# Final Report: Efficient Databases for MPC Microdata

Michael A. Bender    Martín Farach-Colton    Bradley C. Kuszmaul

August 16, 2012

## 1 Project Overview

The purpose of this grant was to develop the theory and practice of high-performance databases for massive streamed datasets. Over the last three years, we have developed *fast indexing* technology, that is, technology for rapidly ingesting data and storing that data so that it can be efficiently queried and analyzed.

Streamed datasets include:

**Simulation querying:** Many high-end simulations produce vast quantities of data, and that data needs to be stored and queried.

**Astronomical imaging:** The entire sky may be imaged on a regular basis and later the images from a particular region in the picture are compared, for example, to monitor supernovae.

**Reconnaissance monitoring:** A set of regions may be imaged daily, and later some regions are compared over time, for example, to monitor enemy movement.

**Network monitoring:** A communications network is instrumented, with each router providing information about network flows and traffic, and then information about traffic from a particular source is examined.

For data-analysis applications involving high-bandwidth data streams, HPC practitioners typically must make a difficult choice between employing a database or a file system for storing their persistent data. Databases often perform operations slowly, partially because of the extra weight from the SLQ from end and from the transactional support, leading programmers to choose file systems instead. On the other hand, file systems provide poor data-consistency guarantees, which are increasingly important for large data sets.

However, both file systems and databases typically provide poor performance when they are *indexing microdata*. *Microdata* means small objects, much smaller than the natural block size of storage devices. *Indexing* means ingesting objects and ordering them on disk so that they can be queried efficiently. For example, one can ingest mouse-click data roughly in temporal order, but then index the clicks first ordered by user and website location and by then time; then a query to find all mouse clicks on a particularly website from users, say, in Long Island is efficient. As another example, one can ingest astronomical data in rough temporal order but index the data to optimize queries about differences in particular starts or regions of sky over time.

During this project we developed the technology so that high-bandwidth data streams can be indexed and queried efficiently. Our technology has been proven to work data sets composed of tens of billions of rows when the data streams arrives at over 40,000 rows per second. We achieved these numbers even on a single disk driven by two cores. Our work comprised (1) new write-optimized data structures with better asymptotic complexity than traditional structures, (2) implementation, and (3) benchmarking. We furthermore developed a prototype of TokuFS, a middleware layer that can handle microdata I/O packaged up in an MPI-IO abstraction.

### Massively Parallel Streaming B-trees

During this grant we built a parallel and concurrent *streaming B-tree* or *Fractal Tree*. We developed the technology to transform the streaming B-tree into a highly parallel, concurrent, scalable structure of interest to users of high-end clusters. We showed how such a data structure can be used to build a high-performance database capable of supporting data sets containing tens or hundreds of billions of pieces of microdata in which, at each node of a cluster, tens or hundreds of thousands of new pieces of data arrive per second.

The *streaming B-tree* is a write-optimized alternative to the B-tree that was developed by the PIs Bender, Farach-Colton, and Kuszmaul in prior work [4]. The asymptotics of the streaming B-tree outstrip those of the B-tree: a streaming B-tree incurs $O((\log N)/B^{1-\varepsilon})$ or even $O((\log N)/B)$ disk I/Os per insert, whereas a B-tree incurs $O(\log_B N)$ disk I/Os in the worst-case. These asymptotics translate to two order-of-magnitude improvements in insertion performance in a streaming B-tree compared to a traditional B-tree, with no asymptotic loss in point-query performance. The streaming B-tree as several nice additional features, for example, it is "cache-oblivious." For details about what this means, see [2,4,8].

Before this project began, the streaming B-tree was fundamentally a serial data structures, not yet ready to be used for HPC applications. It did not scale with the number of cores, clients, or disks. It did not support recovery from failures, and did not have safety properties that one should expect for secure storage of important data.

## TokuDB

Many of the innovations from this project have been incorporated into TokuDB [16], the transaction-safe streaming B-tree being developed and commercialized at Tokutek. TokuDB can perform over 40,000–100,000 high-entropy insertions/deletions per second on a single disk.

In contrast Oracle Berkeley DB [9] or InnoDB [14] are transaction-safe B-trees, which can perform only several hundred insertions/deletions per second on a single disk when data is random and data sets are much larger than main memory.

Currently TokuDB can scale to hundreds of clients, up to 50 cores, and up to 30 disks. (Our most scalable version is not yet GA.) TokuDB can ingest data sets comprising tens to hundreds of billions of rows without hitting performance cliffs. TokuDB has additional features that we have been able to develop thanks to the parallel and concurrency exposed as part of this project, notably:

- ACID semantics
- Multi-threaded bulk loader
- Online schema changes (hot-column addition, hot indexing)
- Scalable lock management
- Support for hundreds of clients
- Concurrent, multithreaded updates.

TokuDB has a Berkeley DB API, enabling it to be used as a Berkeley DB-style key-value store, although with faster performance. TokuDB also has a "handlerton" layer enabling it to work as a storage engine for MySQL. TokuDB is currently being used in production for such applications as metadata maintenance in a cloud file system, log file analysis, index-rich OLTP, and OLAP.

One should think about TokuDB as a data structure that ingests almost as quickly as if the data were simply logged, but supports range query and point-query performance as efficiently as if the data were stored in a B-tree.

We also developed TokuFS, a middleware layer that uses TokuDB's parallel streaming B-tree technology to provide an MPI-IO layer.

## Overview

In the rest of this report, we describe some of the outcomes from this project. In Section 2 we give a simple serial streaming B-tree. It has impressive asymptotics but significant limitations, including serial bottlenecks, jittery performance, no ACID guarantees, etc. However, as we established, this data structure can be transformed into something of use to HEC applications. In Section 3 we highlight some outcomes from our project. Section 3.1 describes the iiBench Benchmark. Section 3.2 shows how the streaming B-tree performs on this benchmark, both for rotational disks and

SSDs. Section 3.3 describes the search-insert asymmetry inherent in write-optimized structures such as streaming B-trees. This search-insert asymmetry has huge advantages: fast insertions are a currency enabling fast queries. It also includes disadvantages: many insertion operations in traditional B-tree-based storage systems have “hidden” searches (cryptosearches) coupled with insertions, and these cryptosearches may throttle insertion performance. Section 3.4 describes a messaging mechanism we devised (“upsert” mechanism) to eradicate many common cryptosearches. Section 3.5 gives new data structures we developed for answering approximate membership queries. The most well known previous structure is the Bloom filter, which does not perform well when the data structure is stored on an SSD rather than in RAM. Section 3.6 describes how we use the upsert message mechanism from Section 3.4 to develop a fast algorithm for hot column addition. Section 3.7 describes our bulk loader for loading stored data sets very rapidly. Sections 3.8 and 3.9 describe some of the locking and concurrency control strategies we have developed for increasing the parallelism in our streaming B-tree. Section 3.10 highlights one such concurrency control mechanism that we implemented, multi-version concurrency control. Section 3.11 describes some of the I/O optimizations we have done for improving multiclient performance. Section 3.12 describes TokuFS.

## 2 Serial Streaming B-tree

In this section we present a serial streaming B-tree. We explain some of the limitations of the basic structure, which we addressed in the course of our project.

### Basic Streaming B-tree

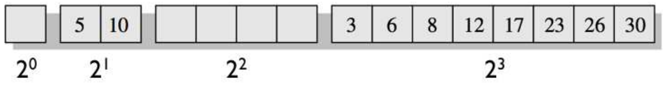


Figure 1: A simplified streaming B-tree.

Figures 1 and 2 show a streaming B-tree comprising arrays of exponentially increasing size embedded in a large array. Within each array, elements are sorted, but there is no set relationship between elements in two separate arrays. Each array is either completely full or completely empty. Whether the array is full or empty depends on the bit representation of $N$. Thus, in Figure 1 there are ten elements. The bit representation of ten is 1010, meaning that exactly the fourth and second arrays are full. A newly inserted value is added into the smallest array in the data structure. If we try to add elements to an array that is already full, we flush the elements in that array into the next larger array. Thus, in the second and third line in Figure 2, element 12 is being inserted, but element 17 is already in position. So 12 and 17 are merged into the second array. In the penultimate line, element 26 is being inserted, but the first three arrays are already full, so all elements are merged into the fourth array.

This data structure can be analyzed as follows. The cost to flush an array of size $X$ is $O(X/B)$ disk transfers, leading to an amortized cost of $O(1/B)$ per element. The maximum number of times that an element can be flushed is $O(\log N)$ leading to a total amortized insert cost of $O((\log N)/B)$. Searching requires a binary search at each level for a total cost of $O(\log(N/B)+\log(N/B)-1+\log(N/B)-2+\ldots+2+1) = O(\log^2(N/B))$. This cost is far better than a table scan, but it is not as good as a B-tree or a real streaming B-tree index.

Figure 3 shows how to achieve a search cost of $O((\log N)/B)$. Each array also includes some elements from the next larger array, and maintains vertical pointers between these elements. The pointers reduce the search cost per level to $O(1)$ for a total search cost of $O(\log N)$. To reduce the search cost to $O(\log_B N)$, use arrays that are exponentially increasing by an $\Omega(1)$ factor, reducing the insertion cost slightly to $O((\log N)/B^{1-\varepsilon})$, for $\varepsilon \ll 1$, which is still close to disk bandwidth.

This streaming B-tree is *cache-oblivious* [8]. By cache-oblivious, we mean that the data structure is *platform-independent* or *memory-hierarchy universal*. Remarkably, the streaming B-tree simultaneously achieves approximately optimal I/O performance for all memory sizes and block sizes even though the data structure is not parameter-

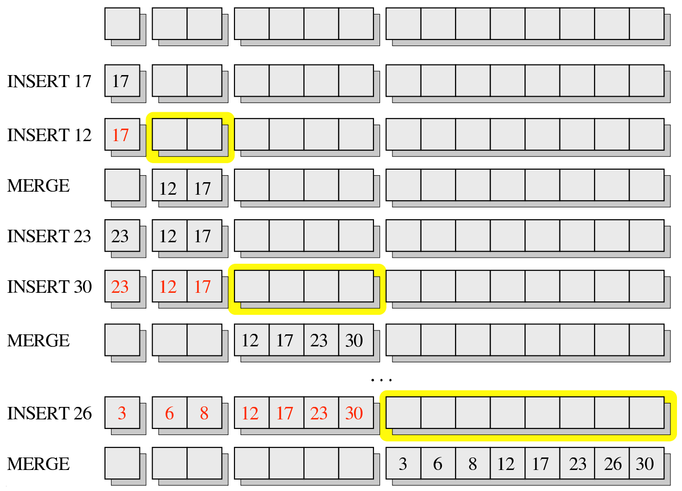


Figure 2: A sequence of states showing how a streaming B-tree structure evolves when there are insertions.

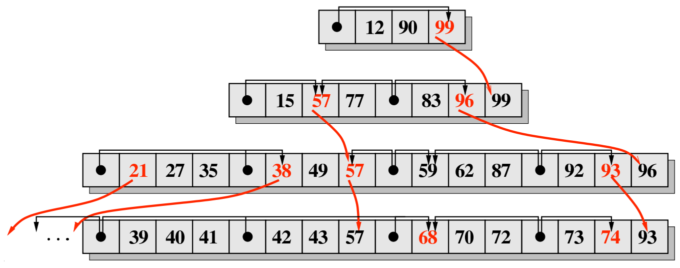


Figure 3: A simplified streaming B-tree yielding a better search performance via redundant storage of elements and pointers between levels.

ized by the value of *B*. The theory of cache-obliviousness was proposed by principal investigator Leiserson [8] and the first cache-oblivious data structures were proposed by principal investigators Bender and Farach-Colton [1].

### Issues With Serial Streaming B-tree

This simplified data structure has limitations (e.g., significant jitter on insertions, same-size key-value pairs, no compression, no ACID, etc).

However, the main issue, addressed in this DOE project, is that the data structure is fundamentally serial, and our work was to build a highly parallel and scalable streaming B-tree.

## 3 Outcomes

In section summarize some of the work that was supported as part of this grant.

### 3.1 Indexed Insertion Benchmark (iiBench)

In addition to the well-recognized TPC benchmarks [7], several benchmarks such as the Star Schema Benchmark [13], sysbench [10], and the DBT series [17], have been proposed and implemented to measure database performance. These benchmarks simulate and measure performance for workloads and data characteristics similar to those measured by the TPC benchmarks.

We developed the iiBench benchmark [11] to measure performance for a use case that occurs commonly in production applications requiring fast indexing of high-bandwidth data. iiBench measures the rate at which a database can insert new rows while maintaining several secondary indexes, a pattern of usage required in always-on applications that:

- require fast query performance and hence require indexes,
- have high data insert rates,
- cannot wait for offline batch processing and hence require the indexes be maintained as data comes in.

We developed iiBench as an open-source benchmark, allowing others to freely use it, extend it, and contribute their changes back. We originally unveiled the benchmark in the context of a challenge issued at the OpenSQL camp. Since then, iiBench has been downloaded and used many times.

Tokutek implemented the original iiBench in C++, and Callaghan ported it to a Python version, and extended it with additional query benchmarking capabilities [11]. He contributed his enhanced version back to the community via Launchpad. We have further enhanced the Python version of iiBench and have contributed our changes back to Callaghan's Launchpad project. Going forward, we will continue to contribute to Callaghan's version, and we do not expect to maintain the C++ version of iiBench.

We have used variations of this benchmark over the past couple of years to stress different aspects of a streaming database: (1) We created a version that measures performance while at the same time adding indexes. This measures the effectiveness of hot indexing. (2) We improved the performance of iiBench so that the test overhead was not impacting measurement results. (3) We created a stripped down version to measure update performance. (4) We added the ability to create Zipfian distributions of all of elements that were queried. This last did not yield substantially different results than random queries. However, we believe that in the future, with improved caching strategies in TokuDB, it might.

## 3.2 Streaming B-tree Performance

In this subsection we present some performance graphs for TokuDB. Figure 4 shows performance graphs for the iiBench benchmark [11]. These measurements were made by Percona [15], a MySQL performance consulting firm (and one of the foremost authorities on MySQL performance). We found that we could not get the same performance out of the InnoDB B-tree that Percona could, and we wanted to measure InnoDB's performance when properly configured. TokuDB required no configuration.

The iiBench benchmark trickle-loads a billion rows while maintaining three high-entropy multicolumn secondary indexes. Figure 4(a) shows insertions only. For the last million rows InnoDB inserts 876 rows/s, whereas TokuDB inserts 16,507 rows/s (19x faster).

In the deletion benchmark, first the database is filled using only insertions. Once the database is has 250,000 rows in it, then the benchmark alternates between insertions and deletions, acting as a FIFO. For InnoDB, the terminal insertion rate is 204 rows/s, and for TokuDB it is 6,496 rows/s (32x faster).

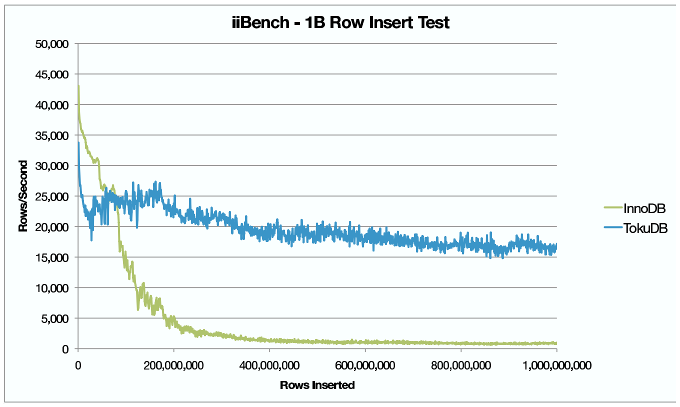
For this study the system ran Linux CentOS 5.1 on a two Socket, Quad Core, Xeon 3.16 GHz X5460, 16GB Main Memory, and a six-disk RAID 0 system comprising 146GB, 10,000 RPM SAS Drives.

TokuDB is "only" 19x and 32x faster because it is CPU bound on six disks (or even one disk). Thus, the insertion speed on a single disk is nearly the same as that reported in the graph, and would increase with the number of disks once there is better multithreading. We report these graphs on a RAID is because this is more indicative of the speedup that a user would see on a typical setup.

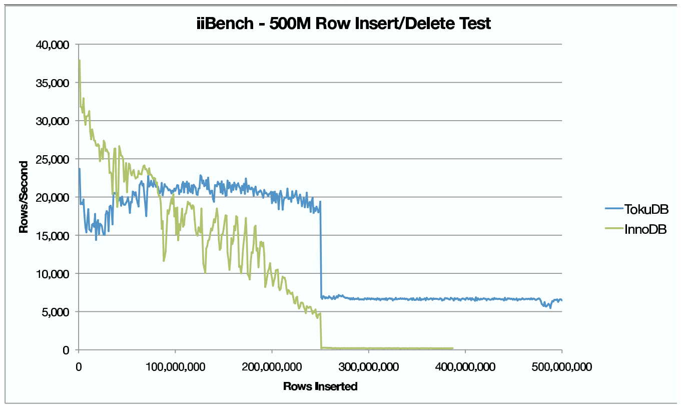
But the promise with multithreading is over 100x. We have a proposed design that we believe will meet this promise, but it has not yet been implemented.

Another reason why TokudB is "only" 19x and 32x faster is that InnoDB employs the in-memory insert buffer, and therefore is already partially write-optimized. See Figure 6 for a comparison when there is no insert-buffer.

Figure 5 shows the performance of TokuDB as measured by Percona on SSDs. In this case, the it is an iiBench insertion workload on a RAID10 (with magnetic disks), on an Intel 32GB X25-E SSD, and on a FusionIO 160GB SSD. On the fastest SSD, InnoDB was inserting 2,287 rows/s, and TokuDB was inserting 17,382 rows/s (7.6x faster).



(a)



(b)

Figure 4: The iiBench benchmark. (a) Insertion performance. (b) Deletion performance. In both graphs, the horizontal axis is the cumulative number of insertions. The vertical axis is the rate of insertions for the most recent million insertions. The blue line is a streaming B-tree, and the green line is an InnoDB B-tree. The discontinuity in (b) is where insertions start, slowing down the insertion rate for TokuDB by about a factor of 2.5, and InnoDB by several orders of magnitude.

Even though TokuDB is not explicitly optimized for SSDs, TokUDB is an order of magnitude faster on a single disk than InnoDB is on a SSD. At least an additional order-of-magnitude improvement remains on the table because TokuDB does not employ multithreading.

## 3.3 Cryptosearches

A streaming B-tree performs inserts/deletes orders of magnitude faster than it performs searches. We call this disparity the *search-insert asymmetry*. Some of our work focused on how to take advantage of this asymmetry. Some of our work focused on how to overcome the obstacles that this asymmetry introduces.

*Search-insert asymmetry is an obstacle and an opportunity.* Although the streaming B-tree is a drop-in replacement for the B-tree, simply replacing a B-tree with a streaming B-tree in a database or file system may not translate to performance improvements. In a typical storage system, there are many implicit, or hidden, searches (which we call *cryptosearches*) associated with insertions or deletions. For example, a database may return an error when a row is inserted with a duplicate key. In this case, to insert a row, the database must search to determine whether the key previously exists. That search is not explicitly visible to the end user, who simply inserted data into the database. Similarly, to create a new file in a file system, the file system first checks whether a file with that name/path already exists, incurring an implicit search.

These searches are innocuous in a B-tree because searches are cheaper than insertions. They kill the performance advantage in streaming B-trees, however, because searches are far more expensive than insertions. Thus, instead of

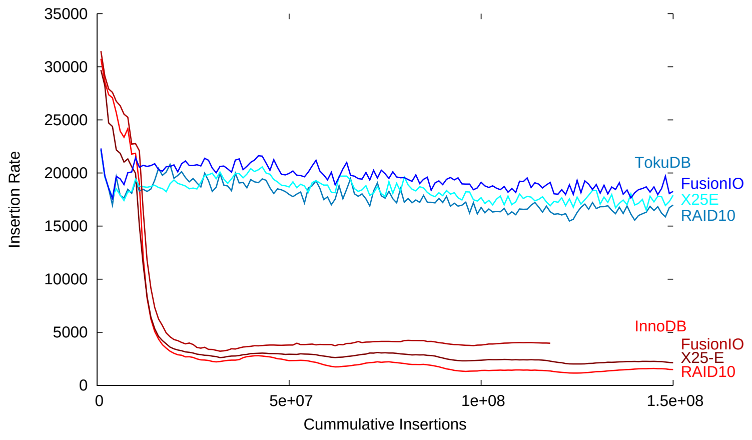


Figure 5: SSD performance of TokuDB. The lower curves are InnoDB, from bottom to top, on a RAID10, on an Intel X25-E SSD, and on a 160GB FusionIO SSD. The upper curves are TokuDB, also in the same order, from bottom to top.

40,000 inserts per second per disk into the database or 40,000 file creates per second in the file system, there will be only several hundred. Cryptosearches can slow the streaming B-tree down by two orders of magnitude, so that it is only as fast as an ordinary B-tree.

Search-insert asymmetry can to solve OHPC problems, however. Since insertions run fast, rich indexes can be maintained, making queries run faster. Insertions are a kind of currency that can be used to speed up queries.

## 3.4 Upsert Mechanism: How to Avoid Cryptosearches

There are many cases where, at first glance, a search (and therefore an expensive disk seek) is required to satisfy a query. With our "upsert messaging algorithm," many of these cases can be handled without a search. As a result, TokuDB's upsert algorithm enables a wide array of applications to run fast by avoiding searches, replacing them with powerful upsert messages.

Here is an example. Consider performing an "insert on duplicate key update" on a table with a random key and a value that is the sum of the values with the same insertion key. A B-tree based system would perform the following steps:

(1) Search for the row with the key.

(2) If the row exists, set the new value to be the old value plus the new value.

(3) Otherwise, put a new row into the tree storing the new value.

Here is an overview of how upsert works and why the search in step (1) is avoided. The user defines a callback function encoded in a message in the data structure. That is, rather than inserting elements into the data structure, we insert messages with callaback functions. The message is inserted in the root, and the callback function is run once the message makes its way to the leaves of the tree. By the time the message hits the leaves of the tree, the callback function can determine whether the row containing the key exists (the new value is set the new value to be the old value plus the new value) or not (the row stores the new value).

This upsert implementation has the same functionality with no required disk seeks per update. Thus, by sending messages down the tree, many (although not all) cryptosearches can be avoided.

See Figure 6 for a comparison of TokuDB with its upsert mechanism and Berkeley DB with the standard searching mechanism.

TokuDB currently uses the upsert algorithm for fast column addition/deletion by simply placing "broadcast" update messages in the root of the tree; see Section 3.6. This upsert algorithm is also heavily used in our database scaleout work; see Section 3.12.

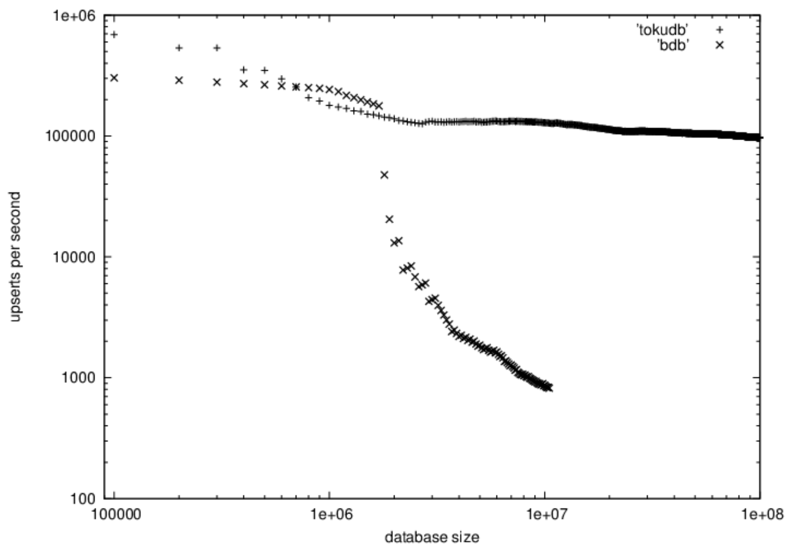


Figure 6: Performance on an upsert benchmark, comparing TokuDB to Berkeley DB. The benchmark is simulating an insert on duplicate key update on a table with a random key and a value that is the sum of the values with the same insertion key. The operation: Search for the row with a given key. if the row exists, set the new value to be the old value plus the new value. Otherwise, just put a new row into the tree. While it looks like this algorithm requires searches, in fact, it can be implemented via our upsert algorithm using only messages. The message algorithm is to put an update message into the tree with the key and value. The update callback, when called, generates a new value for the key equal to the old value and the update value. This message algorithm is useful for a write-optimized data structure such as TokuDB, but does not help a traditional structure such as Berkeley DB. As expected, Berkeley DB performs well while the data set fits in memory, and then the performance crashes. In contrast, TokuDB continues to perform well.

## 3.5 Data Structures for Approximate Membership Queries (When Cryptosearches Cannot be Avoided

Some cryptosearches cannot be avoided. For example, if uniqueness constraints have to be maintained, the upsert mechanism that we developed does not help. We have published data structures to deal with this critical case [3], but have not yet implemented something into TokuDB. We propose to do so in future work.

The trick is to use data structures that support approximate membership queries (AMQ). The classic example of such a data structure is the Bloom filter [6]. An AMQ data structure supports the following dictionary operations on a set of keys: insert, lookup, and optionally delete. For a key in the set, lookup returns “present.” For a key not in the set, lookup returns “absent” with probability at least $1-\varepsilon$, where $\varepsilon$ is a tunable false-positive rate. There is a tradeoff between $\varepsilon$ and the space consumption.

We developed the Cascade Filter, which scales to SSDs, whereas the Bloom Filter exhibits poor insertion and query performance once it is too big to fit in memory. The Cascade Filter supports over half a million insertions/deletions per second and over 500 lookups per second on a commodity flash-based SSD. Bloom filter capabilities are important for maintaining streamed datasets, e.g., for maintaining data sets with unique-key constraints. It is critical to scale AMQs to larger-than-memory sizes because such a scaling dramatically extends the range of applications for which such data structures can be used. AMQ allow relatively cheap CPU cycles to offset relatively expensive IOs and have been used in many IO-intensive systems.

## 3.6 Hot Column Addition and Deletion

We showed how a streaming B-tree enables an efficient implementation *hot column addition and deletion (HCAD)*. The database columns are modified in parallel with subsequent insertions, deletions, and queries to the database.

Many practitioners have the experience of loading a bunch of data into a table and associated indexes, only to find that adding some columns or removing them would be useful. The command

```
alter table X add column Y int default 0;
```

takes a long time, hours or more, during which time the table is write locked, meaning no insertions/deletions/updates and no queries on the new column until the alter table is done.

Changing the row format in B-tree-based storage engines is a significant project, and is an obstacle in having MySQL scale to large tables. Some commercial databases (e.g., Oracle) do optimize for fast schema changes. But, as we explain, the problem is much cleaner with the messaging/upsert structure of a streaming B-tree (see also Section 3.4).

Using a publicly available air-traffic data set (see, e.g., `http://www.tokutek.com/air-traffic-data-hcad/`) we obtained the following results:

TokuDB:

```
mysql> alter table ontime add column totalTime int default 0;
Query OK, 0 rows affected (3.33 sec)
```

InnoDB:

```
mysql> alter table ontime add column totalTime int default 0;
Query OK, 122225386 rows affected (17 hours 44 min 40.85 sec)
```

That is 19,000 times faster. The downtime of InnoDB is proportional to the size of the database, whereas the downtime for TokuDB 5.0 depends on the time it takes for MySQL to close and reopen a table  a time independent of database size.

One of the important features of streaming B-trees/Fractal Trees is that they replace random I/O with sequential I/O. The way this happens has an impact on how HCAD works.

We can think of all data in a streaming B-tree storage engine as a *message*. There can be messages to insert a row, or update a row, or delete a row. Rather than delivering these messages immediately, the messages are bundled up by common destination, and they progress towards the leaves of the streaming B-tree indexes when there are enough of them to make the disk-head movement worthwhile. Naturally, they are applied in the right order to guarantee that the semantics of storage engine commands are correct.

Even a query can be thought of as a message, except in the case of a query, it has to be delivered instantly, even if that means moving the disk head. The query "sees" all the messages ahead of it, so it gets all the right answers, once again, according to the storage engine/SQL semantics.

An HCAD command generates yet another type of message: it is a broadcast message that needs to be applied to every row. Sticking the message into the streaming B-tree is fast.

The work of changing the rows does not happen when the HCAD message is injected. Rather, the broadcast message makes its way down to the leaves as other messages push it along. In this process, when an HCAD message reaches a row either because other messages push it along or because of a query, the row gets rewritten to include the added column or exclude the delete column, as the case may be. Once the work is done to rewrite a row, the HCAD work is done for that row. The user can choose to have this work done immediately, say by a query that touches all rows, or lazily, as part of the normal operation of the database. Neither case involves downtime, and once the work is done to rewrite a row, the HCAD work is done for that row.

## 3.7 Multithreaded Bulk Loader

We built a multithreaded bulk loader. The bulk loader is used to initialize TokuDB en masse with a pre-existing data set, and also, importantly, to create new indexes when needed by an application.

This project accrued several advantages from the the bulk loader. First, the bulk loaded helped us to support hot index creation, another hot schema change in the same spirit as hot column additiona. Second, bulk loading has been a good test bed for preparing to multi-thread more complicated parts of the the system, as we describe in Section 3.8.

The bulk loader uses a parallel, multi-stage merge sort algorithm to populate a set of streaming B-trees from an unsorted sequence of rows. the source rows can be a CSV file that contains the rows (in the case of the initial load of a table), or another streaming B-tree (in the case of creating a new index on an existing table).

We have incorporated new parallelizing compiler technology, Cilk Arts, into TokuDB. Cilk Arts performs thread-to-core scheduling in a provably optimal way. We expect the use of Cilk Arts to be critical as we improve the multi-threading of the trickle loader and query engine.

We also developed some internal infrastructure to instrument the bulk loader. We found that we needed fine-grain metrics on CPU performance in order to optimize the bulk loader. Further, it was essential to keep such metrics in memory until runs were completed. This is a special challenge for databases, because experiments must run for a long time in order to demonstrate the effects of out-of-memory computation.

Finally, we found that we must not only parallelize the computation, but we must pipeline the phases to keep the cores fully loaded.

In one experiment we found a 2.1x speedup on a 2 core machine, and a 4.2x speedup on an 8 core machine. Our old code was using approximately 2 cores, so this represents a linear speedup.

In another experiment on Amazon, we found a 8.2x speedup on Amazon Web Services c1.large node with 8 cores while loading a table with 256 byte rows. We will be exploring performance on very high-core count machines going forward.

## 3.8 Lock Management

There are two types of locking in a storage engine, such as TokuDB, *concurrency control mechanisms*, which control how different clients and transactions can access shared data, and *lock management*, which control how the underlying threads of the data structure access the shared data/nodes. Here we talk about our work in both these areas.

First we discuss lock management. When this project began, much of the complexity of the streaming B-tree algorithms was protected by a global lock. Our path has been to introduce multithreading gradually by protecting a decreasing fraction of the system with this lock. We first implemented multithreading for limited pieces, such as compression/decompression/deserialization, and then for restricted components, such as the bulk loader. Next we came up with a design for holding the global lock only for some operations, but not for slow operations such as I/O. Currently we have an implementation that does fine grained locking for writes, but still holds a global lock at times for reads. The next stage of the work (currently being designed) is to perform this fine-grained locking for reads. Even with this global lock, we scale to many hundreds clients.

We emphasize that the serial streaming B-tree (see Section 2) has bottlenecks absent in the traditional B-tree. For example, in this serial structure, writes modify the root of the structure. If there are many concurrent threads or clients performing insertions/deletions, then there is contention to modify the root of the tree. In contrast, in a B-tree, most writes do not modify the root so there is no significant contention.

## 3.9 Concurrency Control

Our development strategy for concurrency control has been to incrementally increase the complexity of our concurrency control mechanisms. We began by supporting pessimistic locking. Then we introduced mechanisms such as read committed and read uncommitted. We introduced a deadlock detector for TokuDB. We also introduced full-on Multiversion Concurrency Control.

## 3.10 Multi-version Concurrency Control

Our rich messaging infrastructure allows us to implement transactions. Within our transactional system, we have MVCC, which allows queries without holding any locks on rows read. Holding locks on read rows is problematic because the locks are on ranges and not individual rows. So, if a query wants to find all the rows where the key is between 0 and 100, no other thread may write a key to the database between 0 and 100 while the query runs. This limits concurrency. The larger the ranges, the more impact on concurrency.

With MVCC, where each transaction sees its own "snapshot" of the dictionary, locking is not necessary. Values that exist when a currently live transaction begins are ensured to stay available throughout the lifetime of the transaction. As a result, any value read by a live transaction is guaranteed to be the value that existed when the transaction began. As a result, read transactions need not grab range locks to protect the integrity of the data they read, because the system will know to keep a copy available throughout the lifetime of the read transaction. With the requirement to grab row locks gone, the system becomes more concurrent because there are fewer actions to synchronize.

## 3.11 I/O Optimizations

I/O bottlenecks serialize computations. It is therefore critical that we reduce I/O as much as possible. We added ACID transactions to TokuDB, and we therefore have increased the I/O load through logging of transactions. We have invested considerable work in optimizing the logging to decrease the I/O load.

We have also worked on optimizations around commits on multiple clients. In theory, each commit requires an fsync, and therefore a disk-head movement. When many clients are committing at the same time, it is possible to bundle the commits and fsync them as a group. This is called a *group commit*, and we have found that it improves the transactional rate in such cases by an order of magnitude.

Finally, we have found that in many cases, included when testing TPCC multi-client benchmarks, that performance oscillates. On further examination, we found that the cause of the oscillations is that queries can sometimes request pages that are being compressed and further processed for writing to disk. We have a design document for implementing a block-cloning optimization which should substantially improve the performance of TokuDB on TPCC and related benchmarks.

## 3.12 Scaleout: TokuFS

Since most HPC practitioners use file systems rather than databases for their persistent storage, we worked on demonstrating streaming B-trees to implement what looks like a file system. Although we believe that databases can offer enough performance to meet the needs of the HPC community, we realize that most MPI code is written with a file system in mind. This section first describes TokuFS 0.5, a middleware layer we developed that provides MPI-IO access to a streaming B-tree. Then this section shows that TokuFS can provide performance gains for some workloads on which many other HPC file systems perform badly. N Although TokuFS 0.5 is not a full file system (its limitations will be outlined below), but TokuFS 0.5 does illustrate the kinds of performance gains that can be achieved using a file-system programming interface.

File systems have traditionally been implemented using a B-tree index (or its algorithmic equivalent), which performs well for applications performing large aligned sequential I/O. A B-tree writes data into its final location immediately when any kind of update or write occurs, which means writes with good locality require few disk I/Os. The downside of B-trees appears in the opposite scenario, in which writes have poor locality and thus require many disk I/Os. In this situation one will see slow writes, because B-trees perform badly on microdata, often combined with slow reads, because B-trees are prone to fragmentation. Here we compare two approaches to improving the write performance: PLFS [5], and TokuFS. We found that PLFS can write microdata at high speed but its read performance is slow because of a kind of fragmentation, whereas TokuFS performs well on microdata reads and writes and is fragmentation free.

## PLFS

PLFS [5] was developed to address the microdata write problem. PLFS is a write-optimized middleware layer that sits on a native file system (we used XFS for our experiments). PLFS was designed to facilitate parallel writes to shared files. When *N* processes write to a single logical file, PLFS actually writes the data to *N* different data files (each process writes to its own file). PLFS maintains a log file for each process containing offset-timestamp pairs. All writes to the log file are appended to the end of the file. To write a block using PLFS, a process appends the data to the process's data file, and appends the offset and timestamp to its log file. To read a file, the PLFS library aggregates the *N* log files into a single global offset lookup table, which maps every logical offset to the file and offset where the actual data is stored. Thus, PLFS can be thought of as using a non-clustered B-tree index over offset-data pairs.

Theoretically, non-clustered B-tree indexes have clear sweet spots (and are used for example in some databases, such as MyISAM), but their performance outside of their sweet spots falls off sharply. For example, they perform well when the read order matches the write order, but reading out of order will require random, rather than sequential, access to the log file. And if the index does not fit in memory, then random writes will be induced during data ingestion.

We measured PLFS and found that those theoretical predictions based on data structures considerations match the actual performance of PLFS. In one experiment a single process wrote 20GB of data using 575-byte records, on a machine with 16GB and then read them back. For both sequential and random insertions PLFS inserted them at 48MB/s (the performance was the same since the index was in main memory). For reads, however, the non-clustered nature of the indexes showed a big difference. PLFS required 90 seconds to reorganize the index before any data could be accessed, and then for sequentially-written data, sequential reads achieved 27MB/s. Reads of randomly-written data, however, performed only 0.07MB/s, corresponding to 122 blocks per second. PLFS makes random writes reasonably fast, but subsequent sequential reads become very slow (which is also what happens for other log-structured file systems).

By choosing an append-to-file structure for the data, which is to say a non-clustering B-tree index, the write speed can be increased but at the expense of reads. If we had chosen smaller read/write block sizes, the read performance would have dropped off linearly, since our experiments show that the disk access time is the bottleneck in our tests.

## TokuFS

TokuFS is middleware that implements virtual, user-space file system. The TokuFS library functionality includes the following operations:

- `mount(local_path)`: attempt to mount tokufs at the given local path
- `unmount()`: unmount tokufs
- `open(&toku_fd, tokufs_path)`: open a file in the TokuFS namespace
- `close(toku_fd)`: close an open file
- `read_at(toku_fd, buf, count, offset)`: read at a given offset
- `write_at(toku_fd, buf, count, offset)`: write at a given offset

TokuFS requires an existing underlying file system for storage of its streaming B-tree data structures. The `mount()` and `unmount()` functions allow an application to specify where on the local file system TokuFS should keep its data. In this way, TokuFS is a layered file system, where the top layer is a TokuFS namespace implemented with a streaming B-tree, and the bottom layer is a local namespace implemented in a traditional file system.

TokuFS is built on top of TokuDB, an implementation of key value storage using streaming B-trees. TokuFS views files as a collection of blocks, each numbered incrementally from 0. These block numbers are used as keys into TokuDB, where the value is a block of bytes to hold the data. For simplicity, there exists a single table for each file.

The TokuFS middleware takes those operations, and combined with a variation of ADIO, offers an MPI-IO interface.

We ran the same experiment against TokuFS that we ran against PLFS. Although TokuFS is CPU bound, it ingested data written serially at 35MB/s and read the data at 32MB/s. Randomly written data data was written at 4.8MB/s, and was read at 20MB/s. Since disk utilization was around 50% during writes for TokuFS, we believe that further multicore parallelization could double the rate of insertions.

| | Sequential Writes | Sequential Reads | Random Writes | Random Reads |
|---|---|---|---|---|
| PLFS | 48MB/s | 27MB/s | 48MB/s | 0.07MB/s |
| TokuFS | 35MB/s | 35MB/s | 4.8MB/s | 20 MB/s |

In summary, under small random writes with a single reader and writer, TokuFS was 10x slower on writes than a logging system (with a potential of become 5x slower after further optimization), whereas it was 286x faster on reads.

As expected, if we replace Fractal Trees with Berkeley DB [9], a clustered B-tree implementation, the random write speed drops to well under 0.1MB/s, which is consistent with a random I/O per insertion.

## *N*-to-1 Read/Writes.

A more typical scale-out situation is one in which *N* readers and writes are writing to a single logical file. Since it is not uncommon in MPI applications for there to be more processors than storage nodes, we consider here experiments involving a single disk and 8 read/write threads.

This workload was modeled on an *N*-to-1 parallel checkpoint in MPI-IO. This workload is problematic for parallel file systems because small writes occurring in parallel on a shared file block are serialized due to file system locking mechanisms. Typically, the locking granularity is at the block level, and if blocks are larger than the writes, the number of writes that fit into a block cannot occur in parallel.

One solution is to transform *N* processes writing to one logical file into *N* processes writing to *N* physical files, therefore removing all lock contention, since such workloads typically have no shared regions among writers. PLFS is based on a such a no-sharing, no-locking, no-inadvertent serialization solution.

However, as noted above, even PLFS suffers for reads and writes of microdata. Thus, when PLFS is used on microdata, it is often used in special cases where each process writes out monotonically increasing offsets and reads it back in the same way.

We created an MPI-IO interface for TokuFS using ADIO [12]. The ADIO interface allows storage systems such as TokuFS to be used in MPI-IO applications transparently. All these applications need to do is refer to files with the prefix `tokufs:`, and link then TokuFS library is used to read and write the file data.

MPI-IO is used to coordinate the IO among parallel processes for high performance. Since TokuDB environments and dictionaries are only accessible to one process at a time, the ADIO implementation represents files as the union of each process share in the underlying TokuFS file. The *share* of a certain pid `P` for file `F` is defined as all of the data written to file `F` by processes previously opening the file as pid `P`. Multiple processes writing to the same logical region of file `F` (and thus different shares) produces undefined behavior. When a process wants to open the share for pid `P` of file `F`, the open call looks like:

```
open(&toku_fd, F, P);
```

Subsequent reads will produce data only if the region read was previously written to by some process that opened file `F` as pid `P`. In the simplest case, the reading process is the same as the writing process. To help illustrate how files are represented in the ADIO layer for TokuFS, consider 4 processes writing non overlapping 4 byte segments to form a 16 byte file. The following pseudo code is run by each process in parallel:

```
// who are we in the world of mpi? get a unique process identifer,
// which is somewhere in the range of 0..N-1 for N processes
pid = get_mpi_io_pid();
// open out.file as that pid, then write 10 bytes of stuff.
tokufs_open(&fd, "myfile", pid);
// write out our pid to the file, which is simply 4 bytes.
tokufs_write_at(fd, &pid, 4, pid * 4)
tokufs_close(fd);
```

TokuFS uses a container directory to house the TokuDB environment used by each process taking part in the write of some file. At the top level, the TokuFS mount point looks like:

```
                         /mnt/tokufs
                              .
                              .
      env.0         env.1       env.2         env.3
        .             .           .             .
        .             .           .             .
 data.tokudb    data.tokudb    data.tokudb     data.tokudb
```

The contents of each data.tokudb form sparse files whose union yields the logical ADIO file. Graphically:

```
env.0/data.tokudb

    ['0', null, null, null]

env.1/data.tokudb

    [null, '1', null null]

env.2/data.tokudb

    [null, null, '2', null]

env.3/data.tokudb

    [null, null, null, '3']

= ADIO logical "/mnt/tokufs/myfile"

    ['0', '1', '2', '3']
```

We summarize the performance numbers in the following table:

| | Writes | Reads |
|---|---|---|
| TokuFS | 63.5MB/s | 47.2MB/s |
| PLFS | 66.3MB/s | 24.9MB/s |
| BDBFS | 15.6MB/s | 17.6MB/s |
| XFS | 72.1MB/s | 15.6MB/s |

## 4 Products and Activities

### Software

The TokuFS middleware software is available under an open-source license. TokuFS can be used either with BerkeleyDB or TokUDB for its underlying database library. Contact `martin@tokutek.com` to obtain the software.

## Conferences Attended and Talks Presented

1. Michael A. Bender. “Keynote: From Streaming B-trees to Tokutek: How a Theoretician Learned to be VP of Engineering.” *Symposium on Experimental Algorithms (SEA).* Dortmund, Germany, 2009.
2. Michael A. Bender. “Keynote: Gaps in My Education: Mailboxes, Libraries, and How to Insert into an Array.” *Annual Case Lecture*, St. Louis University, St. Louis, MO, 2009.
3. Michael A. Bender. “Performance of Fractal-Tree Databases.” *IBM Supercomputing Professional Interest Community,* IBM TJ Watson, 2009.
4. Michael A. Bender. “Performance of Fractal-Tree Databases.” *Scalable Approaches to High Performance and High Productivity Computing (ScalPerf)* Bertinoro, Italy, 2009.
5. Michael A. Bender. “Performance of Fractal-Tree Databases.” *Brookhaven National Laboratory*, 2009.
6. Michael A. Bender and Martin Farach-Colton. “Fractal Tree Databases: Data Structures for Fun and Profit.” *Dagstuhl Seminar on Data Structures,* Wadern, Germany, 2010.
7. Michael A. Bender. “Fractal Tree Databases: Concurrency Challenges.” *New Topics in Distributed Algorithms.* Ecole Polytechnique Federale de Lausanne, Switzerland, 2010.
8. Michael A. Bender. “Performance of Fractal Tree Databases.” St. Louis University. St. Louis, MO, 2010.
9. Michael A. Bender. “Performance Guarantees for B-trees with Different-Sized Atomic Keys.” *Principles of Database Systems (PODS)*, Indianapolis, IN, 2010.
10. Michael A. Bender. “Multidimensional and String Indexes for Streaming Data.” High End Computing File Systems and I/O Workshop (HEC FSIO), Arlington, VA, 2010.
11. Michael A. Bender. “Scheduling DAGs on Asynchronous Processors.” CNRS Workshop on New Challenges in Scheduling Theory, Fréjus, France, 2010.
12. Michael A. Bender. “How to Index Massive Data Sets Quickly.” Hofstra University, Hempstead, NY, 2010.
13. Michael A. Bender. “How to Index Massive Data Sets Quickly.” Morrelly Homeland Security Center, Bethpage, NY, 2011.
14. Michael A. Bender. “How Fast Indexing Makes Databases Greener.” Sustainable Energy-Efficient Data Management (SEEDM), Arlington, VA, 2011.
15. Michael A. Bender. “Don’t Thrash: How to Cache Your Hash on Flash.” Bertinoro Workshop on Algorithms and Data Structures (ADS), Bertinoro, Italy, June 2011. Workshop talk.
16. Michael A. Bender. “Better Metadata Management Through Data Structures.” High End Computing File Systems and I/O Workshop (HEC FSIO), Arlington, VA, 2011.
17. Bradley C. Kuszmaul. “Covering Indexes: Orders-of-Magnitude Improvements.” *Percona Performance Conference* at the MySQL User Conference, Santa Clara, CA, April 22, 2009.
18. Bradley C. Kuszmaul. Brief announcement: TeraByte TokuSampleSort sorts 1TB in 197s. *SPAA 2009*, Calgary, Alberta, Canada, August 8-12, 2010. p.127-129
19. Bradley C. Kuszmaul. Obliviousness for High Performance. *ScalPerf ’09*, Bertinoro, Italy, September 20-25, 2010.
20. Bradley C. Kuszmaul. What is a Performance Model for SSDs?. *High Perforamnce Transaction Systems (HPTS)*, Asilomar, CA, October 25-28, 2009.

21. Bradley C. Kuszmaul. An Open Storage Engine API. *OpenSQL Camp 2009*, Portland, OR, November 14-15, 2009.

22. Bradley C. Kuszmaul. How Fractal Trees Work. *OpenSQL Camp 2009*, Portland, OR, November 14-15, 2009.

23. Bradley C. Kuszmaul. How Fractal Trees Work. *MySQL User Conference*, Santa Clara, CA, April 2010.

24. Bradley C. Kuszmaul. Lightning Talk: a Performance Model for SSDs. *MySQL User Conference*, Santa Clara, CA, April 2010.

Note: The travel costs for attending these conferences was *not* paid out of DOE-provided funds.

## Publications

1. K. Agrawal, M. A. Bender, and J. T. Fineman. The Worst Page-Replacement Policy. *Theory of Computing Systems*, Special Issue on *FUN '07*, 44(2): 175-185, 2009.

2. E. M. Arkin, M. A. Bender, J. S. B. Mitchell, and V. Polishchuk. The Snowblower Problem. *Computational Geometry*, Volume 44(8): 370-384, 2011.

3. M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz. The Cost of Cache-Oblivious Searching. *Algorithmica*, 2011. In press.

4. M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model. *Theory of Computing Systems*, Special Issue on *SPAA '07*, 47(4): 934-962, 2010.

5. M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011.

6. M. A. Bender, S. P. Fekete, T. Kamphans, and N. Schweer. Maintaining Arrays of Contiguous Objects. *Proceedings of the 17th International Symposium on the Fundamentals of Computation Theory (FCT)*, LLNCS Volume 6599, pages 14–25, 2009.

7. M. A. Bender, J. T. Fineman, and S. Gilbert. A New Approach to Incremental Topological Ordering. *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1108–1115, 2009.

8. M. A. Bender and S. Gilbert. Mutual Exclusion with $O(\log^2 \log n)$ Amortized Work. *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2011. To appear.

9. M. A. Bender, H. Hu, and B. C. Kuszmaul. Performance Guarantees for B-trees with Different-Sized Atomic Keys. *Proceedings of the 29th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 305–316, 2010.

10. M. A. Bender, B. C. Kuszmaul, S.-H.-Teng, and K. Wang. Optimal Cache-Oblivious Mesh Layouts. *Theory of Computing Systems*, 48(2): 269-296, 2011.

11. M. Farach-Colton, RJ. Fernandes and MA. Mosteiro. Bootstrapping a Hop-Optimal Network in the Weak Sensor Model. *ACM TRANSACTIONS ON ALGORITHMS*, vol. 5, (2009).

12. Bradley C. Kuszmaul. Brief Announcement: TeraByte TokuSampleSort Sorts 1TB in 197s. *The 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2009)*, Calgary, Canada, August 2009.

13. A. Mitrofanova, M. Farach-Colton, and B. Mishra. Efficient and Robust Prediction Algorithms for Protein Complexes using Gomory-Hu Tree. *Pacific Symposium on Biocomputing (PSB 09).*

14. Y. Tang, R. Chowdhury, BC. Kuszmaul, CK. Luk and CE. Leiserson. The Pochoir Stencil Compiler. *Proceedings of the 20th ACM Symposium on Parallel Algorithms and Architectures* (SPAA).

## Service

1. Martín Farach-Colton is serving on the program committee for LATIN 2012, and served on the committee for the String Processing and Information Retrieval (SPIRE 10), the International Conference on Parallel Processing (ICPP 2009) and the Symposium on Combinatorial Pattern Matching (CPM 2009).

2. Bradley C. Kuszmaul served as Vice Chair of the Architecture Track for the International Parallel & Distributed Processing Symposium (IPDPS 2010) and local arrangements chair for the OpenSQL Camp 2010.

3. Michael A. Bender served as the program committee chair for the Symposium on Parallelism in Algorithms and Architectures (SPAA 2009), on the programming committee for the International Parallel & Distributed Processing Symposium (IPDPS 2010) and Principles of Distributed Computing (PODC) 2010, as Vice-chair for the Algorithms Track of International Conference on Distributed Computing Systems (ICDCS) 2011, on the Engineering and Applications Track of the 19th Annual European Symposium on Algorithms (ESA) 2011, on the program committee for LATIN 2011.

## Honors

1. Bradley C. Kuszmaul won 4 of the 12 challenges in the Intel 2009 Threading Challenge (`http://software.intel.com/en-us/contests/Threading-Challenge-2009/codecontest.php`). Kuszmaul's winning codes were for string matching and finding line segment intersections, both of which are examples of problems that require rich indexes in databases. The string matching problem in databases corresponds to building a full-text index, whereas the line-segment intersection problem corresponds to building a geometric index. Both entries were written in Cilk++, which Tokutek is using to improve the multithreaded performance of our database. The writeups can be found at

   - `http://bradley.csail.mit.edu/~bradley/stringmatch/kuszmaul-stringmatch.tar.gz`, and
   - `http://bradley.csail.mit.edu/~bradley/lineseg/kuszmaul-lineseg.tar.gz`

   He also advised Leif Walsh (SUNY SB undergrad) in his project "Duckduckbase: A Concurrent, Durable, In-memory Database Using Solid State" (`http://db.csail.mit.edu/sigmod11contest/walsh.pdf`), which won 3rd prize in the SIGMOD 2011 Programming Contest.

# References

[1] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 399–409, Redondo Beach, California, 2000.

[2] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[3] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *HotStorage '11: Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage*, June 2011.

[4] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 81–92, 2007.

[5] John Bent, Garth A. Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*, 2009.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[7] Transaction Processing Performance Council. Tpc benchmarks. `http://www.tpc.org/information/benchmarks.asp`, 2011.

[8] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science (FOCS 1999)*, pages 285–297, New York, New York, October 1999.

[9] Oracle Inc. Oracle berkeley db 11g. `http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html`, 2011.

[10] Alexey Kopytov. Sysbench: a system performance benchmark. `http://sysbench.sourceforge.net/`, 2011.

[11] Bradley C. Kuszmaul and Mark Callaghan. iibench. `http://bazaar.launchpad.net/~mdcallag/mysql-patch/mytools/annotate/head:/bench/ibench/iibench.py`, 2011.

[12] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE)*, pages 15–24, 2008.

[13] Pat O'Neil, Betty O'Neil, and Xuedong Chen. The star schema benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`, 2007.

[14] Innobase Oy. Innodb. `http://www.innodb.com/`, 2011.

[15] Percona. (`http://www.percona.com/`).

[16] Tokutek Inc. TokuDB. `http://www.tokutek.com/`, 2011.

[17] Mark Wong and Timothy Witham. Database test suite. `http://sourceforge.net/apps/mediawiki/osdldbt/`, 2011.