Filesystems for Streaming Databases

Bradley C. Kuszmaul*
Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory (MIT CSAIL)
bradley@mit.edu

Abstract

Workloads for high-performance streaming databases often contain many writes of small data blocks (for example, of metadata) followed by large subrange queries. Most of today's file systems and databases either cannot provide adequate performance for the write phase, the read phase, or both. The supercomputing technologies group at MIT CSAIL has been investigating cache-aware and cache-oblivious data structures for disk-resident streaming data. We investigated the cache-aware buffered repository tree (BRT). A BRT with a block size of B can theoretically perform a write in time $O((\log_B n)/\sqrt{B})$, as compared to B-tree's $O(\log_B N)$, and can perform reads only a constant factor slower than the B-tree. We implemented a prototype of a cache-aware streaming B-tree. For 1-megabyte blocks the streaming B-tree achieves a 230-fold speedup for random insertions, at a cost of slowing down serial insertions by factor of 6. Preliminary measurements on the SSCA#3 IO benchmark show a 1.4-fold speedup using stream B-trees instead of standard B-trees. We have also designed a cache-oblivious data structure called the cache-oblivious lookahead tree, which should be able to achieve similar bounds without requiring us to tune for the block size.

1. Introduction

The time to perform input/output can dominate the performance of many embedded applications. The performance of many embedded applications is limited by how fast they can perform large numbers of operations on small amounts of data, or *microdata*. These applications would like to store the microdata in *microfiles* that contain only a few bytes of information, indexing the microdata with a file name that describes the site on the genome. For example, in the area of synthetic-aperture radar (SAR) knowledge formation, an application may write many microfiles, each containing a small image, where the file name of the image is an encoding of its coordinates. Later processing steps perform range queries on the file names, extracting all the images that relate to a particular geographic region. These are but two applications that are hampered by the inability of today's storage systems to handle large volumes of microdata efficiently. This research project is aimed at understanding and developing the technology of *microdata storage systems*, which perform as well for microfiles as they do for macrofiles.

Traditional file systems overcome the large cost of accessing disk by organizing data into blocks. To gain performance, the block size is tuned to amortize the disk-head positioning time over the data transfer. For microfile operations, performance is typically dominated by the cost of updating a file's *metadata*, such as its file name or file-structure index. Even if microfiles are operated on in batches, existing file systems cannot amortize the disk-head positioning time over the actual operations to be performed on the data. Thus, high-end applications that perform a large volume of microdata operations use only a tiny fraction of the available disk bandwidth.

2. B-trees perform poorly on microdata

The performance of a disk-storage system depends on three factors: the hardware, the storage structure, and the workload. For a typical hardware configuration implemented with a traditional B-tree storage structure, let us examine the performance of a microdata-intensive workload. In this analysis, we shall make several assumptions favorable to the traditional storage structure, because we shall use it as a "straw man" to compare with our proposed microdata storage structure.

For hardware, suppose that our disk-storage system is configured with 10 high-performance SCSI 150GB disk drives rotating at $15,000$ RPM, yielding a 2ms average rotational latency for each drive. The average seek time of each disk is 4ms, and its minimum track-to-track seek time is 0.6ms. These parameters yield a 6ms average head-positioning latency and a 2.6ms track-to-track positioning latency. Each drive can sustain data transfer rates of about 90MB/s and contains on the order of $100,000$ tracks. Since we have 10 drives, the peak sustainable bandwidth of the storage system is about 900MB/s.

Let us assume that the traditional disk-storage structure is a file structure employing a B-tree variant [3, 6, 7, 9, 13, 14], as opposed to a log-structured file system [10], using $4,096$-byte blocks. Let us also assume, to give this file system the benefit, that metadata and data are written to nearby locations on the disk, and thus we only need to pay 1 disk seek for both.[1] Our analysis exploits an unfortunate property of B-trees: they age over time, leading the blocks to become randomly distributed across disk [12, 14]. (Journaling file systems age poorly too [5].) Even when the file system attempts to cluster directories together [7], many disk blocks can end up placed far from the their parents on disk.

Suppose that our workload is a high-end microdata-intensive application, such as a synthetic-aperture radar (SAR) knowledge formation, consisting of two phases. In Phase I, the application serially might write many 100-byte microfiles to different directories in the file system. Each write also updates about 100 bytes of metadata. Phase II might consist of a series of *range queries*, each of which reads many files that are lexicographically adjacent to each other in the file system.

Let us now examine how well this traditional storage system performs on this microdata-intensive workload.

For Phase I, the different files likely reside on different tracks, because there are so many tracks. In order not to spend a full seek on each write, let us assume that the application employs a huge number of threads to perform the writes and that disk-head movement is scheduled optimally. Under these favorable assumptions, the application suffers only the 2.6ms track-to-track positioning latency per write, rather than the full 6ms average head-positioning latency. Since each write is only 200 bytes, microdata and metadata combined, the bandwidth achieved to each disk drive is 77KB/s, or 770KB/s for the 10-disk system. Despite these heroic measures and optimistic assumptions, the traditional file system achieves only about 0.09% of the peak disk bandwidth on Phase I.

For Phase II, let us assume that the traditional file system manages to pack each of the $4,096$-byte blocks with the metadata for about 40 lexicographically adjacent files. After performing a random seek (6ms) to locate the start of the range, the $4,096$ bytes of metadata can be read (0.05ms), and then the process repeats for the next block in the range, because the file system cannot guarantee proximity between adjacent blocks in the range. Thus, the system achieves 680KB/s of bandwidth, or about 0.07% of the 10 disks'

*This work was supported in part by the Singapore-MIT Alliance and by NSF Grants CNS-0305606 and ACI-0324974

[1] The situation would be even worse if the metadata is not placed next to the data. Moreover, some file systems require several disk writes to create a single file [10].

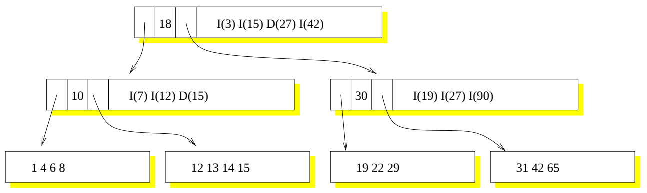

Figure 1: A streaming B-tree in which each node maintains a 1-block buffer. Each internal node contains a set of keys (e.g., 18 on the root), as well as a list of insertion and deletion commands (shown as I and D.) A BRB-tree operates on key-value pairs, but here we show only the keys.

collective bandwidth. Making the B-tree blocks larger can help: 16KB blocks would achieve 2.7MB/s, but to achieve half the bandwidth of just a single disk would require blocks of size 540KB. Large blocks can introduce many other problems for B-trees, however, such as internal fragmentation, where large amounts of memory are wasted when only a few bytes are needed from a block. Moreover, manipulating data within large blocks is cumbersome.

If instead of using a B-tree file system, we were to use a log-structured file system [10], we would still have performance problems for our microdata application. During Phase I, a log-structured file system wold run at full disk bandwidth. During Phase II, however, the range queries would still run slowly, since the microfiles would be distributed on disk in the order they were written, rather than being grouped locally according to their names. A log-structured file system can further confound programmers by giving different performance numbers every time a range query runs, since these file systems require a “cleaning” task that moves data around on disk in unpredictable ways.

Thus, log-structured file systems can make Phase I run fast, and B-tree file systems have the potential to make Phase II run fast (at least with large blocks). Is there a storage structure that can give good performance on both phases?

3. Streaming B-trees

We propose to improve the performance of the traditional B-tree storage structure by exploiting a new storage structure, called a *streaming B-tree*, for short, which we have developed in collaboration with Michael Bender of SUNY at Stony Brook and Martin Farach-Colton of Rutgers. Our data structures draw inspiration from a data structure called a buffered repository tree which was briefly described in [1, 2], and has not, to our knowledge, previously been implemented.

The layout of one kind of streaming B-tree is shown in Figure 1, where $r = 2$. The data structure is a B-tree in which each node also maintains a 1-block *buffer*. When a key-value pair is inserted, instead of traversing the tree to store the value at the leaf, as in a normal B-tree, the pair is simply inserted into the buffer of the root. When a node's buffer fills up, the buffered data is properly dispersed to the appropriate children of the node. When a block is transfered from disk to main memory, an average of $1/r$ of a block's worth of data is productively modified. Eventually, the buffered data migrates down to the leaves.

To perform a query on a BRB-tree, first examine the buffer at the root. If the value is not found, then search in the appropriate child recursively. In the example shown in Figure 1, if we were to search for key 42, we would find an “insert 42” command in the root, for some value v, indicating that key 42 is in the tree. If we were to search for key 14, we would find no mention of 14 in the root or in the appropriate child of the root (in this case the left child), but we would find it eventually at a leaf.

4. Performance

In Spring 2006, Jelani Nelson implemented a buffered repository tree using 4KiB blocks, and achieved a a 17-fold speedup for

	B-tree		Streaming B-tree	
	insertions	bandwidth	insertions	bandwidth
Serial I/O	140,000 /s	11.0 MB/s	22100 /s	1.7 MB/s
Random I/O	180 /s	0.014 MB/s	42200 /s	3.2 MB/s

Figure 2: Performance of B-trees vs. streaming B-trees on serial and random I/O.

insertions of 16-byte key-value pairs, with a 3-fold slowdown for random searches [8].

We recently implemented a new data structure that employs 1MiB blocks. We compared our implementation to the Berkeley Database (BDB) [11]. We configured BDB to use no transactions or locking, and to use a 512MiB cache. We first performed 10^7 sequential insertions (that is, serial I/O), comprising 783MB of data, then measured the performance of insertions of randomly chosen keys (random I/O). Figure 2 shows the performance. The streaming B-tree achieves over 230-fold speedup for random I/O, compared to the B-tree. For serial I/O the streaming B-tree performs about 6 times slower than the traditional B-trees. Our streaming B-tree has not been optimized yet, and we believe that we can improve both the serial and random performance.

We also built a version of the SSCA#3 I/O Benchmark [4] using the data structures. We achieved 5.6 MB/s with B-trees, 8.2 MB/s with our streaming B-tree. Using ordinary file I/O we achieve 15.3MB/s.

5. References

[1] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 546–554, 2003.

[2] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 859–860, 2000.

[3] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[4] HPCS challenge benchmarks scalable synthetic compact application — SSCA#3: Sensor processing and knowledge formation. MIT Lincoln Laboratory, `http://www.highproductivity.org/SSCABmks.htm`, 2005.

[5] C. Loizides. Journaling filesystem fragmentation project. `http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/agetest.html`, 2004.

[6] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[7] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *USENIX Winter 1991 Tech. Conf.*, pp. 33–43, Dallas, TX, USA, 1991.

[8] J. Nelson. External-memory search trees with fast insertions. Master's thesis, MIT EECS, 2006.

[9] H. T. Reiser. Reiser file system white paper. `http://www.namesys.com`, 2002.

[10] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992. , Volume 10, Number 1, February.

[11] Sleepycat Software. The Berkeley Database. `http://www.sleepycat.com`, 2005.

[12] K. A. Smith and M. I. Seltzer. File system aging - increasing the relevance of file system benchmarks. In *Proc. 1997 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pp. 203–213, Seattle, Washington, 1997.

[13] A. Sweeny, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In *Proceedings of the 1996 USENIX Technical Conference*, pp. 1–14, San Diego, CA, Jan. 1996.

[14] K. Thompson. Unix implementation. *Bell System Technical Journal*, 57(6), July–Aug. 1978.