TokuDB for MySQL®: Scaling High Performance MySQL Databases

Technology Overview White Paper

© 2009 Tokutek, Inc. All rights reserved. MySQL is a registered trademark of MySQL AB in the United States, the European Union and other countries. InnoDB is a registered trademark of Oracle Corporation. All other company, brand and product names contained herein may be trademarks or registered trademarks of their respective holders.

TokuDB for MySQL

TokuDB for MySQL® is a high-performance storage engine that increases MySQL performance and scalability on mixed workloads by one to two orders of magnitude. Rather than optimizing for a narrow or specialized set of use cases by re-engineering existing technology, Tokutek is introducing a new Fractal Tree™ indexing algorithm, which is based on Cache-Oblivious Algorithmics, a fundamentally new approach to building memory-efficient systems that was pioneered by Tokutek's founders. This document will:

- outline the performance characteristics of Fractal Trees and TokuDB for MySQL
- discuss how TokuDB increases performance and eliminates awkward workarounds
- present benchmark results demonstrating TokuDB's performance

Introduction

The majority of relational databases today use indexes to increase query performance. Databases can leverage indexes to significantly reduce the amount of data they examine in order to respond to a query. Indexes are commonly implemented with B-trees, a data structure first described in 1970. The B-tree interface provides simple operations like inserting data and sorted order iteration, the primary operation used by an index. However, B-tree performance is limited by the random I/O characteristics of disks. Database vendors have invested significant engineering effort to work around this limitation, but their efforts have not changed the inherent limiting properties of B-trees: unpredictable performance, poor tradeoffs for particular use cases, and significant administrative complexity.

Fractal Trees

Fractal Trees are a new indexing data structure invented by Tokutek's founders. Fractal Trees are functionally equivalent to B-trees (wherever a B-tree is used, a Fractal Tree can be used instead), but run significantly faster. The Fractal Tree performance advantage can be understood by exploring how Fractal Trees convert random I/O, which involves painfully slow disk seeks, into sequential I/O, which provides up to two orders of magnitude more performance. Consider these examples of sequential versus random I/O for B-trees.

- Loads, in which the rows are sorted before insertion (sequential I/O) can be up to two orders of magnitude faster than inserting into a B-tree as data arrives, for arbitrary primary and secondary keys (random I/O).
- Range queries on a clustered B-tree index can run more than an order of magnitude faster than range queries on a non-clustered B-tree index. Clustering a B-tree index makes I/O more sequential.

- B-tree indexes can have brittle and and unpredictable behavior as they age. While freshly loaded databases tend to have sequential behavior, this behavior becomes increasingly difficult to maintain as the database grows, resulting in more random I/O and poorer performance.

In short, many database management and tuning practices are an attempt to forestall or mitigate random I/O.

By converting random I/O into sequential I/O, Fractal Trees:

- index data at near disk bandwidth rates, no matter what the primary and secondary keys look like
- have range queries that stream data off disk at near disk bandwidth rates, even as the database grows and ages
- deliver performance that is smooth and predictable, with no abrupt performance cliffs

In short, Fractal Trees index data as it arrives while simultaneously providing clustered B-tree-like query performance. In the field, Fractal Trees have the following advantages:

Advantage #1: Fast Indexed Insertions

Fractal Trees are equally fast at indexing data of widely varying characteristics. Whether the data has high cardinality or low cardinality, whether it is more random or more sequential, Fractal Trees always maintain high indexing performance. Thus, they match B-trees in their indexing sweet spot – sequential data – and are up to two orders of magnitude faster elsewhere. Slow indexing is at the root of many problems in databases, problems which often manifest themselves at query time.

For example, DBAs tend to maintain few indexes in databases because keeping more than a few indexes up to date can reduce insertion rates to unacceptably low levels. However, inadequate indexes results in slow queries. The difference in speed between a query against an index and the same query when no index is available can be many orders of magnitude. We have observed up to five orders of magnitude improvement in query time on customer data when the right index is up to date. It should be noted that this claim is not specific to Fractal Trees. Any clustered index would enjoy the same query-performance boost by maintaining the right indexes, but other solutions simply can't keep up with the insertion load. Thus, **fast indexing means more indexes, which means fast queries**.

A second tool for dealing with slow indexing is to buffer data, sort it offline, and then merge the sorted data into the database. Such procedures are notoriously brittle, and due to processing delay, can result in stale data. Fast indexing means that data is indexed as it arrives. Thus, **fast indexing means fresh data**.

Finally, it is possible to speed up B-trees by adding disks. More disks means more random seeks per second but the TCO of the database system goes up as disks are over-provisioned. Fractal Trees get more performance out of each disk so that systems can use fewer disks, consuming less power and taking up less space in a data center, while still delivering high performance. Thus, **fast indexing means lower hardware and data center costs**.

Advantage #2: Fast Index Deletions

Deletions are also fast in Fractal Trees. In some databases, deletions are an even bigger performance sore point than insertions. Almost all storage engines suffer from worse deletion performance than insertion performance. A common use case involving deletions is to maintain an interval of recent data (e.g. the last 30 days). Deleting the oldest data as new data is inserted is the natural solution, but since deletion performance is weak in conventional databases, partitioning and sharding are commonly used workarounds. In partitioning, the data is grouped into blocks of time -- say weeks, for simplicity -- and as the newest partition fills up at the end of a week, the oldest block is dropped.

Partition support has been added in MySQL 5.1, but partitions have drawbacks. Some queries that cut across several partitions suffer a performance hit as the same query must be computed in each partition. Partitions are also an administrative headache: more complexity to manage, more possible points of failure, and often additional hardware to maintain.

The Fractal Tree-based solution is the natural solution to this problem: delete stale data as fresh data arrives. It's as simple as that. Thus, **fast deletions means no need for partitions**. It's important to note that partitions are sometimes used to speed up insertions. But insertions are also fast in Fractal Trees, so partitions are not needed for this reason either.

Advantage #3: Cache Obliviousness

The basis for our algorithmic breakthrough is the theory of Cache-Oblivious Algorithmics (COA). Tokutek's founders pioneered COA, a technique for writing code that works equally well on all memory configurations. The key to designing algorithms and data structures in the COA model is to write code that includes no dependencies on any block sizes in the memory system. Rather, and somewhat surprisingly, COA code is designed to work well at all times with all possible block transfer sizes. The Tokutek founders showed how to use this *block-size universality* to produce very fast code, and the consequences of this for Fractal Trees are outlined above. But the cache obliviousness of Fractal Trees has several other beneficial consequences, notably with respect to compression, tuning, and database aging.

Compression: The compressibility of data is affected by two factors:

- If the data looks more similar, you get more compression. Column stores use this property to achieve high compression.
- If you compress bigger chunks of data at once, you can achieve better compression. Since Fractal Trees are indifferent to memory size configurations, they cannot be tied down to a single block size the way B-trees are. Thus, data in a Fractal Tree can be compressed in a larger chunk.

We typically measure a 3x-12x compression rate on customer data. High compression means less data transferred to and from disk, further improving query performance. Also, since the ideal use of Fractal Trees is to maintain many indexes, compression is important to offset the space requirements of these indexes. Thus, **high compression means faster queries, more indexes and a smaller footprint.**

Tuning: Since algorithms in traditional storage engines must make assumptions about memory size, disk bandwidth and many other factors, they must be tuned to take into account data characteristics, workload, and hardware configuration. Since Fractal Trees are cache oblivious, they do not need to make any such assumptions: block-size universality makes most tuning parameters meaningless. Thus, **Fractal Trees need no tuning.**

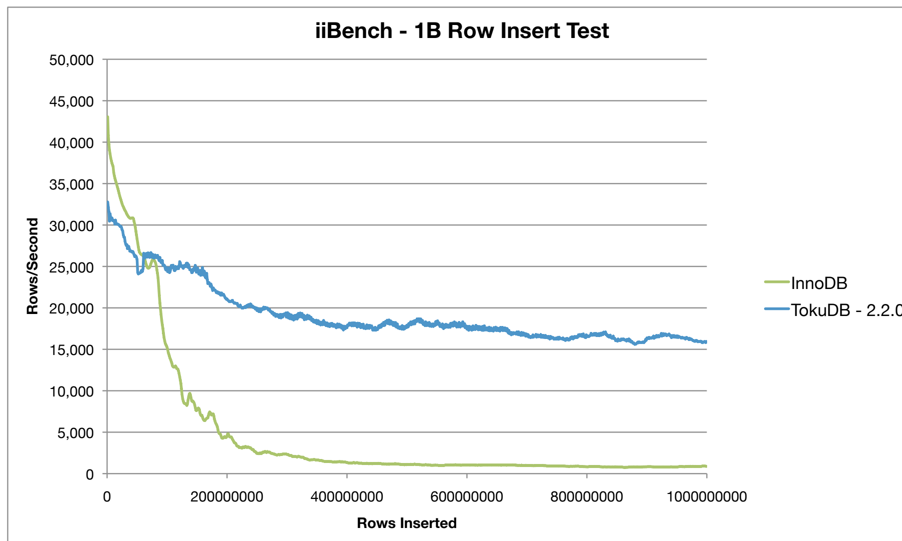
Performance Predictability: Typical storage engines exhibit sudden changes in behavior as various size and age thresholds are crossed. It is virtually impossible to predict when the bottom is going to drop out of the insertion or query rate. Often, the only fix is to dump and reload the database, since freshly loaded data tends to behave better than data accumulated on the fly. But such procedures are time consuming if they work at all and may be impossible to implement on a 24x7 production system. There's no substitute for reliable performance with no sudden discontinuities in performance. Fractal Trees work smoothly across all kinds of known and unknown memory thresholds because they are by design universal with respect to memory effects. The result is smooth performance with respect to the age of the database, its size, and how it was built. Thus, **Fractal Trees have smooth, predictable behavior, with no sudden performance cliffs.**

TokuDB for MySQL

TokuDB for MySQL is a storage engine for MySQL Enterprise 5.1. It has an extensive feature set that includes compression, row-level locking and ACID compliant transactions.

Benchmarking and Comparisons

In this section, we show how TokuDB for MySQL compares with the two most commonly used MySQL storage engines, InnoDB® and MyISAM. In all these tests, we compare MySQL 5.1.30 using TokuDB 2.2.0 versus InnoDB or MyISAM. The tests come from two sources. First we present some insertion load tests against InnoDB using the iiBench benchmark (an open source benchmark available on Launchpad: https://launchpad.net/mysql-patch/mytools). iiBench inserts into a table with an auto-increment primary key and 3 secondary indexes. Inserts are committed after every 1000 insertions. See http://tokutek.com/iibench for details.

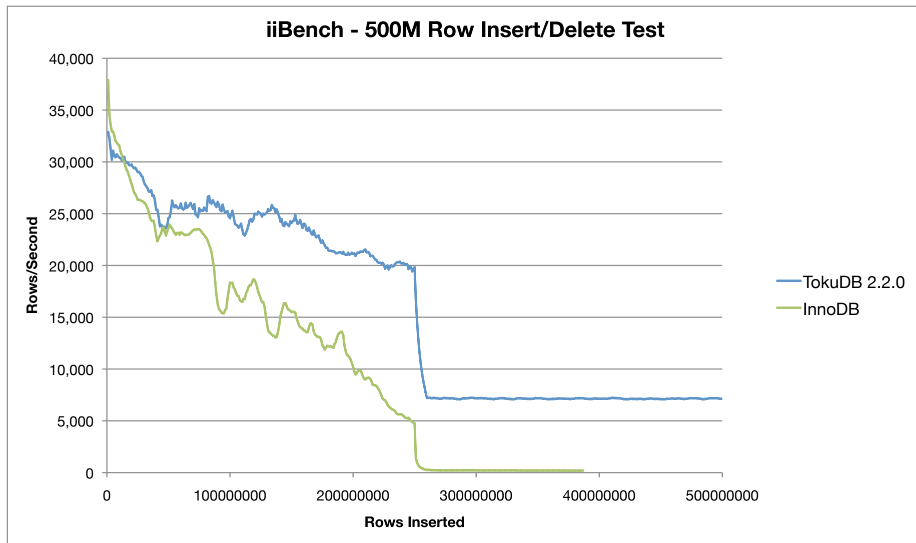


The elapsed time to insert 1B rows was 14.9 hours for TokuDB and 207.5 hours for InnoDB. The terminal rates (last 10M rows) were 15,583 rows/sec for TokuDB and 876 rows/sec for InnoDB.

Observations:

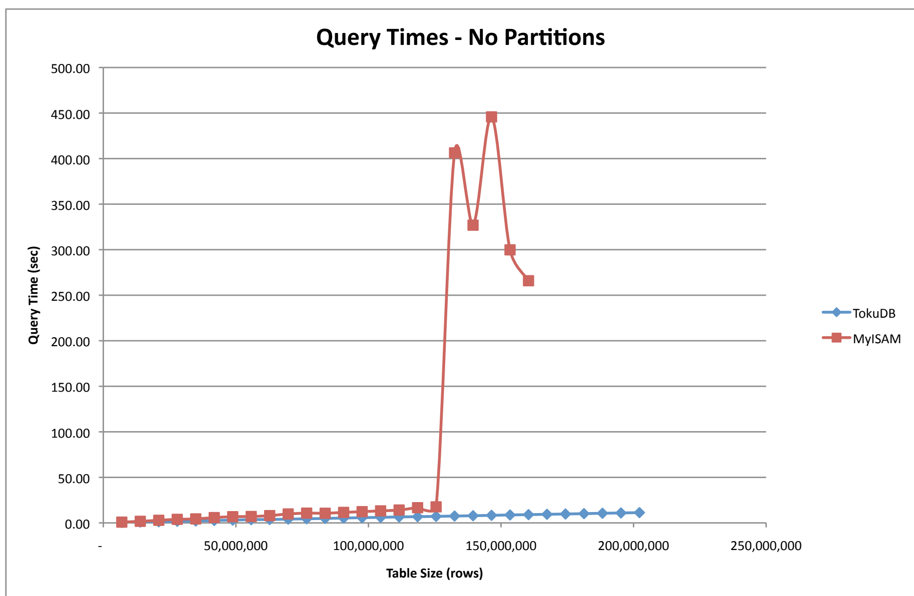
- InnoDB starts out faster than TokuDB when the data fits in memory. As soon as the data no longer fits in memory, the performance of InnoDB drops off dramatically. In contrast, TokuDB has only a gradual decline associated with falling out of memory and is soon more than an order of magnitude faster, ending the run at almost 18x faster.
- Throughout, TokuDB is CPU bound, so we expect to deliver higher performance in future releases by exploiting multi-core architectures.

When we include deletions, the results are more striking. At 250M rows inserted, we started



deleting the oldest 1000 rows each time we inserted 1000 new rows, thus maintaining a rolling window of the most recent 250M rows. The graph measures the insertion rate. The sudden drop in inserts at 250M rows is caused by the time taken by both storage engines to delete rows. TokuDB's insertion rate dropped to 7,109 rows/sec, whereas InnoDB's dropped to 204 rows/sec, a difference of almost 35x.

It's easy enough to get fast insertions if one doesn't care about queries, for example by simply logging rows as they come in. However, the following graph shows that TokuDB also achieves fast and predictable query times. The results show a lab test run by one of our customers. The test is designed to measure query time as a function of table size. The test loads about 7M rows of sample production data into a table, runs a query, and then repeats. Each point shows the time required to complete the query versus the total number of rows in the table.



In this graph, TokuDB and MyISAM scale linearly until 125 million rows have been inserted. TokuDB runs about twice as fast as MyISAM in this range. At that point, MyISAM undergoes a sudden discontinuity, with performance dropping by a factor of 30 and becoming erratic and unpredictable. Our customer reports that TokuDB's performance remains linear for as far out as they measured it.

This graph provides quantitative support for similar qualitative reports we hear from our customers about MySQL performance having sudden performance drops for other storage engines while scaling smoothly and predictably for TokuDB.

Performance data for additional use cases as well as for the most recent version of Tokutek's software can be found on our blog at http://tokutek.com/blog/.

Conclusion

Customer experiences in the field continue to arrive and add to the growing body of evidence showing dramatic advantages of Fractal Tree™ based storage over conventional B-tree based storage. For a wide range of query types, storage sizes, insertion rates, and workload types that reflect the real world need to simultaneously store *and* query, Fractal Tree performance is 10x-50x faster than conventional B-tree based designs.

Free Evaluation

To see how you can improve the performance and reduce the administrative costs of your MySQL infrastructure, please visit http://tokutek.com/tech-wp to request a free evaluation of TokuDB for MySQL.

About Tokutek®

Tokutek provides a 10x-50x performance boost to MySQL users challenged with interactive querying in high volume, always-on applications. The TokuDB Storage Engine, powered by Tokutek's Fractal Tree™ technology, is 100% compatible with existing MySQL code and scales without painful workarounds such as partitioning, compromising on indexes, or expensive hardware.

Tokutek's beta customers use TokuDB for applications ranging from network management to advertising to user-experience personalization on interactive web properties to web 2.0 and clickstream analytics.

Tokutek's team consists of technologists from Akamai, Google, Microsoft, MIT, Rutgers, and SUNY Stony Brook.

© 2009 Tokutek, Inc. All rights reserved.