Caching your application data with MySQL and TokuDB

Rick Pizzi & Andrea Ponzo
Lastminute.com
We are a publicly traded international company, amongst the worldwide leaders in the online travel & leisure industry, and we operate a portfolio of well established European brands: lastminute.com, Bravofly, Rumbo, Volagratis and Jetcost.

**Who is lastminute.com group?**

- 1200 people in 12 countries
- 40 countries
- Nearly 5 million APP downloads
- Around 10 million passengers handled per year
- Offering and services available in 17 languages

*Percona Live Santa Clara 2017*
Who are Rick & Andrea?

Rick

- 20+ years as UNIX system administrator
- Sr MySQL DBA at PalominoDB
- leading Lastminute.com Database Team since 2014

Andrea

- 10+ years on MySQL
- Sr MySQL DBA at Venere.com (Expedia Inc.)
- joined Lastminute.com Database Team in 2015
Caching your application data with MySQL & TokuDB - Agenda

- Overview of storage engines
- Cache design
- InnoDB vs TokuDB
- Cache implementation details
- Cache management
- TokuDB tuning
- Caveats
- Q&A
MySQL storage engines

Storage engines are MySQL components that sit below the SQL layer and handle the actual I/O to/from the tablespaces.

Let’s have a quick look at some of the most commonly used and/or known engines:

- MyISAM
- InnoDB
- TokuDB
- RocksDB
MySQL storage engines: MYISAM

- First storage engine to come with MySQL, since 3.23 bundled with server
- BTREE based
- Very fast

MyISAM drawbacks:

- Table level locking granularity only - no row locking, hence big penalty for concurrent access
- No support for transactions and referential integrity
- No crash recovery ("table is marked as crashed" anyone?) - data loss happens frequently

Not suitable for production use, except in very particular situations and with read only workloads
MySQL storage engines: INNODB

- very mature and full featured engine
- B+TREE based (improved range scans, amongst other things)
- reasonably fast in most cases
- foreign keys support
- fully ACID compliant
- crash safe
- has row level locking supporting a certain degree of concurrency

InnoDB drawbacks:

- INSERT performance degrades as dataset size increases
- B+TREE performance suboptimal with random INSERTs (when INSERTs are not in primary key order)
MySQL storage engines: TOKUDB

- based on Fractal Tree technology (PerconaFT)
- better performance when inserting keys out of order
- uses “message injection” for updates when page not in memory (defers I/O to a later time)
- optimized for compression (although we are not leveraging it this time!)
- reduced I/O compared to InnoDB (also due to larger block size) - SSD friendly!
- transactions, ACID compliant, crash safe, row level locking

TokuDB drawbacks:

- cannot be a replacement for InnoDB in all situations (e.g. lack of foreign keys support)
- although mature, is not as mature as InnoDB
- often slower than InnoDB for read workloads
Mysql storage engines: ROCKSDB

- relatively new engine, not yet included in Percona Server (but will be soon)
- developed by Facebook
- almost production ready (Facebook uses it in production since some time already)
- based on LSM (Log Structured Merge)
- optimized for writes and compression, reduces I/O enormously - very SSD friendly!

RocksDB drawbacks:

- still very young, and under heavy development (not yet GA)
- some features not yet supported, and there are other limitations (e.g. “bin” collation)
- Foreign Key and FullText Index are not supported
- ROW based binary logging MUST be used
Cache design
There are dozens of persistent caches out there, why use MySQL for caching?

- leverage existing technology (MySQL heavily used throughout the company)
- leverage existing knowledge (developing, deploying and maintaining)
- reduce cost of ownership (no need for dedicated skills on other tech)
- we love MySQL!
Cache design (cont’d)

We have several MySQL based cache servers used for caching different types of objects, across different applications.

What we cache:

- data fetched by calls to external providers (where we pay by # of requests, or when it’s expensive to reach the provider)
- temporary object storage for applications that need short lived persistence, eg. application needs to reuse previously generated data
Cache design: workflow

CREATE TABLE `BIG_STORAGE_00` (  
`HASH_ID` char(64) NOT NULL,  
`SERIALIZATION_TYPE` enum('JAVA','PROTOSTUFF') NOT NULL,  
`COMPRESSION_TYPE` enum('DEFLATE','LZ4','NONE','GZIP') NOT NULL,  
`RAW_DATA` mediumblob NOT NULL,  
`LAST_UPDATE` datetime NOT NULL,  
`EXPIRE_DATE` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,  
KEY `HASH_ID_IX` (`HASH_ID`),  
KEY `EXPIRE_DATE_IX` (`EXPIRE_DATE`)  
) ENGINE=TokuDB DEFAULT CHARSET=latin1  
ROW_FORMAT=TOKUDB_UNCOMPRESSED
In order to be able to scale out, our application caching library contains sharding capabilities.

We can:
- shard across any number of tables on same database server
- shard across multiple database servers by using multiple connection pools
Based on [CACHE_KEY % 10]
BIG_STORAGE_00|04 ➔ CACHE01 (cname)
BIG_STORAGE_05|09 ➔ CACHE02 (cname)

Change cache02 DNS
Although TokuDB is really good at compression, we decided to compress the data at the application layer; type of compression used for the payload is saved into COMPRESSION_TYPE column.

Benefits:

- compression methods can be easily added or changed (not limited to what’s available at the DB layer)
- reduced network bandwidth usage between application and database layer
- reduced CPU usage on DB server
- faster query response time (no compression/decompression overhead)
InnoDB vs TokuDB

or why we chose Toku for our cache servers
InnoDB vs. TokuDB: benchmark details

We ran some benchmarks to compare the two engines

- based on sysbench 0.5
- using our cache table definition for the benchmarks
- inserting and reading random rows to/from a single partitioned table
- each row had a randomly variable payload between 0 and 16384 bytes
- each row is inserted into a random partition (partitions spanning 10 days)
InnoDB vs. TokuDB: why we wanted an InnoDB alternative

- it’s known that InnoDB doesn’t perform well with random primary keys
- we have observed InnoDB insert performance degradation when dataset becomes large in size
- unfortunately we initially found out that TokuDB didn’t perform well either... actually, it’s worse than InnoDB at inserting, when PK is really random...

![Graph showing transactions per second (tps) for TokuDB and InnoDB with AVG Payload 16K and Random PK over time.]
InnoDB vs. TokuDB: INSERT benchmark

45 minutes sysbench benchmark run, 40 threads, 8kb avg blob size, 56x Intel Xeon E5-2660 v4 @ 2.00GHz, 2x800G Intel SSD DC P3608

- # InnoDB
  - innodb_flush_method = O_DIRECT
  - innodb_log_files_in_group = 2
  - innodb_log_file_size = 512M
  - innodb_flush_log_at_trx_commit = 2
  - innodb_file_per_table = 1
  - innodb_buffer_pool_size = 2G
  - innodb_file_format=barracuda
  - innodb_write_io_threads=8
  - innodb_read_io_threads=8
  - innodb_io_capacity = 600

- # TokuDB
  - tokudb_commit_sync=OFF
  - tokudb_fsync_log_period=5000
  - tokudb_checkpointing_period=15
  - tokudb_cleaner_iterations=10000
  - tokudb_disable_prefetching=ON
  - tokudb_directio=ON
  - tokudb_cache_size=2G
  - tokudb_row_format=tokudb_uncompressed
  - tokudb_empty_scan=disabled

InnoDB inserted 10,707,190 rows

TokuDB inserted 7,489,946 rows :-(

LastMinute.com Group

Percona Live Santa Clara 2017
InnoDB vs. TokuDB: do we really need a primary key?
InnoDB vs. TokuDB: INSERT benchmark, without Primary Key

45 minutes sysbench benchmark run, 40 threads, 8kb avg blob size, 56x Intel Xeon E5-2660 v4 @ 2.00GHz, 2 x800G Intel SSD DC P3608

InnoDB inserted 20,902,852 rows

TokuDB inserted 70,302,791 rows :-)

# InnoDB
innodb_flush_method = O_DIRECT
innodb_log_files_in_group = 2
innodb_log_file_size = 512M
innodb_flush_log_at_trx_commit = 2
innodb_file_per_table = 1
innodb_buffer_pool_size = 2G
innodb_file_format=barracuda
innodb_write_io_threads=8
innodb_read_io_threads=8
innodb_io_capacity = 600

# TokuDB
tokudb_commit_sync=OFF
tokudb_fsync_log_period=5000
tokudb_checkpointing_period=15
tokudb_cleaner_iterations=10000
tokudb_disable_prefetching=ON
tokudb_directio=ON
tokudb_cache_size=2G
tokudb_row_format=tokudb_uncompressed
tokudb_empty_scan=disabled
InnoDB vs TokuDB: more benchmarks without a primary key

2 hours sysbench benchmark, random key INSERT with 2K avg payload, workload not in memory, SSD

- TokuDB is able to perform 3x better than InnoDB (~45k QPS vs ~15k)
- it does so using less I/O!
- throughput is stable over time, there is no degradation
Benefits of not using a primary key

- not having to look up keys when inserting saves lots of read I/O
- TokuDB is very good at inserting rows when there is no primary key defined, much better than InnoDB
Cache implementation details

Access from application to caches happens through a common library nicknamed “Big Storage”.

Key facts:

- cache consists in key/value pairs
- key is always a SHA-256 hash (64 bytes) - very random
- payload is a compressed BLOB of variable size (avg size depends on cache type, ranging from 4k to 512m)
- number of objects varies wildly depending on cache type and retention policy
Here’s our standard table definition for caches. Note that there is no Primary Key defined :-)

CREATE TABLE `BIG_STORAGE_00` (
    `HASH_ID` char(64) NOT NULL,
    `SERIALIZATION_TYPE` enum('JAVA','PROTOSTUFF') NOT NULL,
    `COMPRESSION_TYPE` enum('DEFLATE','LZ4','NONE','GZIP') NOT NULL,
    `RAW_DATA` mediumblob NOT NULL,
    `LAST_UPDATE` datetime NOT NULL,
    `EXPIRE_DATE` datetime NOT NULL,
    KEY `HASH_ID_IX` (`HASH_ID`),
    KEY `EXPIRE_DATE_IX` (`EXPIRE_DATE`)
) ENGINE=TokuDB DEFAULT CHARSET=latin1 ROW_FORMAT=TOKUDB_UNCOMPRESSED

(partitioning info omitted for the purpose of this slide)
InnoDB vs. TokuDB: real use case - add object to cache*

1 hour sysbench benchmark run, 64 threads, 8kb avg blob size, 56x Intel Xeon E5-2660 v4 @ 2.00GHz, 2 x800G Intel SSD DC P3608

# InnoDB
innodb_flush_method = O_DIRECT
innodb_log_files_in_group = 2
innodb_log_file_size = 512M
innodb_flush_log_at_trx_commit = 2
innodb_flush_log_at_timeout = 5
innodb_file_per_table = 1
innodb_buffer_pool_size = 10G
innodb_file_format = barracuda
innodb_write_io_threads = 8
innodb_read_io_threads = 8
innodb_io_capacity = 600

# TokuDB
tokudb_commit_sync = OFF
tokudb_fsync_log_period = 5000
tokudb_checkpointing_period = 15
tokudb_cleaner_iterations = 10000
tokudb_disable_prefetching = ON
tokudb_directio = ON
tokudb_cache_size = 10G
tokudb_row_format = tokudb_uncompressed
tokudb_empty_scan = disabled
tokudb_enable_partial_eviction = ON
tokudb_read_block_size = 16384

(*) InnoDB purge thread lags terribly behind in this test

local s1 = sb_rand_str("#######")
db_query("BEGIN");
db_query("DELETE FROM BIG_STORAGE_INNO WHERE HASH_ID = SHA2(" .. s1 .. ", 256) AND EXPIRE_DATE > NOW()");
db_query("INSERT INTO BIG_STORAGE_INNO VALUES(SHA2(" .. s1 .. ", 256), 'PROTOSTUFF', 'GZIP', REPEAT(CHAR(FLOOR(RAND()*96)+32), FLOOR(RAND()*16384)), NOW(), DATE_ADD(NOW(), INTERVAL FLOOR(RAND()*(96)+32), FLOOR(RAND()*(16384)), NOW()), DATE_ADD(NOW(), INTERVAL FLOOR(RAND()*(10)+1 day))");
db_query("COMMIT");

Percona Live Santa Clara 2017
InnoDB vs. TokuDB: real use case - fetch object from cache

1 hour sysbench benchmark run, 64 threads, 8kb avg blob size, 56x Intel Xeon E5-2660 v4 @ 2.00GHz, 2x800G Intel SSD DC P3608

# InnoDB
innodb_flush_method = O_DIRECT
innodb_log_files_in_group = 2
innodb_log_file_size = 512M
innodb_flush_log_at_trx_commit = 2
innodb_flush_log_at_timeout = 5
innodb_file_per_table = 1
innodb_buffer_pool_size = 10G
innodb_file_format = barracuda
innodb_write_io_threads = 8
innodb_read_io_threads = 8
innodb_io_capacity = 600

# TokuDB
tokudb_commit_sync = OFF
tokudb_fsync_log_period = 5000
tokudb_checkpointing_period = 15
tokudb_cleaner_iterations = 10000
tokudb_disable_prefetching = ON
tokudb_directio = ON
tokudb_cache_size = 10G
tokudb_row_format = tokudb_uncompressed
tokudb_empty_scan = disabled
tokudb_enable_partial_eviction = ON
tokudb_read_block_size = 16384

(after filling table with approx 7M rows as per previous benchmark)

local s1 = sb_rand_str("#####")
db_query("SELECT RAW_DATA FROM BIG_STORAGE_TOKU WHERE HASH_ID = SHA2("..s1 ..", 256) AND EXPIRE_DATE > NOW()");
Cache management
Cache management is done through MySQL range partitioning.

Normally partitioned by day, partitions are created in advance, and every night we drop previous day’s partition to keep cache footprint small.

Partitions are dropped at night when traffic is low, although TokuDB is pretty good at doing that in background, without noticeable impact.

For special caches that have very short expiration and high volumes of traffic, we partition by the hour and drop old partitions hourly.
A partition management script is needed to create new partitions and drop old ones.

PalominoDB created a small yet powerful perl script for partition management as part of their data management tools. It’s called pdb-parted and can be found on GitHub at this link:


We are using pdb-parted to handle the whole partition management across our clusters, including caches, since many years now.
Cache management (cont’d)

Example usage of pdb-parted

Adding partitions for next 31 days:

```
/usr/local/dba/sbin/pdb-parted --add --interval d +31d h=10.10.10.10,D=thiscache,t=BIG_STORAGE_00,u=partman,p=******
```

Dropping yesterday’s partition:

```
/usr/local/dba/sbin/pdb-parted --drop -0d h=10.10.10.10,D=thiscache,t=BIG_STORAGE_00,u=partman,p=******
```
TokuDB tuning
TokuDB tuning

Here is the relevant part of the MySQL configuration we use in production on SSD based servers...

tokudb_commit_sync=OFF
tokudb_fsync_log_period=5000
tokudb_row_format=tokudb_uncompressed
tokudb_checkpointing_period=15
tokudb_disable_prefetching=ON
tokudb_cleaner_iterations=10000
tokudb_enable_partial_eviction=ON
tokudb_read_block_size=16384
Rule of thumb for Toku memory and I/O: keep the defaults as per Percona recommendation :-) 

- TokuDB is very happy when Toku files can be cached by the O.S. 
- cache will default to 50% of memory that will be used for FT cachetable, remaining memory will be used to map in memory Toku data and indexes for faster access 
- a small cache size (e.g. 1G) will seriously penalize TokuDB for SELECTs (creates cache access contention)
Since our cache data can be rebuilt in case of a crash, we do not enforce durability and are more interested in performance and overall response time.

Hence:

- tokudb_commit_sync=OFF  (similar to InnoDB innodb_flush_log_at_trx_commit=2)
- tokudb_fsync_log_period=5000: redo log is flushed to disk every 5 seconds (in case of crash lose last 5 seconds of data, but we do not really care, could be set higher)
TokuDB tuning: checkpointing

TokuDB checkpointing will impact your performance (QPS) by introducing a lot of variance if left to the default of 60 seconds or set higher.

We keep it running continuously by setting `tokudb_checkpointing_period=15`

Continuous checkpointing will yield a more homogeneous response time and QPS rate.
There are two variables that control size of nodes in PerconaFT:

- `tokudb_block_size` *(default 4M, size of intermediate FT nodes)*
- `tokudb_read_block_size` *(default 64K, size of basement FT nodes)*

Defaults are meant for old spinning disks, not really proper for SSD.

- a 16K value for `tokudb_read_block_size` brought us big performance gain both for read and write on SSD based server
- best thing is to experiment (but don’t set cache memory too low)
Caveats when using TokuDB in production
TokuDB has many moving parts...

Unlike InnoDB, there are many moving parts in TokuDB that if tweaked can have an huge impact on your workload. Most notably ones listed below:

- `tokudb_cache_size`
- `tokudb_checkpointing_period`
- `tokudb_cleaner_iterations`
- `tokudb_cleaner_period`
- `tokudb_fsync_log_period` (if `tokudb_commit_sync OFF`)
- `tokudb_block_size`
- `tokudb_read_block_size`
- `tokudb_fanout`
Beware of important TokuDB performance bugs!

- Implement tree map file block allocation strategy
  
- TokuDB ::info uses bad/lengthy function to provide table status*
  
- tokudb does not use index even if cardinality is good*

Bugs fixed in:

- Percona Server release 5.6.35-81.0
- Percona Server release 5.7.17-12

* reported by us
Beware of unresolved MySQL optimizer bugs...

Statistics for partitioned tables in MySQL are only computed on largest partition (see https://bugs.mysql.com/bug.php?id=44059).

If largest partition is in the past, where all rows have EXPIRE_DATE < NOW(), we are in trouble as both indexes (on HASH_ID and EXPIRE_DATE) have same cost and the optimizer can randomly pick the wrong one and scan all partitions.

Workaround: make sure the index on HASH_ID comes *before* the one on EXPIRE_DATE to avoid production downtime!
See: https://bugs.mysql.com/bug.php?id=85126

In any case don’t keep stale partitions around - drop them!
Q & A
Let’s keep in touch

riccardo.pizzi@lastminute.com
andrea.ponzo@lastminute.com

https://mysqlnoob.blogspot.com [ Rick’s blog ]
THANKS

www.lastminute.comgroup.com