

# BabuDB: Fast and Efficient File System Metadata Storage

Jan Stender, Björn Kolbeck, Mikael Höggqvist  
*Zuse Institute Berlin*  
Berlin, Germany  
Email: stender@zib.de, kolbeck@zib.de,  
hoeggqvist@zib.de

Felix Hupfeld  
*Google GmbH*  
Zurich, Switzerland  
Email: hupfeld@google.com

**Abstract**—Today’s distributed file system architectures scale well to large amounts of data. Their performance, however, is often limited by their metadata server. In this paper, we reconsider the database backend of the metadata server and propose a design that simplifies implementation and enhances performance.

In particular, we argue that the concept of log-structured merge (LSM) trees is a better foundation for the storage layer of a metadata server than the traditionally used B-trees. We present BabuDB, a database that relies on LSM-tree-like index structures, and describe how it stores file system metadata.

We show that our solution offers better scalability and performance than equivalent ext4 and BerkeleyDB-based metadata server implementations. Our experiments include real-world metadata traces from a Linux kernel build and an IMAP mail server. Results show that BabuDB is up to twice as fast as the ext4-based backend and outperforms a BerkeleyDB setup by an order of magnitude.

## I. INTRODUCTION

Modern distributed file system architectures, like the object-based design [1], [2] or the Google File System [3], separate the management of metadata from the storage of the file content. These architectures have been proven to easily scale the storage capacity and bandwidth. However, the management of the metadata remains a bottleneck [4], [5], [6]. Studies have shown that approximately 75% of all file system calls access file metadata [7]. Therefore, the efficient management of metadata is crucial for the overall file system performance. As a metadata server is only a thin layer on top of a database, improving the performance of the underlying database is the key to better performance of the file system.

File systems traditionally use variations of B-trees as storage backends for their metadata, e.g. ZFS, btrfs, Lustre [8] or PVFS2. We break with this tradition and propose to use an index structure similar to a log-structured merge (LSM) tree [9] as a simpler and more efficient alternative for metadata storage. These search trees are multi-version data structures composed of several in-memory trees and an on-disk index. The on-disk index is an immutable and sorted list of records. Updates are applied to an active in-memory tree and persistently stored in an operations log. A storage backend based on such an index structure is well suited for

managing file system metadata for the following reasons:

- It can handle short-lived files particularly well: updates to the metadata are persistently written to the operations log, but do not modify the on-disk index and do not require a re-balancing of a tree on disk. Since about 50% of all files are deleted within 5 minutes and 20% exist for less than 30 seconds [10], the efficient handling of these files is important for the overall performance.
- Atomic operations that require an atomic update of multiple records such as `rename` can be done without expensive transaction management. Since updates affect only the in-memory trees, there is no need for locking pages to prevent incomplete writes.
- Since data is never changed in-place, our index structure can easily handle variable-length keys and data. This facilitates the handling of file system metadata like extended attributes or ACLs.
- Listing directory content is particularly efficient. Since the on-disk index is sorted, a directory listing results in sequential reads of contiguous data.
- Database snapshots can be created instantaneously and written to disk in an asynchronous fashion. This is an important building block to implement file system snapshots.

Several other properties of our index structure enhance the performance on modern computer systems and simplify the implementation:

- Since the on-disk index is immutable and sorted, its data can easily be compressed resulting in a more effective memory and disk bandwidth [11], [12]. Similarly, the layout of the on-disk index can be designed for cache-efficiency as it does not need to handle updates.
- There is no need for custom memory management to control the swapping of dirty pages. The on-disk index is mapped read-only into memory letting the operating system take care of paging.
- The lack of in-place overwrite of data reduces the number of disk seeks significantly. Our indices also do not need a block allocator that manages space on disk, which simplifies the implementation substantially.

In this paper, we present BabuDB, a database which is based on LSM-tree-like index structures. BabuDB is used as the storage backend for metadata in the object-based file system XtreamFS [13]. Both, BabuDB and the XtreamFS metadata server are implemented in Java. Using LSM-tree-like index structures leads to both less and simpler code: our core index classes have about ten times less lines of code compared to BerkeleyDB for Java.

We first present the general design of BabuDB in section II. We continue with a description of how we efficiently map the file system metadata onto a flat index structure. In section IV, we compare the performance of our approach with ext4 to show how our approach performs for metadata management. As BabuDB is implemented in Java, we compare it also with BerkeleyDB for Java (BDB4J). We show that BabuDB is up to twice as fast as ext4 and an order of magnitude faster than BDB4J while providing additional functionality. An overview of the related work follows in section V.

## II. DESIGN OF BABUDB

A BabuDB database consists of a set of indices, a log manager and a checkpointer. Indices are data structures optimized for searching and storing records. Each record consists of a key and a value of variable length. The log manager persistently logs database modifications. When adding, deleting or updating records, it appends new log entries to a persistently stored operations log. The state of the database can be restored from the operations log after a system restart or crash. As soon as the size of the log exceeds a threshold, a checkpoint is created asynchronously by the checkpointer. After successful creation of a checkpoint the operations log is truncated. Details about indices and the internals of BabuDB will be described in the remainder of this section.

### A. Indices

An index in BabuDB consists of a list of  $N$  in-memory trees and a single on-disk index (see Fig. 1). Records are inserted, updated and deleted in the *active* tree (i.e. tree  $N$ ), whereas all other trees and the on-disk index remain immutable. A lookup is done by searching through all in-memory trees from  $N$  to 1 and finally the on-disk index. As soon as a record with a matching key is found, it is returned. This ensures that a lookup always returns the latest version of a record, as the list of trees is sorted such that 1 refers to the oldest and  $N$  refers to the newest tree.

*In-memory Snapshots:* A *snapshot* is a consistent view of the state of the index at a certain point in time. An in-memory snapshot is created by appending a new (empty) tree to the list of in-memory trees (see Fig. 1). Thereby, the newly appended tree  $N + 1$  becomes the active tree, and the formerly active tree  $N$  becomes immutable. Thus, the

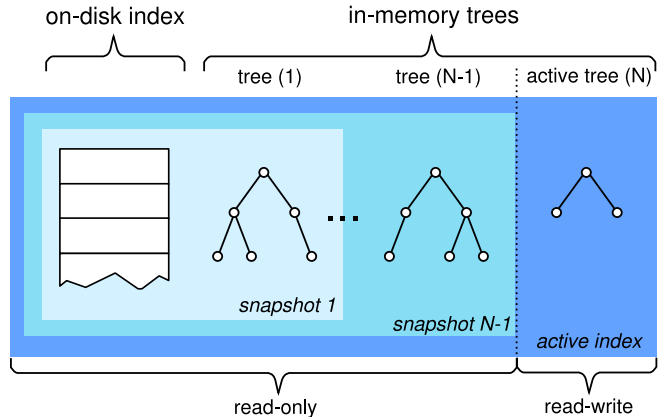


Figure 1: Illustration of a BabuDB index. Each index contains a single mutable *active* in-memory tree, zero or more immutable in-memory trees and an immutable on-disk index.

state of a snapshot  $K$  is determined by the state of snapshot  $K - 1$  and tree  $K$ . Lookups on an in-memory snapshot  $K$  are performed in the same manner as regular lookups on the active index, except that the search starts on tree  $K$  instead of the active tree.

*Record Deletion:* When deleting a record, it is important to keep in mind that one or more of the immutable trees may contain older versions of the record. To guarantee correct lookups, it is necessary to mark a record as deleted instead of removing it from the active tree. This is done by adding a special tombstone record. When a tombstone record is found while searching the trees, the lookup aborts and returns a result indicating that the record is not in the index.

*Prefix Lookups:* In addition to normal lookups, BabuDB supports prefix lookups that return all records with keys that start with a given prefix key. The result of a prefix lookup is an iterator that returns the respective records in ascending key order. To create such an iterator, the prefix lookup is executed on all in-memory trees. The resulting iterators are incrementally merged, so that only the latest version of a record is returned if records with the same key are contained in several iterators. Prefix lookups are implemented as a special case of range lookups.

*On-Disk Index:* An on-disk index is an immutable and persistently stored list of records sorted by keys. The on-disk index is created by materializing the current state of the database to disk. It contains three parts (Fig. 2): A sorted list of records which is divided into *blocks*, where each block contains a fixed number of key-value pairs, a *block index* that lists the first key and the offset of each block and a *pointer* to the start of the block index.

Looking up a record requires two steps. First, the block index is searched for the block that covers the range in which

the requested key is in. Then, the block itself is searched for the matching record. We use binary search to find keys, both within the block index and within the blocks.

The block index is always kept in memory. Therefore, each lookup requires at most one disk seek. Additionally, by mapping the index file into memory using `mmap`, indices larger than the main memory are supported. Using `mmap` leaves the memory management to the operating system and thus avoids the complexity of a separate disk block manager.

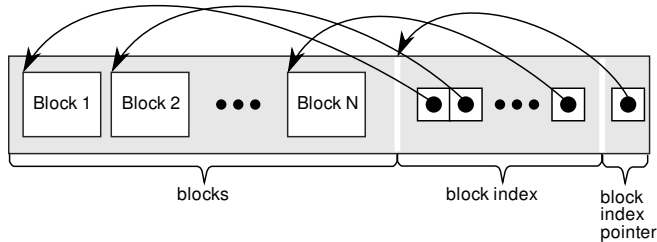


Figure 2: Layout of an on-disk index file.

The internal layout of a block has been inspired by PAX [14], which improves cache locality by grouping keys and values together within a block. Each block starts with a header followed by two mini-pages, one for keys and one for values. The header contains the number of entries in the block and a pointer to the mini-page with the values. The mini-pages store their entries sequentially followed by a list of pointers to each entry. This design allows for variable length entries within a mini-page.

### B. Checkpoints

Frequent insertions and deletions of records may cause a growing memory footprint of the database, even if the database itself does not grow, e.g. because of new trees being created and tombstone records being added. Moreover, a large operations log slows down the recovery of the database when the system is restarted.

From time to time, checkpoints are created to clean up the database and to truncate the operations log. They are initiated by the checkpointer without interfering with regular database operations. When creating a checkpoint, a new in-memory snapshot is created. This snapshot is then traversed using a prefix lookup. All records returned by the prefix lookup are sequentially written to a new on-disk index file. Finally, the current on-disk index file is replaced with the new file. All in-memory trees but the active one are discarded, and the obsolete operations log is deleted.

### C. Persistent Snapshots

In-memory snapshots can capture and freeze former states of an index, but do not survive checkpoints or system restarts. To make snapshots persistent, they can be written to individual on-disk index files before a checkpoint is created. Subsequent accesses to a snapshot will then be directed to a

new immutable index backed by the respective on-disk index file. To ensure that (in-memory) snapshots can be restored if the system is restarted between a snapshot and a checkpoint, a log entry recording the snapshot creation is appended to the operations log when the snapshot is created.

In some cases, it may be sufficient to only capture a subset of all records of an index in a snapshot. Such partial snapshots save disk bandwidth and storage space. The set of records to be included in a partial snapshot can e.g. be specified by means of key patterns or prefixes. When writing the on-disk index for a partial snapshot, only records with keys that match one of the given prefixes or patterns will be included.

### D. Concurrency

To further simplify our architecture, we decided to reduce concurrency to a shared reader/exclusive writer access per index. This means that concurrent modifications are not allowed on the active index. However, lookups can be executed in parallel. Snapshots are immutable and can always be accessed independently, which allows us to do checkpoints and to write snapshots asynchronously in the background. As a consequence, there is no need to implement complex locking, latching and transaction handling mechanisms.

Even without concurrency, it is sometimes necessary to update multiple records atomically. We support atomic updates by grouping multiple record updates into an *insert group*. When an insert group is processed, a single log entry is appended to the operations log instead of a set of individual log entries for each record. In the event of a log replay, the insert group will be processed as a whole, thus preserving atomicity. Since there is an exclusive write lock on the index, all updates from the insert group will be perceived as an atomic operation by other readers.

## III. FILE SYSTEM METADATA MANAGEMENT

In this section we describe how the file system metadata is stored and managed with BabuDB.

### A. Metadata Mappings

BabuDB records are arranged in a flat namespace, whereas file systems maintain their files in hierarchically arranged directory trees. Thus, maintaining file system metadata in BabuDB requires the directory tree with the metadata of all nested files and directories to be mapped to a non-hierarchical set of database records.

With respect to the mapping of metadata, it is important to consider that any access to a file or directory requires each directory on the path to be checked for access rights. Getting owner and permissions of a file or directory should therefore require no more than a single database lookup. To access directories as fast as possible, the `readdir` operation that lists the content of a directory should map to a single prefix

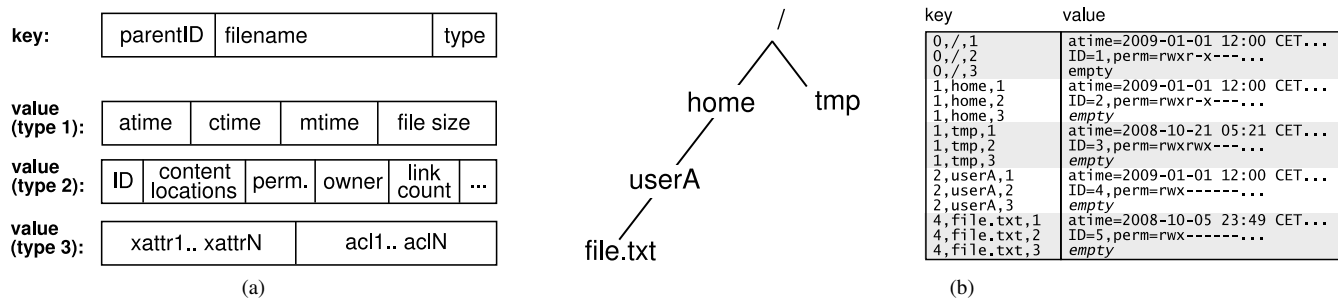


Figure 3: (a) Metadata layout. (b) Illustration of a mapping.

lookup in the database. This means that all directory entries must have a common key prefix.

We decided to use record keys that consist of three parts: the file’s or directory’s parent directory ID, the name and a type. Thus, the metadata for each path component can be retrieved with a single lookup, and the `readdir` operation can retrieve the content of a whole directory including `getattr` results for each nested element with a prefix lookup to the parent directory’s ID.

Some attributes of a file’s or directory’s metadata are updated more frequently than others. Time stamps and file sizes are changed more often than metadata related to e.g. file content locations, ownership, access control or extended attributes. Because of the different access patterns, we decided to split up the metadata of files and directories into three different records, as shown in Fig. 3. The *type 1* records contain frequently updated attributes, *type 2* and *type 3* records contain rarely updated attributes. Thus, most metadata updates affect only one of the three records, which results in small log entries to be written to the operations log.

### B. Hard Links

The mapping described above does not allow a file to be linked to more than a single directory, as the unique database key under which the metadata is stored reflects a certain file name and parent directory. To support hard links with POSIX semantics, an additional hard link index is used. The hard link index resembles the original metadata index, except that its keys start with file IDs instead of parent IDs and file names. It stores the metadata of linked files, i.e. files to which hard links have been created.

The `link` operation requires multiple steps to create a hard link to an existing file. First, it checks whether the file is linked already. A linked file is identified via a special link record in the original metadata index, with a value that only contains the linked file’s ID. If a link record is encountered when looking up the file’s metadata, a new link can be added by inserting an additional link record with the new link’s parent ID and name as the key and the linked file’s ID as the

value. Adding a link also requires the link count in the type 2 metadata to be incremented. The link count is decremented each time a link is removed; if it becomes zero, all of the file’s metadata can be deleted.

If the file is not linked, i.e. normal metadata records are encountered, it is necessary to make the file a linked file. This includes an insertion of its metadata records in the hard link index and a deletion of them from the metadata index. In addition, a link record for the original link needs to be added to the metadata index.

Retrieving metadata of a file with multiple hard links first requires a retrieval of the file ID from the link record in the metadata index. In a second step, the file ID can be used to retrieve the metadata from the hard link index. As we expect multiple hard links to a file to be the exception rather than the norm, we consider the overhead caused by an additional lookup as acceptable.

### C. Atomic Operations

As defined in the POSIX standard, file system operations such as `rename` and `link` have to be atomic, regardless of whether they require multiple modifications in the storage backend. The correctness of many applications, e.g. the Dovecot mail server, depends on this atomicity. Any storage backend for a metadata server must therefore support atomic executions of multiple modifications. Traditional databases can offer such semantics via transactions, which, however, often involve a considerable performance penalty. With BabuDB, atomic insert groups can be used as a lightweight alternative.

### D. File System Snapshots

To take a snapshot of a distributed file system, it is necessary to take a snapshot of the file system’s metadata. A BabuDB storage backend can instantaneously do this by taking a database snapshot. Since all files stored in a directory are mapped to records with the same key prefix, snapshots of single directories can be easily created by means of a partial snapshot of all records with the key prefix. A recursive snapshot of a directory also requires the key prefixes of all subdirectories to be included.

## IV. EVALUATION

To assess the performance and scalability of our approach, we conducted experiments with the XtreamFS metadata server with three different storage backends: BabuDB, BerkeleyDB for Java and ext4. We decided to include BerkeleyDB for Java as it is based on a B+tree. The ext4 backend uses a regular ext4 file system to store the metadata as directories and empty files. This approach is similar to Lustre which uses ext4 as the storage backend for its metadata server. We choose ext4 because it implements an HTree, a variation of the B-tree optimized for file system workloads.

We used the unmodified XtreamFS metadata server (MRC) from the XtreamFS release 1.0RC1. For the BerkeleyDB for Java and ext4 backends we took the code of the XtreamFS MRC and replaced the storage backend. This allows us to use identical code for most parts (except backend) which should yield a fair comparison.

The first experiment measures the duration of file create operations with the BabuDB backend. We compare the duration during normal operations and while taking an asynchronous database snapshot to assess the performance impact of writing snapshots. The second experiment compares the scalability of BabuDB, BerkeleyDB for Java and ext4 by creating a large number of files in the same directory. To evaluate the BabuDB on-disk file layout, we also compare the performance for a directory listing (`ls`) from cache and directly from disk (flushing the system’s page, inode and dentry cache). Finally, we use two real-world workloads to compare the performance of the different backends. We recorded the metadata operations executed during a Linux kernel build and from the Dovecot IMAP server. These traces were replayed against all three backends.

### A. Setup

All experiments were done on a single machine (one Xeon E5335 CPU with four cores, 4 GB RAM and two 73GB SAS 10k RPM hard drives) running Linux kernel 2.6.27. We used OpenJDK 1.7 (build 1.7.0-ea-b52) as we required the features of NIO2 for the ext4 backend. We used version 3.3.82 of BerkeleyDB for Java, locking and transactions were disabled and deferred writes enabled.

### B. Asynchronous Snapshots

One of the main features of BabuDB is the ability to create database snapshots and checkpoints asynchronously without interrupting normal operations of the MRC. In this experiment we measured the latency of requests executed sequentially during normal operations and while writing a database snapshot concurrently. We used one disk for the BabuDB operations log and the other disk for the database snapshot files. Table I shows the results for creating 100,000 and 1,000,000 files, respectively. The results demonstrate that BabuDB can provide acceptable response rates even

while writing snapshots to disk. The high standard deviation  $\sigma$  together with the low  $P_{99}$  for 1,000,000 files during snapshot indicate that there are some outliers with very high latency (maximum is 1.5 ms). However, the vast majority of the requests ( $P_{99}$ ) showed only an increase of up to 30% in the response time while writing a snapshot.

### C. File Creates

We submitted several batch requests of 1,000 to 1,000,000 create operations to the MRC using pipelining. After all files have been created, we executed an `ls` operation (`readdir` plus `getattr` for each file). We measured the total duration of the `create` and `ls` with and without the influence of system caches. To disable caching, we forced BabuDB to create a checkpoint, forced all updates to be written to disk (`sync`) and dropped all system caches between creating the files and executing the `ls`. The results for file creates (Fig. 4, left) show that the ext4 and BabuDB backend exhibit a similar performance for up to 100,000 files in a directory, but BabuDB scales better with larger directories. BerkeleyDB for Java has a good performance for very small databases but does not scale well. Creating one million files was not possible with BerkeleyDB for Java due to timeouts in the client.

The performance characteristics shown for `ls` (Fig. 4, right) are similar. For very large directories, BabuDB is even faster with flushed caches, i.e. when reading from disk. This is caused by the data structure used for the in-memory trees (Java’s red-black trees), which is not optimized for range reads.

### D. Linux Kernel Build

We used a FUSE module to record all metadata operations executed while building the Linux kernel (2.6.27). We replayed the trace sequentially with the BabuDB, the ext4 and the BerkeleyDB for Java backend and measured the total execution time. The trace consists of 9.9 million operations (44% `getattr`, 40% `open`, 15% `readlink`, 1% others). BabuDB and the ext4 backend showed a similar performance, BabuDB took 1,799 seconds and ext4 took 1,904 seconds. In contrast, BerkeleyDB for Java needed

# files	avg. (ms)	$\sigma$ (ms)	$P_{99}$ (ms)	size
<b>normal</b>				
100,000	0.1908	1.9145	0.1974	16 MB
1,000,000	0.1934	3.2928	0.2047	155 MB
<b>during checkpoint</b>				
100,000	0.3166	4.6740	0.2563	16 MB
1,000,000	0.4155	15.4259	0.2251	155 MB

Table I: Duration of a file create during normal operations and while checkpointing.  $\sigma$  is the standard deviation,  $P_{99}$  the maximum among the fastest 99% of all requests, size is the on-disk index size.

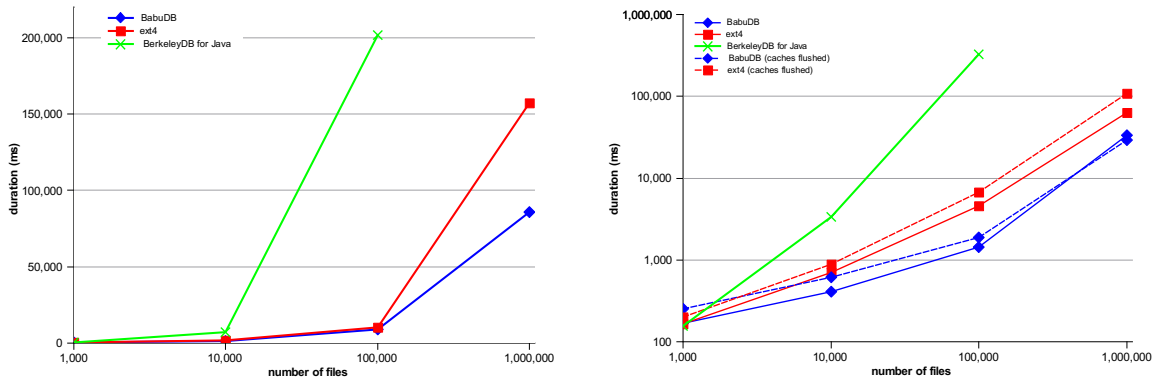


Figure 4: **left:** duration of file creates **right:** duration of ls operation (with and without caches)

36,323 seconds to complete. We analyzed a partial replay of the trace with the Netbeans profiler. The profiling revealed that BerkeleyDB for Java still does subtree locking despite locking being disabled in the database configuration. However, the locking accounted only for approx. 50% of the time while the rest of the time was spent on the B-tree search/insert/remove operations and the persistent operations log.

#### E. Dovecot IMAP Server

For this experiment, we recorded the metadata operations executed by a Dovecot mail server in the mailbox directory. We configured Dovecot to be compatible with networked file systems (e.g. no mmap,fcntl locking) and to use the maildir format. To generate load, we ran the `imaptest` stress test provided by the Dovecot developers which simulated five concurrent clients for 10 minutes. This resulted in a trace with approximately 2 million metadata operations (51% `getattr`, 48% `open`, 1% others). Again, BabuDB and ext4 showed a similar performance with 367 and 385 seconds, respectively. BerkeleyDB for Java was an order of magnitude slower with 5,912 seconds.

#### F. Summary

The experiments show that BabuDB is a good fit as a metadata storage backend. It has a similar performance as ext4 but scales better with large numbers of files. At the same time, BabuDB offers asynchronous snapshots which are not available in the other backends. We showed that these snapshots have a low impact on the latency of file creates.

### V. RELATED WORK

Log-structured merge trees [9] have been proposed as an alternative to B-trees for databases with a high rate of record inserts. Unlike BabuDB indices, LSM-trees consist of a single in-memory tree and multiple persistent on-disk indices. The idea of applying concepts of LSM-trees to

large-scale databases has been made popular by Google’s Bigtable [15] architecture. The BabuDB on-disk index layout resembles the design of an SSTable that persistently stores data of a Bigtable. With Bigtable, Google has demonstrated that these concepts are suitable for very large databases in the range of hundreds of terabytes.

Many database systems such as BerkeleyDB, MySQL and modern file systems like btrfs use B-trees to organize their data. A B-tree is a data structure for storing large, ordered indices [16]. Its design is optimized for data which exceeds the size of the main memory and has to be stored on a hard disk. Several variations of the original B-tree [17] exist, for example the B+tree which stores data only in the leaf nodes. Multi-version B-trees [18], sometimes referred to as copy-on-write B-trees, allow lookups and range queries on any version of the tree. These can be used to implement asynchronous checkpoints and snapshots with B-trees. The ext4 file system uses a variant of the B-tree with a very high fan-out called HTree [19], [20], which is optimized for file system directories.

Database techniques have been applied to the problem of metadata management in distributed file systems in various ways. Lustre [8], the most prominent open-source cluster file system, stores its metadata as inodes in a local ext4 file system. Advanced features like snapshots and clustering of metadata servers have been announced but not yet released.

Panasas ActiveScale [21] offers a commercially distributed alternative to Lustre. It relies on *metadata managers* that mediate client access to file system metadata. The metadata itself is attached to the file content and persistently stored across the file servers, which use a proprietary local file system purpose-built for object storage. Panasas offers copy-on-write-based snapshot support at object and partition level.

The CEPH parallel file system [22] stores its metadata in an in-memory directory tree. Operations are persistently logged for recovery. Such an architecture does not require persistent checkpoints but only supports databases that fit in

the main memory.

A different approach to store file system metadata is to use a dedicated database, as e.g. done by PVFS2 [23]. PVFS2 metadata is stored in BerkeleyDB, a general-purpose key-value store [4]. In connection with PVFS, a variety of alternative metadata distribution schemes have been examined [24], [25]. Most of them, however, require additional mechanisms to ensure atomicity of operations that affect multiple storage servers. With our BabuDB-based approach, we circumvent such problems by offering the possibility of atomic insertions of multiple records.

## VI. CONCLUSION

We have presented BabuDB, a database that is based on concepts of log-structured merge (LSM) trees, and described its application to metadata storage for distributed file systems. BabuDB only consists of a database log, an on-disk search tree and a checkpointing mechanism, but is well suited to the problem of storing file system metadata. We have argued that BabuDB indices can handle short-lived files, variable-length metadata such as extended attributes and ACLs, and typical file system operations such as `rename` and `readdir` very well. In addition, BabuDB supports asynchronous database snapshots that provide the basis for file system snapshots, which would otherwise require complex multi-version B-trees.

Our experiments have demonstrated that BabuDB scales better and is up to twice as fast as `ext4` when used as a storage backend for our metadata server, and that asynchronously writing a checkpoint to disk has little impact on the metadata server's performance.

As future work, approaches will be studied to decrease the storage overhead of persistent snapshots. When a snapshot is written to disk, a `diff` to a previously taken snapshot could be stored on disk instead of a complete representation of the whole database state. This could help to reduce the need for storage space if a considerable number of snapshots is created.

BabuDB and XtreamFS are open source and available for download at [www.xtreamfs.org](http://www.xtreamfs.org).

## ACKNOWLEDGMENTS

This work was partially supported by the EU IST program as part of the XtreamOS project (contract FP6-033576) and SELFMAN.

## REFERENCES

- [1] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: The future building block for storage systems," in *Local to Global Data Interoperability - Challenges and Technologies*. Washington, DC, USA: IEEE Computer Society, 2005.
- [2] M. Mesnier, G. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 8, pp. 84–90, 2003.
- [3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
- [4] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, April 2009.
- [5] M. K. McKusick and S. Quinlan, "GFS: Evolution on fast-forward," *Queue*, vol. 7, no. 7, pp. 10–20, 2009.
- [6] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004.
- [7] D. R. Jacob, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000, pp. 41–54.
- [8] Cluster File Systems Inc., "Lustre: A scalable, high-performance file system," 2002.
- [9] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.
- [10] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A trace-driven analysis of the UNIX 4.2BSD file system," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-85-230, Apr 1985. [Online]. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/1985/5199.html>
- [11] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt, "How to barter bits for chronons: compression and bandwidth trade offs for database scans," in *SIGMOD Conference*, C. Y. Chan, B. C. Ooi, and A. Zhou, Eds. ACM, 2007, pp. 389–400.
- [12] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, "Super-scalar RAM-CPU cache compression," in *ICDE*, L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, Eds. IEEE Computer Society, 2006, p. 59.
- [13] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "XtreamFS: a case for object-based storage in Grid data management," in *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB 2007*, 2007.
- [14] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *VLDB*, P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, Eds. Morgan Kaufmann, 2001, pp. 169–180.
- [15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 205–218.

- [16] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indices," *Acta Inf.*, vol. 1, pp. 173–189, 1972.
- [17] D. Comer, "Ubiquitous B-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979.
- [18] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion B-tree," *The VLDB Journal*, vol. 5, no. 4, pp. 264–275, 1996.
- [19] D. Phillips, "A directory index for Ext2," in *Proceedings of the 2001 Annual Linux Showcase and Conference*, 2001, pp. 20–20.
- [20] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium. Ottawa, ON, CA*, 2007.
- [21] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the Panasas parallel file system," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–17.
- [22] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "CEPH: A scalable, high-performance distributed file system," in *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI '06)*, vol. 7. USENIX, November 2006.
- [23] P. H. Carns, W. B. Lingon III, R. B. Ross, and R. Thakur, "PVFS: a parallel file system for linux clusters," in *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*. Berkeley, CA, USA: USENIX Association, 2000, p. 28.
- [24] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan, "Revisiting the metadata architecture of parallel file systems," in *Third Petascale Data Storage Workshop, Supercomputing*, November 2008. [Online]. Available: <http://www.cse.ohio-state.edu/alin/papers/pdsw2008.pdf>
- [25] —, "An OSD-based approach to managing directory operations in parallel file systems," in *IEEE International Conference on Cluster Computing*, September 2008. [Online]. Available: <http://www.cse.ohio-state.edu/alin/papers/cluster2008.pdf>