MySQL Conference & Expo

Presented by MySQL Conference & Expo O'REILLY

InnoDB Performance Optimization

Heikki Tuuri, Innobase Oy/Oracle Corp.
Peter Zaitsev, Percona Ltd

April 23-26 2007

About Speakers

- Heikki Tuuri

  Creator of InnoDB Storage engine

  InnoDB Lead Developer

- Peter Zaitsev

  MySQL/InnoDB Performance Expert

  Long time InnoDB User

- Speaking together

  to share internals and practical use insights

It all starts with Application Design

General Application Design is Paramount

- Design your schema, indexes and queries right

  Storage engine aspects are often fine tuning

- In some cases Storage Engine selection may affect your schema layout and indexes
- We're not covering general schema design guidelines in this presentation, but will focus on the InnoDB Storage Engine.

Each storage engine is special

- MySQL offers multiple Storage Engines
- Each of them has unique design and operating properties
- Application written for one storage engine may not perform best with other storage engines
- Each Storage Engine has special optimizations so they can benefit from certain design patterns
- We'll cover DO and DON'T for the InnoDB Storage Engine

Make use of Transactions

- There are always transactions with InnoDB, even if you do not use them explicitly

  Each statement will be in its own transaction (assuming you run in the “autocommit mode)

  With transaction commit overhead for each statement

- Wrap multiple updates in the same transaction for efficient operation (SET AUTOCOMMIT = 0; ... COMMIT; ... COMMIT;)

  Do not make transactions too large, however

  Make sure you're catching Deadlocks and Wait Timeouts

Do not use LOCK TABLES

- LOCK TABLES is designed to work with table level locking storage engines

  With row level lock storage engines, transactions are better choice

  LOCK TABLES behavior with InnoDB tables is different in MySQL versions and depends on –**innodb_table_locks**

  can give problems for portable applications if you port from MySQL-4.0 to later

  Behavior might not be as may be used to with MyISAM tables.

PRIMARY KEY Clustering

- PRIMARY KEY is Special
  - Accessing data by PRIMARY KEY is faster than other keys
    - True for both In-Memory and Disk Based accesses
    - Try to do most lookups by primary key
  - Data is clustered by PRIMARY KEY
    - Sequential PK values will likely have data on the same page
    - PK range and prefix lookups are very efficient
    - Can be used to cluster data accessed together
      - Storing user messages one can use (user_id,message_id) primary key to keep all users messages in a few pages.
    - PK is a “covering index” for any set of fields in the PK

Cost of Clustered Primary Key

- PRIMARY KEY in random order are costly and lead to table fragmentation (primary key inserts should normally be in an ascending order)

  Load data in primary key order if you can

  Sometimes changing primary key to auto_increment is a good idea

- There is always a clustered key internally even if you do not specify one

  So better define one and use it

- PRIMARY KEY column updates are expensive

  Requires row data physically to be moved from one place in the index to another.

  Generally not a good schema/application design either!

Keep PRIMARY KEY Short

- Secondary indexes use primary key to refer to the clustering index

  Making primary key value part of any index

- Long primary keys make your indexes long and slow

  Keep them short

- You can often change current primary key to UNIQUE KEY and add auto_increment PRIMARY KEY; you can't have InnoDB to create its internal primary key simply by changing a PRIMARY key to UNIQUE because MySQL will internally convert a not-NULL UNIQUE key to a primary key if one is missing

- If you only have PRIMARY KEY on the table and have all lookups done by it, leave it even if it is long, as PK lookups are so much faster.

InnoDB Indexing

- Be easy on UNIQUE Indexes

  They do not use the “insert buffer” which can speed up index updates significantly

- Indexes are not prefix compressed

  so they can take much more space than for MyISAM

  avoid excessive indexes.

- Keep Updates Fully Indexed

  Or you can see unexpected locking problems

  DELETE FROM users WHERE name=“peter”

  may have to lock all rows in the table if the column name is not indexed.

Auto Increment may limit scalability

- Auto-Increment INSERTS may use table level locks (but only to the end of the INSERT statement, not transaction)

  even if you specify the auto_increment column value!

- Limits scalability for concurrent inserts
- A fix being worked on
- Work around by assigning values outside of MySQL

  be careful with uuid as they result in both long and random primary keys.

Multi Versioning

- Complements row level locking to get even better concurrency
- Standard SELECT statements set no locks, just reads appropriate row version

  LOCK IN SHARE MODE, FOR UPDATE modifiers can be done to do locking reads

- Even long running selects do not block updates to the table or other selects
- Overly long queries (transactions in general) are bad for performance as a lot of unpurged versions accumulate. READ COMMITTED can ease the problem.

  InnoDB is only able to remove a row version when no transactions are open which can read it.

... FOR UPDATE and LOCK IN SHARE MODE

- Locking selects are executed in read committed mode

  Because you can't lock a row which does not exist

  So results of these queries can be different than for standard SELECTs

- SELECT ... FOR UPDATE always has to access row data page to set the lock, so it can't run index covered queries which can slow down queries a lot

Reducing Deadlocks

- Deadlocks are normal for a transactional database
  - Non-locking SELECT statements do not deadlock with InnoDB
  - Make sure to handle deadlocks and lock wait timeouts in your application
- Make sure your transactions lock data in the same order when possible
- Have update chunks smaller (chop transactions)
- Use SELECT ... FOR UPDATE if you're going to update most of the selected rows.
- Use external locking to avoid problem - Application level locks, SELECT GET_LOCK('mylock') etc.

How isolation modes affect Performance

- InnoDB supports a number of Isolation Modes, which can be set globally, per connection or per transaction.

  READ UNCOMMITED - Rarely used, can use if you are fine with dirty reads but performance improvement is limited

  READ COMMITED – Results of all committed transactions become visible to the next statement. May be more efficient than higher isolation levels. Allows old versions to be purged faster.

  In MySQL-5.1, InnoDB does little 'gap locking' on this level: use row-based replication and binlogging to avoid problems!

  REPEATABLE READ – Default. Reads within transactions are fully repeatable, no phantom rows.

  SERIALIZABLE - Makes all selects locking selects, avoid when possible.

Foreign Keys Performance

- InnoDB checks foreign keys as soon as a row is updated, no batching is performed or checks delayed till transaction commit

  Foreign keys are often serious performance overhead, but help maintain data consistency

- Foreign Keys increase amount of row level locking done and can make it spread to a lot of tables besides the ones directly updated

- Foreign Key locking in a child table is done when the parent table is updated

  (SELECT ... FOR UPDATE on the parent table does not lock the child table)

Restrict Number of Open Transactions

- InnoDB performs best with a limited number of open transactions and running queries.
- Multiple running queries may cause a lot of thrashing bumping into each other
  - Work is done to improve performance in such cases
  - **innodb_thread_concurrency** can be used to restrict number of threads in InnoDB kernel
- Many open transactions make lock graph building more complicated and increase some other overhead.
- When possible, keep a limited number of queries running at the same time, do queuing on application side

Beware of a very high number of tables

- InnoDB has its own table definition (dictionary) cache independent of the MySQL **table_cache** variable
- Once opened, InnoDB never removes table from the cache.
- 4KB+ may be consumed by each table

  InnoDB in MySQL 5.1 has reduced this number by 50 % - 75 %
- On restart, statistics are recomputed for each table

  So the first time open operation is very expensive

  plus MySQL table_cache serializes these operations

INSERT ... SELECT

- INSERT... SELECT statement runs locking select
- Required for logical level replication to work properly

  problem goes away with MySQL 5.1 row level replication and the READ COMMITTED isolation level

- Behavior is the same whenever you have **log-bin** enabled or not, to keep things consistent
- **innodb_locks_unsafe_for_binlog** helps in 5.0, but your replication may get broken

  it also disables next-key locks

- **SELECT ... INTO OUTFILE + LOAD DATA INFILE** can be often use as non-blocking safe alternative

Next Key Locks (Gap Locks)

- InnoDB not only locks rows themselves but the “gap” between rows as well
- Prevents phantom rows

  Makes “REPEATABLE READ” really repeatable with InnoDB
- Needed for MySQL statement level replication to work properly.
- Increases locking for some write heavy workloads.
- Can be disabled if you're not running binary logging (for replication or recovery)
- Is safe to change in MySQL 5.1 if you use row-based replication

Count(*) facts and myths

- "InnoDB does not handle count(*) queries well" - **Myth**

  Most count(*) queries are executed same way by all storage engines

  SELECT COUNT(*) FROM articles WHERE user_id=5

- "InnoDB does not optimize count(*) queries without where clause" - **Fact**

  SELECT COUNT(*) FROM users;

  InnoDB can't simply store count of the rows as it each transaction has its own view of the table. Significant work required to implement

  You can use triggers and counter table to work it around

  SHOW TABLE STATUS LIKE "users" will show approximate row count for the table (which is changing all the time)

InnoDB and Group Commit

- Group Commit – commit several outstanding transactions with single log write
  - Can improve performance dramatically, especially if no RAID with BBU
- In MySQL 5.0, group commit does not work with binary logging
  - Due to a way XA (distributed transactions) support was implemented
  - Watch out if upgrading from MySQL 4.1

Back to basics with Server Settings Tuning

It all starts with Memory

- **innodb_buffer_pool_size**

  Specifies main InnoDB buffer – Data and Index pages, insert buffer, locks are stored here

  Very important for performance on large data sets

  Much more efficient than OS cache, especially for Writes

  InnoDB has to bypass OS buffering for writes

  Can be set to 70-80% of memory for dedicated InnoDB-Only MySQL

  Default value is just 8M, independent of available memory; make sure to configure it

- **innodb_additional_mem_pool**

  just stores dictionary, automatically increased, do not set too high

InnoDB Logging

- **innodb_log_file_size**
  - Dramatically affects write performance. Keep it high
  - High values increase recovery time though
    - Check how large logs you can afford
  - 4GB total size limit
- **innodb_log_files_in_group**
  - this number of files of specified size are used for log.
  - Usually no need to change default value

InnoDB Logging

- **innodb_log_buffer_size**

  Do not set over 2-8MB unless you use huge BLOBs, Log file is flushed at least once per second anyway

  Check Innodb_os_log_written growth to see how actively your logs are written.

  InnoDB Logs are physio-logical, not page based so they are very compact

- **innodb_flush_logs_at_trx_commit**

  By default logs are flushed to the disk at each transaction commit

  Required for ACID guarantees, expensive

  Can set to 2 or 0 if you can afford losing transactions for last 1 sec or so (ie if you're using it as MyISAM tables replacement)

InnoDB Log Resizing

- Is not as simple as changing option and restarting :)
- Shut down MySQL Server
- Make sure it shut down normally (check error log for errors)
- Move away InnoDB log files ib_log*
- Start MySQL Server
- Check error log files to see it successfully created new log files.

innodb_flush_method

- Specifies a way InnoDB will work with OS File System
- Windows: unbuffered IO mode is always used
- Unix: can use fsync() or O_SYNC/O_DSYNC for flushing files

  fsync() is usually faster; allows accumulating multiple writes and executing them in parallel

  Some OS allow disabling OS caching for InnoDB data files

  Good. You do not want data to be cached twice – waste.

- Linux: O_DIRECT uses direct unbuffered IO

  Avoids double buffering, May make writes slower

innodb_file_per_table

- InnoDB can store each table in its own file
- Main tablespace is still needed for system needs
- Can help to spread tables to multiple disks
- Allows to reclaim space if a table is dropped
- Sometimes slower for writes as fsync() is called sequentially
- Can increase startup/shutdown time with large number of tables

Other File IO Settings

- **innodb_autoextend_increment** – specifies growth increment for shared tablespace (not for per table tablespaces)

  larger values allow to reduce fragmentation.

- **innodb_file_io_threads** – changes number of IO threads, on Windows only. Note all 4 threads are doing different jobs

- **innodb_open_files - ** number of files used for per table tablespaces. Increase if you have a lot of tables

  No stats available so far to show number of re-opens InnoDB needs to do

- **innodb_support_xa** setting to 0 reduces work InnoDB should do on transaction commit. Binlog can get out of sync

Minimizing restart time

- InnoDB buffer pool may have a lot of unflushed data

  So shutdown may take very long time

- If you need to minimize downtime:
- SET GLOBAL **innodb_max_dirty_pages_pct**=0
- Watch **Innodb_buffer_pool_pages_dirty** in SHOW STATUS
- As it gets close to 0 shut down the server
- During this operation performance will be lower as InnoDB will be flushing dirty pages aggressively.

Troubleshooting runaway Purge

- InnoDB does not remove rows on delete (and old row versions on update) because these may be needed by other transactions
- Purge thread is used to clean up these unused rows
- In some workloads, the Purge thread may not be able to keep up and the tablespace will grow without bounds.

Check “TRANSACTIONS” section in **SHOW INNODB STATUS**

- **innodb_max_purge_lag** – limits number of *transactions which have updated/deleted rows*
- *Will delay insert/updates so purge thread can keep up*
- *Why do not we get to have multiple purge threads instead?*

Concurrency Control Settings

- Settings help to adjust how InnoDB handles a large number of concurrent transactions
- **innodb_thread_concurrency** – Number of threads allowed inside InnoDB kernel at the same time (0 – no limit)

  2*(NumCores+NumDisk) is good value in theory, smaller usually work better in practice

- **innodb_commit_concurrency** - Number of threads allowed at commit stage at the same time
- **innodb_concurrency_tickets** - Number of operations thread can do before it has to exit kernel and wait again
- **innodb_thread_sleep_delay**
- **innodb_sync_spin_loops**

Unsafe ways to gain performance

- InnoDB has a lot of checks and techniques to minimize chance of data being corrupted or lost
- **innodb_doublewrite** - protection from partial page writes, only disable if OS guarantees it does not happen
- **innodb_checksums** - checksums for data in pages, helps to discover file system corruption, broken memory and other problems

Causes a few percent of overhead for most workloads

Disable when such performance gain is more important

Benchmarks ?

InnoDB SHOW STATUS Section

- MySQL 5.0 finally has some InnoDB performance counters exported in **SHOW STATUS**

  They are GLOBAL while most of other counters are per thread now

  They are mostly taken from SHOW INNODB STATUS

- Will only list some examples

- **Innodb_buffer_pool_pages_misc** - number of pages in BP used for needs other than caching pages

- **Innodb_buffer_pool_read_ahead_rnd** – number of random read-aheads InnoDB performed

- **Innodb_buffer_pool_read_requests, Innodb_buffer_pool_reads** can be used to compute cache read hit ratio

SHOW INNODB STATUS

- The tool for InnoDB troubleshooting

  "Send a couple of **SHOW INNODB STATUS** outputs when it happens"

- Has information as in **SHOW STATUS** plus much more
- Information about running transactions (their locks etc.)
- Information about last deadlock, foreign key, etc.
- Information about latches, spinlocks, OS waits
- More details

  http://www.mysqlperformanceblog.com/2006/07/17/show-innodb-status-walk-through/

SHOW MUTEX STATUS

- A tool to show what mutexes are hot for your workload
- Details of what really happens with which mutexes – spin locks ? OS Waits ?
- **timed_mutexes** - track how long OS Wait was taking

```
Mutex:&kernel_mutex
Module:srv0srv.c
Count:1828074122
Spin_waits:762647
Spin_rounds:4781433
OS_waits:96879
OS_yields:155883
OS_waits_time:0
```

Hardware and OS Selection

Hardware and OS Selection Checklist

- Which CPUs and how many of them ?
- How Much Memory ?
- How to set up IO Subsystem ?
- Does OS Selection matter ?
- Which File System is best to use ?

Selecting CPUs

- Different CPUs/Architectures scale differently with InnoDB
- Old “NetBurst” based Xeons scale poorly
- New “Core” based Xeons and Opterons are better
- X86_64 is the leading
- Multi-Core works well with InnoDB
- Over 8 cores per system is reasonable limit
  - Depends on workload significantly
  - Innobase is working on further improvements
- Scale Out, use multiple lower end servers.
- 32bit CPUs should be dead by now, so 32bit OS

How much memory ?

- Memory is most frequent performance limiting factor for well tuned applications
- InnoDB can use large memory amounts efficiently
- Working set must fit in memory

  The data pages which are accessed most often

  Do not count by rows:

  100,000,000 of 100 byte rows, random 1,000,000 are working set – can touch most of the pages.

- Can be 5% of total database size or can be 50%
- Make sure to use a 64bit platform, OS and MySQL Version.

How to set up IO SubSystem

- InnoDB loads a few hard drives well, but not 100 of them
  - 6-8 per node seems to be optimal configuration
- Directly Attached storage usually works best
- SAN – increased latency, expensive
- NAS – Avoid, risk of data corruption
- ISCSI – good for some cases, increased latency
- RAID – Battery backed up cache is very important
  - Make sure you have BBU before enabling WriteBack cache
- Hard Drive cache itself should be turned off, or make sure it is flushed on fsync() or corruption can happen in OS crash.

Local storage configuration

- Logs on separate RAID1 volume

  Can be helpful, in many cases better to share disk for data

- Binary logs on separate volume – can be good idea for backup recovery reasons

- RAID10 good for tablespace

  degraded performance can be worse than expected.

- RAID5 can be good for certain workloads

  just make sure you account for degraded performance.

- Large RAID Stripe (128K+) is best in theory but many RAID controllers do not handle these well.

- Software RAID is OK, especially RAID1

Does OS Selection Matter ?

- Consider Performance, Tools available, Community Experience
- Windows – used for development, small installations, few Web/Enterprise scale projects
- Solaris – offering some great tools now, works to make MySQL work well with it, bad community support.
- FreeBSD – had history of problems with MySQL in general, now gets better, fewer tools available, less usage in production.
- Linux – Most commonly used platform for production and Development. Tools like LVM, Journaling filesystems.

Selecting FileSystem

- Applies mainly to Linux which has too many choices
- **EXT3** – default filesystem in most distributions, works OK for lower end installations
- **ReiserFS** – support removed from many Linux distributions. Generally no big win with typical MySQL workload
- **XFS** – Used with a lot of drives in RAID, can give serious performance improvement
- **JFS** – Rarely used at this point.
- **Raw partition** for InnoDB tablespace – rarely used.
- There are often too high expectations about performance gains by switching file systems.

Recent InnoDB Performance Developments

InnoDB Scalability Patches

- Decreased contention over buffer pool pages

  Available in 5.0, backported to 4.1

- Improved sync_array implementation in MySQL 5.1
- Performance gains are very different based on workload,hardware, concurrency
- Can range from few percent to multiple times
- Performance still goes down with high number of concurrent threads.
- Prototype for further scalability improvement patches is available from community

Other Improvements

- Row Level replication in MySQL 5.1 eases gap locking
- Working on removing Auto_increment “table locks”
- Zip compression of database pages
- Fast index creation
  - No full table rebuild required
  - “Sorting” gives less physically fragmented index

Questions from the audience

- pz@mysqlperformanceblog.com
- Visit blog for more Innodb tips

  http://www.mysqlperformanceblog.com
- Looking for some help ?

  consulting@mysqlperformanceblog.com