a HTree hash tree that can be one or two levels deep, based on a 32 bit hash of the directory entry's name. By default only changes to metadata are journaled for durability, and Ext4 asynchronously commits its journal to disk every five seconds. When committing pending data and metadata, data blocks are written to disk before the associated metadata is written to disk.

XFS[48], originally developed by SGI, aggressively and pervasively uses B+ trees to manage all on disk file structures: free space maps, file extent maps, directory entry indices and dynamically allocated inodes. Because all file sizes, disk addresses and inode numbers are 64 bits in XFS, index structures can be large. To reduce the size of these structures XFS partitions the disk into allocation groups, clusters allocation in an allocation group and uses allocation group relative pointers. Free extents are represented in two B+ trees: one indexed by the starting address of the extent and the other indexed by the length of the extent,

to enable efficient search for an appropriately sized extent. Inodes contain either a direct extent map, or a B+ tree of extent maps. Each allocation group has B+ tree indexed by inode number. Inodes for directories have a B+ tree for its directory entries, indexed by a 32 bit hash of the entry's file name. XFS also journals metadata for durability, committing the journal asynchronously when a log buffer (256 KB by default) fills or on synchronous request.

BTRFS[21, 36] is the newest and most sophisticated local file system in our comparison set. Inspired by Rodeh's copy-on-write B-tree[35], as well as features of XFS, NetApp's WAFL and Sun's ZFS[3, 17], BTRFS copies any B-tree node to an unallocated location when it is modified. Provided the modified nodes can be allocated contiguously, B-tree update writing can be highly sequential; although perhaps more data must be written than is minimally needed (write amplification). The other significant feature of BTRFS is its collocation of different metadata components in the same B-tree, called the FS tree. The FS tree is indexed by (inode number, type, offset) and it contains inodes, directory entries and file extent maps, according to the type field: INODE_ITEM for inodes, DIR_ITEM and DIR_INDEX for directory entries, and EXTENT_DATA_REF for file extent maps. Directory entries are stored twice so that they can be ordered differently: in one the offset field of the FS tree index (for the directory's inode) is the hash of the entry's name, for fast single entry lookup, and in the other the offset field is the child file's inode number, allow a range scan of the FS tree to list the inodes of child files and accelerate user operations such as $ls + stat$. BTRFS, by default, delays writes for 30 seconds to increase disk efficiency, and metadata and data is in the same delay queue.

2.2 LevelDB and its Log-Structured Merge Tree

Inspired by a simpler structure in BigTable[7], LevelDB [22] is an open-source key-value storage library that features an Log-Structured Merge (LSM) Tree [33] for on-disk storage. It provides simple APIs such as GET, PUT, DELETE and SCAN. Unlike BigTable, not even single row transactions are supported in LevelDB. Because TABLEFS uses LevelDB, we will review its design in greater detail in this section.

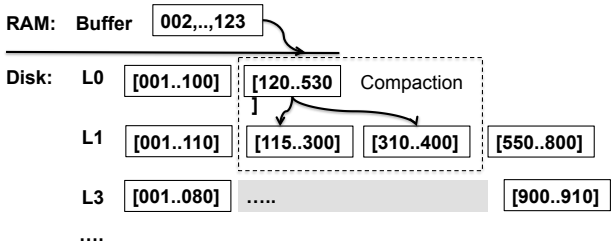


Figure 1: LevelDB represents data on disk in multiple SSTables that store sorted key-value pairs. SSTables are grouped into different levels with lower levels generally containing more recently inserted key-value pairs. Finding a specific pair on disk may search up to all SSTables in level 0 and at most one in each higher level. Compaction is the process of combining SSTables by merge sort into higher levels.

In a simple understanding of an LSM tree, an in memory buffer cache delays writing new and changed entries until it has a significant amount of changes to record on disk. Delaying writes is made more durable by redundantly recording new and changed entries in a write-ahead log, which is pushed to disk periodically and asynchronously by default.

In LevelDB, by default, a set of changes are spilled to disk when the total size of modified entries exceeds 4 MB. When a spill is triggered, called a minor compaction, the changed entries are sorted, indexed and written to disk in a format called an SSTable[7]. These entries may then be discarded by the in memory buffer and can be reloaded by searching each SSTable on disk, possibly stopping when the first match occurs if the SSTables are searched from most recent to oldest. The number of SSTables that need to be searched

can be reduced by maintaining a Bloom filter[6] on each, but with increasing numbers of records the cost of finding a record not in memory increases. Major compaction, or simply "compaction", is the process of combining multiple SSTables into a smaller number of SSTables by merge sort. Compaction is similar to *online defragmentation* in traditional file systems and *cleaning* in log-structured file systems [37].

As illustrated in Figure 1, LevelDB extends this simple approach to further reduce read costs by dividing SSTables into sets, or levels. The 0th level of SSTables follows the simple formulation; each SSTable in this level may contain entries with any key/value, based on what was in memory at the time of its spill. The higher levels of LevelDB's SSTables are the results of compacting SSTables from their own or lower levels. In these higher levels, LevelDB maintains the following invariant: the key range spanning each SSTable is disjoint from the key range of all other SSTables at that level. So querying for an entry in the higher levels only need read at most one SSTable in each level. LevelDB also sizes each of the higher levels differentially: all SSTables have the same maximum size and the sum of the sizes of all SSTables at level L will not exceed 10^L MB. This ensures that the number of level, that is, the maximum number of SSTables that need to be searched in the higher levels, grows logarithmically with increasing numbers of entries.

When LevelDB decides to compact an SSTable at level i, it picks one, finds all other SSTables at the same level and the next higher level that have an overlapping key range, and then sort merges all of these SSTables, producing a set of SSTables with disjoint ranges at the next higher level. If an SSTable at level 0 is selected, it is not unlikely that all other SSTables at level 0 will also be compacted, and many SSTables at level 1 may be included. But at higher levels most compactions will involve a smaller number of SSTables. To select when and what to compact there is a weight associated with compacting each SSTable, and the number of SSTables at level 0 is held in check (by default compaction will be triggered if there are more than four SSTables at level 0). There are also counts associated with SSTables that are searched when looking for an entry, and hotter SSTables will be compacted sooner.

3 TABLEFS

As shown in Figure 2(a), TABLEFS exploits the FUSE user level file system infrastructure to interpose on top of the local file system. TABLEFS represents directories, inodes and small files in one all encompassing table, and only writes to the local disk large objects such as write-ahead logs, SSTables, and files whose size is large.

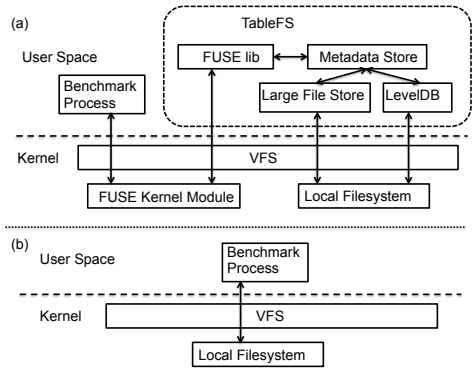


Figure 2: (a) The architecture of TABLEFS. A FUSE kernel module redirects file system calls from a benchmark process to TABLEFS, and TABLEFS stores objects into either LevelDB or a large file store. (b) When we benchmark a local file system, there is no FUSE overhead to be paid.

3.1 Local File System as Object Store

There is no explicit space management in TABLEFS, instead it uses the local file system for allocation and storage of objects. Because TABLEFS packs directories, inodes and small files into a LevelDB table, and LevelDB stores sorted logs (SSTables) of about 2MB each, the local file system sees many fewer, larger objects. We use Ext4 as the object store for TABLEFS in all experiments.

Files larger than T bytes are stored directly in the object store according to their inode number. The 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 $J = I \div 10000$. This is to circumvent any scalability limits on directory entries in the underlying local file systems. In TABLEFS today, T, the threshold for blobbing a file is 4KB, which is the median size of files in desktop workloads [28], although others have suggested T be 256KB as large as 1MB [41].

3.2 Table Schema

TABLEFS's metadata store aggregates directory entries, inode attributes and small files into one LevelDB table with a row for each file. To link together the hierarchical structure of the user's namespace, the rows of the table are ordered by a 128-bit key consisting of the 64-bit inode number of a file's parent directory and a 64-bit hash value of its filename string (final component of its pathname). The value of a row contains the file's full name and inode attributes, such as inode number, ownership, access mode, file size, timestamps (*struct stat* in Linux). For small files, the file's row also contains the file's data.

Figure 3 shows an example of storing a sample file system's metadata into one LevelDB table.

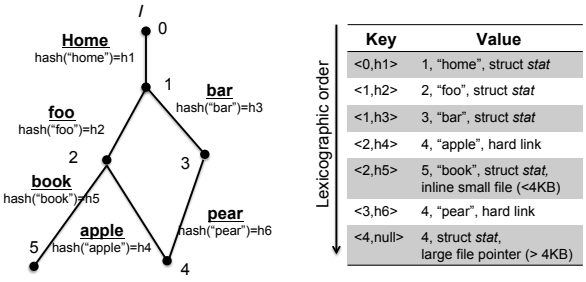


Figure 3: An example illustrates table schema used by TABLEFS's metadata store. The file with inode number 4 has two hard links, one called "apple" from directory *foo* and the other called "pear" from directory *bar*.

All the entries in the same directory have rows that share the same first 64 bits in their 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 their 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. Unlike BTRFS, TABLEFS does not need the second version of each directory entry because the entire attributes are returned in the *readdir* scan.

3.3 Hard Links

Hard links, as usual, are a special case because two or more rows must have the same inode attributes and data. Whenever TABLEFS creates a second hard link to a file, it creates a separate row for the file itself, with

a null name, and its own inode number as its parent's inode number in the row key. As illustrated in Figure 3, creating a hard link also modifies directory entry such that each row naming the file with an attribute indicating the directory entry is a hard link to the file object's inode row.

3.4 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 will not soon be necessary to recycle the inode number of deleted entries. Coping with operating systems that with 32 bit inode numbers may require frequent inode number recycling, a problem beyond the scope of this paper and shared by many file systems.

3.5 Locking and Consistency

LevelDB provides atomic batch insert but does not support atomic row read-modify-write operations. The atomic batch write guarantees that a sequence of updates to the database are applied in order, and committed to the write-ahead log atomically. Thus the *rename* operation can be implemented as a batch of two operations: insert the new directory entry and delete the stale entry. But for operations like *chmod* and *utime*, since all inode attributes are stored in one key-value pair, TABLEFS must read-modify-write attributes atomically. We implemented a light-weight locking mechanism in the TABLEFS core layer to ensure correctness under concurrent accesses.

3.6 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.

3.7 TABLEFS in the Kernel

A kernel-native TABLEFS file system is a stacked file system, similar to eCryptfs [13, 52], treating a second local file system as an object store, as shown in Figure 4(a). An implementation of a Log-Structured Merge (LSM) tree [33] used for storing TABLEFS in the associated object store, such as LevelDB [22], is likely to have an asynchronous compaction thread that is more conveniently executed at user level in a TABLEFS daemon, as illustrated in Figure 4(b).

For the experiments in this paper, we bracket the performance of a kernel-native TABLEFS (Figure 4(a)), between a pure user-level TABLEFS (Figure 4(b)) with no TABLEFS function in the kernel and all of TABLEFS in the user level FUSE daemon) and an application-embedded TABLEFS library, illustrated in Figure 4(c).

TABLEFS entirely at user-level in a FUSE daemon is unrealistically slow because of the excess kernel crossings and scheduling delays experienced by FUSE file systems [5]. TABLEFS embedded entirely in the benchmark application as a library is not sharable, and unrealistically fast because of the infrequency of system call. We approximate the performance of a kernel-native TABLEFS using the library version and preceding each reference to the TABLEFS library with a write("*/dev/null*", *N* bytes) to account for the system call and data transfer overhead. More details of these models will be discussed in Section 4.3.

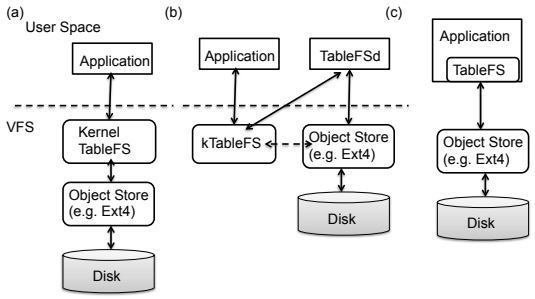


Figure 4: Three different implementations of TABLEFS: (a) the kernel-native TABLEFS. (b) the FUSE verisoin of TABLEFS. and (c) the library version of TABLEFS. In the following evaluation section, (b) and (c) are presented to bracket the performance of (a).

4 Evaluation

4.1 Evaluation System

We evaluate our TABLEFS prototype with Linux desktop computers equipped as follows:

Linux	Ubuntu 11.04, Kernel 3.2.0 64-bit version
CPU	AMD Opteron Processor 242 Dual Core
DRAM	DDR SDRAM 16GB
Hard Disk	Western Digital WD2001FASS-00U0B0
	SATA, 7200rpm, 2TB
	Random Seeks 100 seeks/sec peak
	Sequential Reads 137.6 MB/sec peak
	Sequential Writes 135.4 MB/sec peak

We compare TABLEFS with Linux's most sophisticated local file systems: Ext4, XFS, and BTRFS. Ext4 is mounted with "ordered" journaling to force all data to be flushed out to disk before its metadata is committed to disk. By default, Ext4's journal is asynchronously committed to disks every 5 seconds. XFS and BTRFS uses similar policies asynchronously update journals. BTRFS, by default, duplicates metadata and calculates checksums for data and metadata. We disable both features (unavailable in the other file systems) when benchmarking BTRFS. 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.

4.2 Data-Intensive Macro-benchmarks

We begin our evaluation with three coarse grain tests of the FUSE version of TableFS, the version which provides full featured, transparent application service. Instead of using a metadata-intensive workload, emphasized in the rest of this paper, we emphasize data-intensive work in this section. Our goals are to demonstrate that TableFS is capable of reasonable performance for the traditional workloads that are often used to test local file systems.

For the data in these data-intensive tests we use the Linux 3.0.1 source tree (whose compressed tar archive is about 73 MB in size). Our three macro-benchmarks are 1) untar, 2) grep "nonexistent pattern", and 3) gzip on the entire source tree. The testbed, described in Section 4.1, is allowed to use all 16 GB of memory.

Figure 5 shows the average of three runs of these three macro-benchmarks using EXT4, XFS, BTRFS and TABLEFS. TABLEFS using FUSE is 10-50% slower, but it is also paying significant overhead [5] caused by moving all data through the user-level FUSE daemon and the kernel twice, instead of only through the kernel once, as illustrated in Figure 4. Figure 4 also shows the much slower performance of Ext4 when it is accessed through FUSE.

In the next section we present our model for estimating TableFS performance without FUSE overhead, for metadata-intensive workloads.

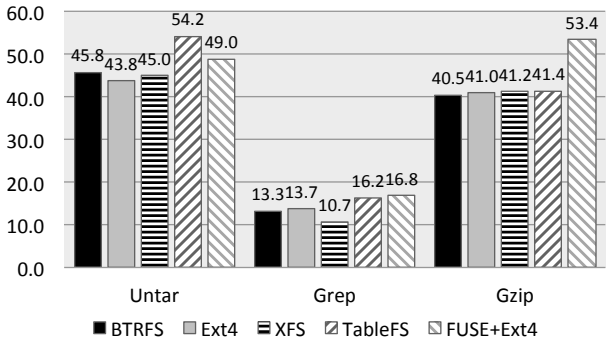


Figure 5: The elapsed time in seconds for unpacking, searching and compressing the Linux kernel package for four test systems.

4.3 TABLEFS-Predict Model

To understand the overhead of FUSE in TABLEFS-Predict, and estimate the performance of its in-kernel version, we ran a micro-benchmark against TABLEFS-FUSE and TABLEFS-Library ((b) and (c) shown in Figure 4). This micro-benchmark creates one million zero-length files in one directory starting with an empty file system. The amount of memory available to the evaluation system is 700 MB.

Figure 6 shows the total runtime of two experiments. TABLEFS-FUSE is more than 3X slower than TABLEFS-Libary. We also tracked disk traffic from Linux proc file system (*/proc/diskstats*). Figure 7 shows the total disk traffic during the test. TABLEFS-FUSE has a lot more bytes read from/written to the disk. This additional disk traffic results from two sources: 1) FUSE framework maintains its own inode cache, and its inode cache competes with the kernel's page cache that stores recently accessed SSTables. 2) Under a slower insertion rate, LevelDB tends to compact more often. For each compaction in Level 0, LevelDB will compact all SSTables with overlapping ranges. When the insertion rate is slow, compaction in Level 0 has less SSTables to compact at each time. Therefore, it tirggers more compactions to achieve the same level of balance.

To separate these two factors, we delibrately slow down TABLEFS-Library to run at the same speed of TABLEFS-FUSE by adding *sleep 150ms* every 1000 operations. This model of TABLEFS is called TABLEFS-Sleep and shown in Figure 6 and 7. Figure 8 shows the running behavior of three versions of TABLEFS. TABLEFS-Sleep causes almost the same number of compactions as does TABLEFS-FUSE. But unlike TABLEFS-FUSE, TABLEFS-Sleep can use more of kernel page cache to store SSTables than TABLEFS-FUSE. Thus, as shown in Figure 7, TABLEFS-Sleep writes the same amount of data as TABLEFS-FUSE does but with much less disk read traffic.

Clearly to estimate TableFS performance without FUSE overhead, we would like to reduce the double caching and emulate the real overhead of context switching between kernel and user-space. Therefore, we use TABLEFS-Sleep model with the following modification: Instead of sleeping, TABLEFS performs

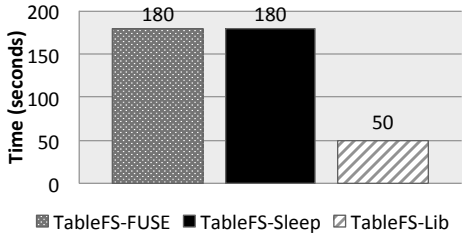


Figure 6: The elapsed time of 1M zero-length file being created on three versions of TABLEFS.

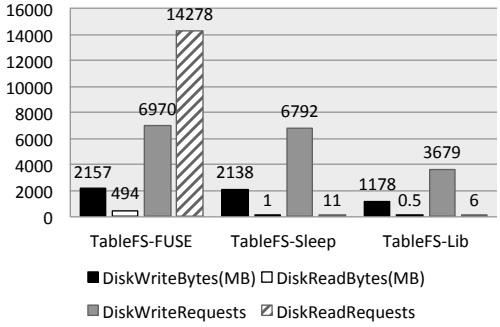


Figure 7: The total disk traffic in MB of zero-length files created.

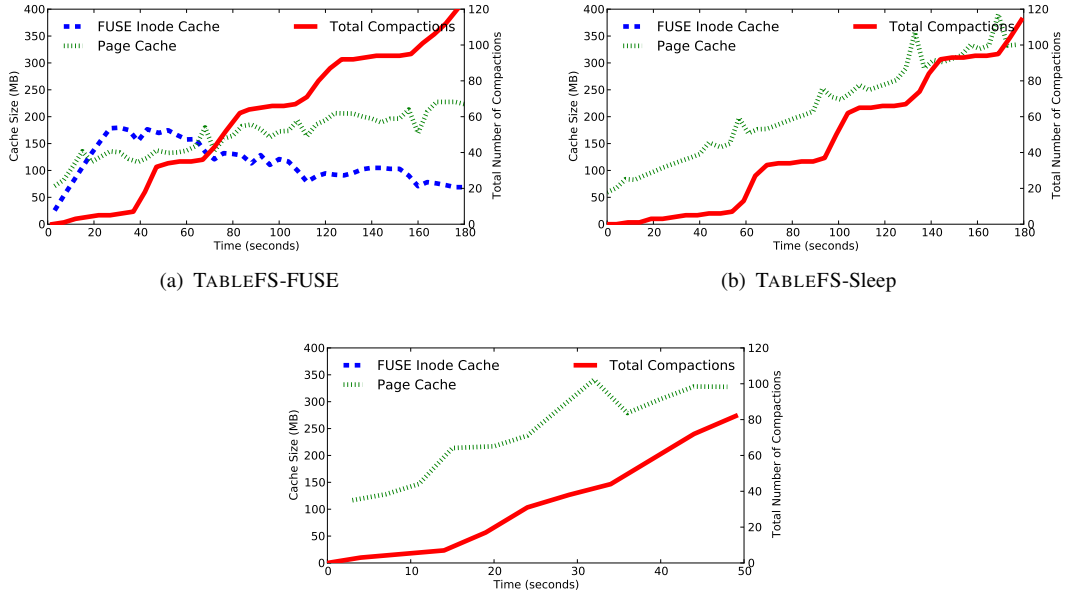
a *write*("*/dev/null*", *N* bytes) on every invocation to account for system call and argument data transfer overhead. This model is called TABLEFS-Predict which is used in the following sections to predict metadata efficiency of TABLEFS.

4.4 Benchmark with Metadata Only

In this section, we micro-benchmark the efficiency of pure metadata operations. The micro-benchmark consists of two phases. The first phase ("creation") generates a file system of two million files, all zero length and in one single directory. The second phase ("query") issues one million random read or write queries to random (uniform) files or directories. A read query calls *stat* on the file, and a write query randomly does either a *chmod* or *utime* to update the mode or the timestamp fields. Between the two phases, we umount and re-mount local filesystems to drop their caches, so that the second phases starts with a cold cache. To better understand the cache effects, we varies the machine's available memory from 350MB to 1500MB, by setting boot parameters of Linux. The former memory size will not fit the entire test in memory and the later will.

Figure 9 shows the performance in operations per second, averaged over three runs of the creation phase. TABLEFS-Predict and library version are almost 2X to 3X faster than the other tested file systems in workloads with larger memory (700MB and 1500MB). They achieve comparable creation performance in the smaller memory tests. The FUSE version is slower than other filesystems especially in the low memory case.

In the creation phase, all file systems start with an empty disk. The workload is not a random insertion workload for file systems such as XFS and Ext4 that have inodes and directory entries stored separately. In these filesystems, newly generated inodes can be sequentially written into the disk, because inodes are indexed by the monotonically increased inode number. And the total size of directory entires is small compared to the size of all the inode attributes. This overcomes the disadvantage of traditional B-trees for



(a) TABLEFS-FUSE

(b) TABLEFS-Sleep

(c) TABLEFS-Library

Figure 8: Cache usage and total number of compactions during the creation of 1M zero-length files for threee TABLEFS models. TABLEFS-Sleep causes almost the same number of compactions as does TABLEFS-FUSE. But unlike TABLEFS-FUSE, TABLEFS-Sleep can use more of kernel page cache to store SSTables than TABLEFS-FUSE.

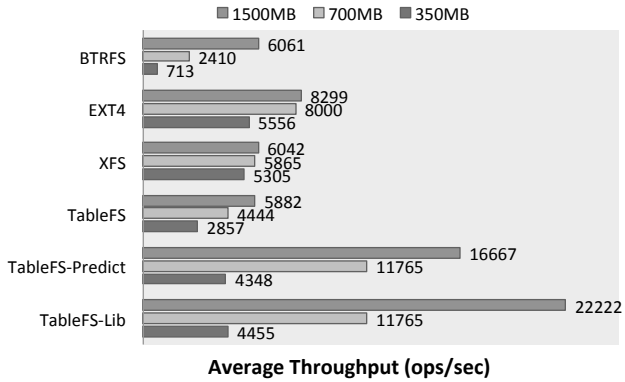
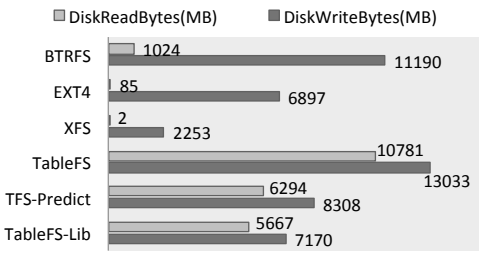


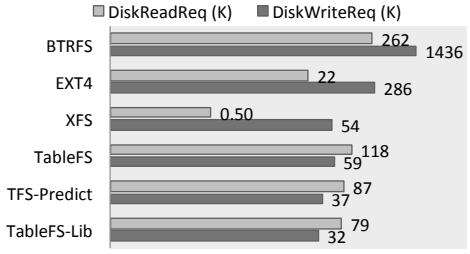
Figure 9: Performance of each file system in the **creation** phase of the metadata-only benchmark.

random insertion workloads. In such cases, TABLEFS does not win as much by sequentially logging every change. Thus, as only in larger memory cases, can TABLEFS take full advantage of its cache to reduce its random reading, and run faster than other file systems.

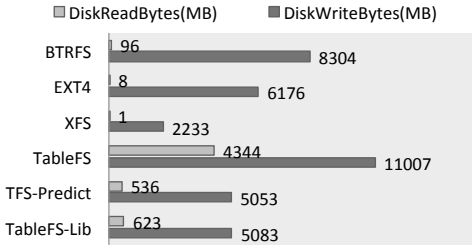
Figure 10 shows the total disk traffic (total size and requests) during the creation phase with memory size of 350MB and 700MB. Figure 10 (a) and (b), show that with 350MB physical memory, although TABLEFS reduces write disk traffic (the number of write requests) a lot, TABLEFS still causes 10X to 100X more read requests than Ext4 and XFS. In larger memory cases shown in Figure 10 (c) and (d), with bloom filtering and more caching, TABLEFS uses less read requests, and therefore its total number of disk requests is fewer than other tested file systems.



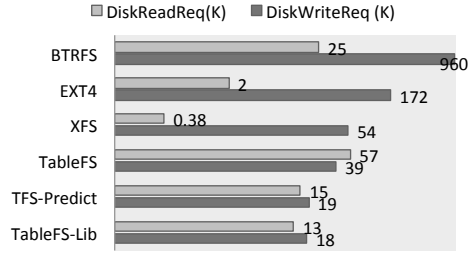
(a) Total disk bytes for 350MB memory (in megabytes)



(b) Total disk requests for 350MB memory (in thousands). Bars are shown in a log scale.



(c) Total disk bytes for 700MB memory (in megabytes)



(d) Total disk requests for 700MB memory (in thousands). Bars are shown in a log scale.

Figure 10: Total disk traffic during the **creation** phase of metadata-only benchmark. The horizontal axes of (b) and (d) are shown in log scale.

Figure 11 demonstrates the performance in operations per second, averaged over 3 runs of the query phase with 50% random read and 50% random write. TABLEFS *is 1.5X to 10X faster than the other tested file systems even in its FUSE version under all memory sizes.*

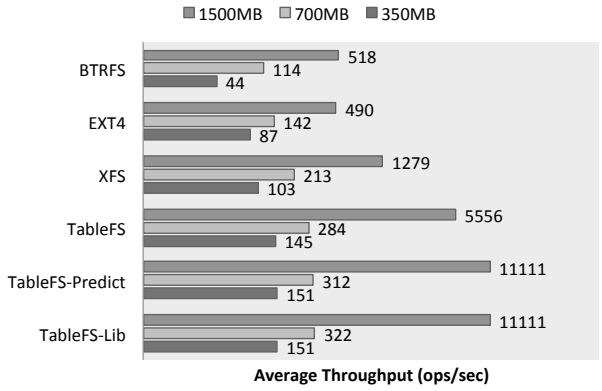
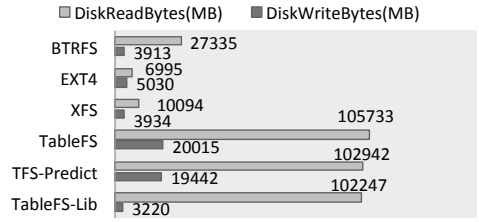


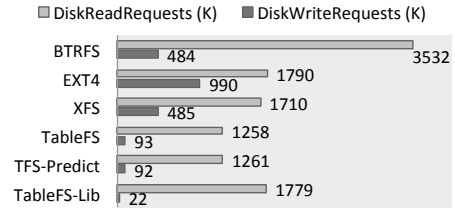
Figure 11: Performance of each file system in the **query** phase of the metadata-only benchmark.

Figure 12 shows the total disk traffic (total size and requests) during the query phase with memory size 350MB and 1500MB. Compared to other file systems, TABLEFS reduces write disk traffic (the number of write requests) a lot. This shows that using LevelDB effectively batches small random writes into large sequential writes. Since this workload starts with a cold cache *stat* of a file randomly involves lots of random seeks to the disk. For small memory size such as 350MB, since the datasets cannot fit into memory, each *stat* operation causes a cache miss. According to Figure 12 (b), for each cache miss, TABLEFS requires

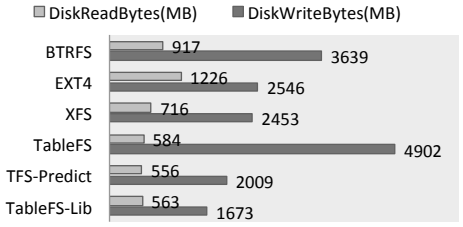
about two requests to read the data from disks, which is comparable to other tested file systems. Figure 12 (a) shows the total number of bytes read by TABLEFS is a lot more than other file systems. This is caused by compactions and prefetching from large sequential SSTables. For larger memory cases shown in Figure 12 (c) and (d), TABLEFS utilizes its cache well and reduces the disk read traffic significantly.



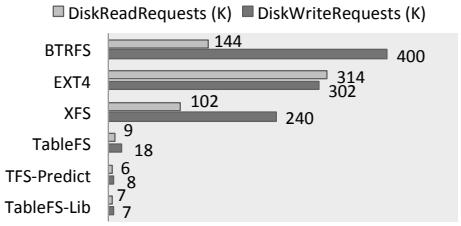
(a) Total disk bytes for 350MB memory (in megabytes)



(b) Total disk requests for 350MB memory (in thousands)



(c) Total disk bytes for 1.5GB memory (in megabytes)



(d) Total disk requests for 1.5GB memory (in thousands)

Figure 12: Total disk traffic during the **query** phase of metadata-only benchmark

4.5 Benchmark with Large Directories

To see TABLEFS's scalability for supporting large directories, we repeat create phase of the metadata-only benchmark, but increase the number of created zero-length files from 2 million to 100 million (a number of files rarely seen in the local file system today). In this benchmark, the memory available to the evaluation system is not limited, and all tested file systems can fully utilize all 16GB physical memory.

Figure 13 shows a throughput timeline for TABLEFS. In the beginning of this test, there is a throughput spike that is caused by everything fitting in the cache. Later in the test, the creation throughput of all tested file systems gradually slows down. BTRFS suffers serious throughput drop, slowing down to 100 operations per second at some points. TABLEFS maintains a more steady performance with an average speed of 2,200 operations per second: TABLEFS *is 10X faster than all other tested file systems.*

All tested file systems have throughput fluctuations during the test and the behavior of TABLEFS's throughput is more smooth than others. This kind of fluctuation in other file systems might be caused by load balancing or splitting in B-Tree. In TABLEFS, this behavior is caused by compactions in LevelDB, in which SSTables are quickly merged and sequentially written back to disks. LevelDB limits the amount of work to do in each compaction, and therefore its throughput is more steady than other file systems.

4.6 Benchmark with Small Files

The second micro-benchmark is similar to metadata-only benchmark except that we create one million small files with size 512B. All small files have the same content (and there is no compression to exploit this). In the query phase, read queries retrieve 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.

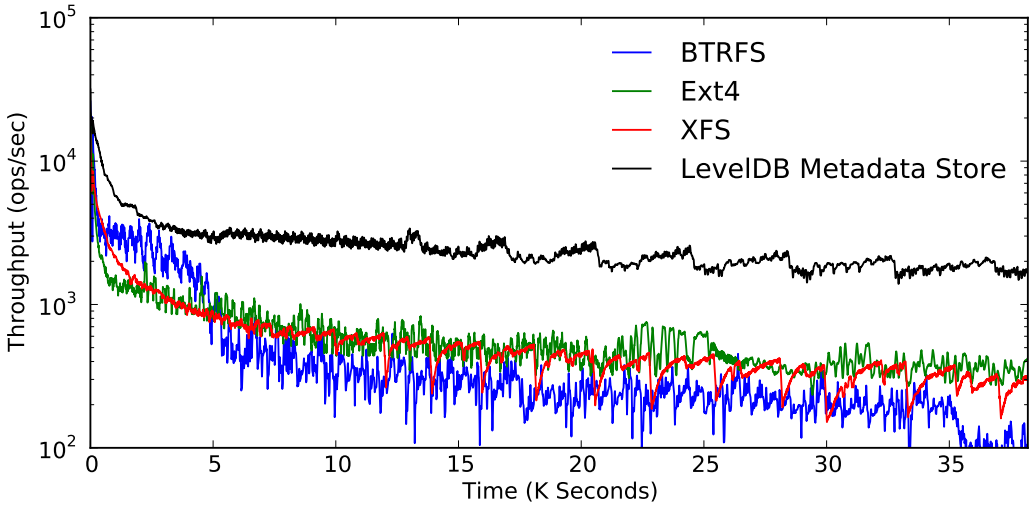


Figure 13: Throughput of creating 100 million zero-length files by four tested file systems during the test. We only graph the time until TABLEFS finished inserting all 100 million zero-length files, because the other file systems were much slower. TABLEFS is almost 10X faster than the other tested file systems in the later stage of this experiment. The data is sampled in every 10 seconds and smoothed over 100 seconds. The veritical axis is shown in a log scale.

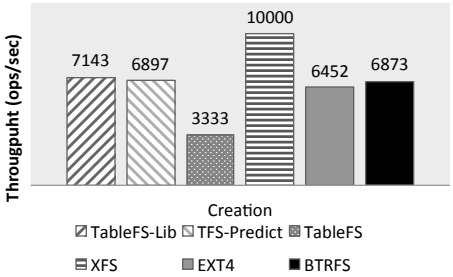


Figure 14: Avarage throughput during the **creation** phase in the small file benchmark where the available memory is 700MB.

Figure 14 shows the results of the creation phase. As in the metadata only benchmark, TABLEFS FUSE and Library are all slower than XFS, the fastest local file system. The performance gap between TABLEFS and XFS in small file creation workload is larger than in the metadata-only workload. That is because with sufficient memory, TABLEFS do not gain much from reducing write seeks. Moreover, since data is inline with metadata in one row in LevelDB, new file data will also be put into LevelDB's write-ahead log, wasting more disk bandwidth.

To understand TABLEFS's performance gain from read and write requests, we also varies the ratio of read and write queries in the query phase. Figure 15 shows the performance in operations per second, averaged over the query phase, for three different ratios of read and write queries: (1) 10% read and 90% write queries, (2) 50% read and 50% write queries, and (3) 90% read and 10% write queries. In the query phase, TABLEFS outperforms all other file systems by 2X. The performance gap between TABLEFS and other file systems are larger in the workload with 90% write queries. This shows TABLEFS is better optimized for random update queries than other file systems.

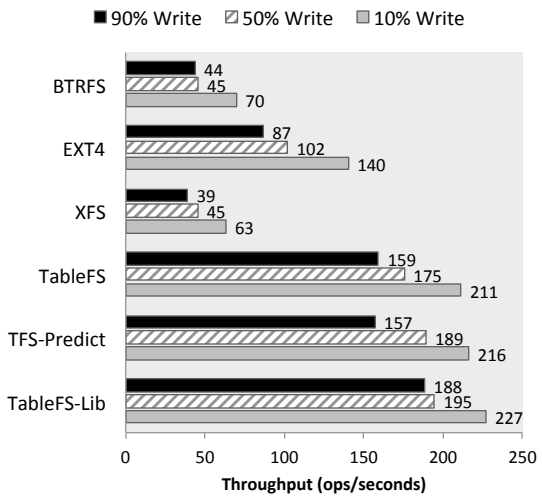


Figure 15: Performance during the **query** phase for all small file benchmarks under different write ratio, by limiting available memory size of 700MB.

4.7 Benchmark with *readdir*

Besides point queries such as *open, mknod* and *stat*, range queries such as *readdir* are also an important metadata operation. To test the performance of *readdir*, we run a benchmarks that performs multiple *readdir* in a realistic desktop filesystem tree. This benchmark first generates a filesystem with 1 million files, all with size 512B. This file system has the same namespace as one author's personal Ubuntu desktop, there are 172,252 directories, each with 11 files on average, and the average depth of the namespace is 8. The benchmark creates this test namespace in depth first order. The query phase issues 10,000 *readdir* on randomly chosen directories. Between the creation phase and query phase the file system is umounted to clean the cache. This experiment runs with 700 MB available memory.

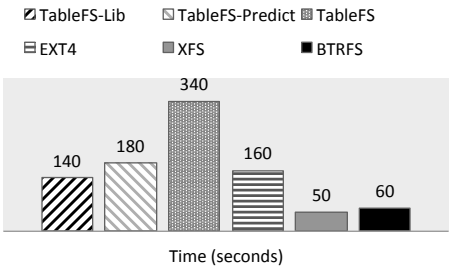
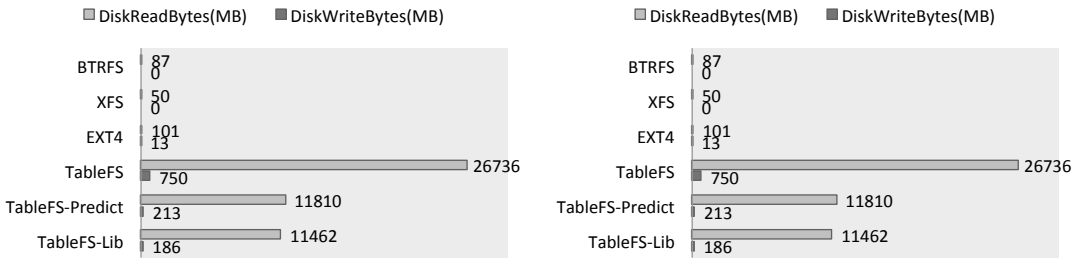


Figure 16: The elapsed time for the entire run of *readdir* benchmark.

Figure 16 shows the results of the *readdir* query phase. TABLEFS is slower than XFS and BTRFS because of read amplification, that is, for each *readdir* operation, TABLEFS fetches directory entries along with inode attributes and file data.

Figure 17 shows the disk traffic during the test. TABLEFS reads a lot more than the other file systems. Figure 17 also shows that TABLEFS incurs write-disk traffic during a read-only workload. This is due to the compaction in the underlying LevelDB. LevelDB maintains an individual counter of false-positive lookups for each SSTable. If one SSTable receives too many false-positive lookups, a compaction will be triggered to merge this SSTable with other SSTables within the same key range.



(a) Total disk bytes for *readdir* benchmark

(b) Total disk requests for *readdir* benchmark

Figure 17: Total disk traffic during the **query** phase of *readdir* benchmark

Figure 18 shows the results of an "ls -l" workload when for each entry returned by *readdir()*, the benchmark does a *stat* on this entry. Since *stat* causes an additional random lookup, the other file systems slow down a lot. However, this does not incur additional overheads for TABLEFS.

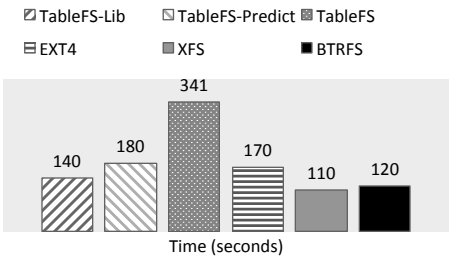


Figure 18: The elapsed time for the entire run of *readdir+stat* benchmark.

4.8 Postmark Benchmark

Postmark was designed to measure the performance of a file system used for e-mails, and web based services [19]. It creates a large number of small randomly-sized files between 512B and 4KB, performs a specified number of transactions on them, and then deletes all of them. Each transaction consists of two sub-transactions, with one being a create or delete and the other being a read or append. The configuration used for these experiments consists of two million transactions on one million files, and the biases for transaction types are equal. The experiments were run against TABLEFS FUSE version with the available memory set to be 1500 MB.

Figure 19 shows the Postmark results for the four tested file systems. Again, TABLEFS outperforms other tested file systems by at least 23% less time. Figure 20 gives the average throughput of each type of operations individually. Similar to previous experiments, TABLEFS runs faster than other tested filesystems for transaction operations, and is slower in *creation*. In LevelDB, *deletion* is implemented as inserting entries with deletion marks. The actual deletion is delayed until compaction procedure reclaims the deleted entries. Such implementation is not as efficient as XFS and Ext4, possibly because XFS and Ext4 can quickly reclaim deleted inodes whose inode numbers are continuous in a range.

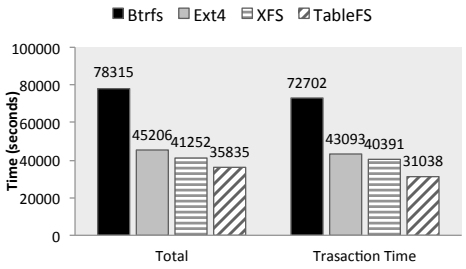


Figure 19: The elapsed time for both the entire run of postmark and the transactions phase of postmark for the four test systems.

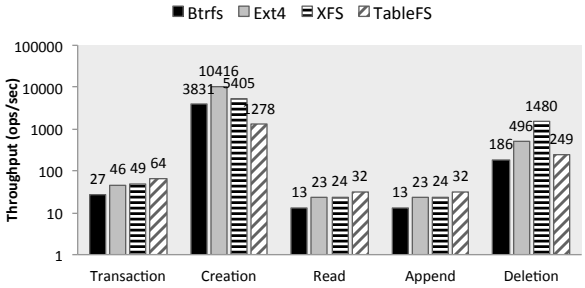


Figure 20: Average throughput of each type of operation in postmark benchmark.

5 Related Work

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 [46, 31]. Modern stacked file systems often expand on the limited structure in file systems, hiding structures inside directories meant to represent files [5, 13, 14, 20, 52], 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 [15, 18, 24, 40, 51].

Early file systems stored directory entries in a linear array in a file and inodes in simple on-disk tables, separate from the data of each file. Clustering within a file was pursued aggressively, but for different files clustering was at the granularity of the same cylinder group. It has long been recognized that small files can be packed into the block pointer space in inodes [29]. C-FFS [11], takes packing further and clusters small files, inodes and their parent directory's entries in the same disk readahead unit, the track. A variation on clustering for efficient prefetching is replication of inode fields in directory entries, as is done in NTFS[8]. TABLEFS pursues an aggressive clustering strategy; each row of a table is ordered in the table with its parent directory, embedding directory entries, inode attributes and the data of small files. This clustering is manifest as adjacency in objects in the lower level object store if these entries were create/updated close together in time, or after compaction has put them back together.

Beginning with the Log-Structured File System (LFS)[37], file systems have exploited write allocation methods that are non-overwrite, log-based and deferred. Variations of log structuring have been implemented in NetApp's WAFL, Sun's ZFS and BSD UNIX [3, 17, 43]. In this paper we are primarily concerned with the performance implications of non-overwrite and log-based writing, although the potential of strictly

ordered logging to simplify failure recovery has been compared to various write ordering schemes such as Soft Updates and Xsyncfs [27, 44, 30]. LevelDB's recovery provisions are consistent with delayed periodic journalling, but could be made consistent with stronger ordering schemes. It is worth noting that the design goals of BTRFS call for non-overwrite (copy-on-write) updates to be clustered and written sequentially[36], largely the same goals of LevelDB in TABLEFS. Our measurements indicate, however, that the BTRFS implementation ends up doing far more small disk accesses in metadata dominant workloads.

Partitioning the contents of a file system into two groups, a set of large file objects and all of the metadata and small files, has been explored in hFS[53]. In their design large file objects do not float as they are modified, and the metadata and small files are log structured. TABLEFS has this split as well, with large file objects handled directly by the backing object store, and metadata updates approximately log structured in LevelDB's partitioned LSM tree in the same backing object store. However, TABLEFS does not handle disk allocation, relying entirely on the backing object store to handle large objects well.

Log-Structured Merge trees [33] were inspired in part by LFS, but focus on representing a large B-tree of small entries in a set of on-disk B-trees constructed of recent changes and merging these on-disk B-trees as needed for lookup reads or in order to merge on-disk trees to reduce the number of future lookup reads. LevelDB [22] and TokuFS [10] are LSM trees. Both are partitioned in that the on-disk B-trees produced by compaction cover small fractions of the key space, to reduce unnecessary lockup reads. TokuFS names its implementation a Fractal Tree, also called streaming B-trees[4], and its compaction may lead to more efficient range queries than LevelDB's LSM tree because LevelDB uses Bloom filters[6] to limits lookup reads, a technique appropriate for point lookups only. When bounding the variance on insert response time is critical, compaction algorithms can be made more carefully scheduled, as is done in bLSM[42]. TABLEFS may benefit from all of these improvements to LevelDB's compaction algorithms, which we observe to sometimes consume disk bandwidth injudiciously.

6 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 10X better than state-of-the-art local file systems in extensive metadata update workloads.

References

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

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

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

[4] Michael A. Bender and et al. Cache-oblivious streaming B-trees. In *SPAA*, 2007.

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

[6] B.H. BLOOM. Space/time trade-offs in hash coding with allowable errors. *Communication of ACM 13, 7*, 1970.

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

[8] H. Custer. Inside the windows NT file system. *Microsoft Press*, 1994.

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

[10] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. The TokuFS streaming file system. *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems (HotStorage12)*, 2012.

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

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

[13] Michael Austin Halcrow. eCryptfs: An Enterprise-class Encrypted Filesystem for Linux. *Proc. of the Linux Symposium, Ottawa, Canada*, 2005.

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

[15] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan. Recovery management in quicksilver. *In Proceedings of the Eleventh ACM Symposium on Operating System Principles*, 1987.

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

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

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

[19] Jeffrey Katcher. Postmark: A new file system benchmark. Technical report, NetApp, 1997.

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

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

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

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

[24] Barbara Liskov and Rodrigo Rodrigues. Transactional file systems can be fast. *Proceedings of the 11th ACM SIGOPS European Workshop*, 2004.

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

[26] Avantika Mathur and et al. The new Ext4 lesystem: current status and future plans. In *Ottawa Linux Symposium*, 2007.

[27] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. *USENIX Annual Technical Conference, FREENIX Track*, 1999.

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

[29] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *SoftwarePractice and Experience*, 1984.

[30] EDMUND B. NIGHTINGALE, KAUSHIK VEERARAGHAVAN, PETER M. CHEN, and JASON FLINN. Rethink the sync. *ACM Transactions on Computer Systems, Vol.26, No.3 Article 6*, 2008.

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

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

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

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

[35] Ohad Rodeh. B-trees, shadowing, and clones. *TOS*, 2008.

[36] Ohad Rodeh, Josef Bacik, and Chris Mason. BRTFS: The Linux B-tree Filesystem. *IBM Research Report RJ10501 (ALM1207-004)*, 2012.

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

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

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

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

[41] Russell Sears and et al. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem? *Microsoft Technique Report*, 2007.

[42] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD12)*, 2012.

[43] Margo I. Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *USENIX Winter Technical Conference*, 1993.

[44] Margo I. Seltzer and et al. Journaling versus soft updates: Asynchronous meta-data protection in file systems. *USENIX Annual Technical Conference*, 2000.

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

[46] Michael Stonebraker. Operating System Support for Database Management. *Commun. ACM*, 1981.

[47] Michael Stonebraker and Ugur Çetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*, 2005.

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

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

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

[51] CHARLES P. WRIGHT, RICHARD SPILLANE, GOPALAN SIVATHANU, and EREZ ZADOK. Extending ACID Semantics to the File System. *ACM Transactions on Storage, Vol.3, No.2*, 2007.

[52] Erez Zadok and Jason Nieh. FiST: A Language for Stackable File Systems. *USENIX Anual Technical Conference*, 2000.

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