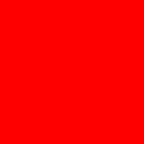


ORACLE®

What's new in MySQL 5.5? Performance/Scale Unleashed

Mikael Ronström
Senior MySQL Architect



The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

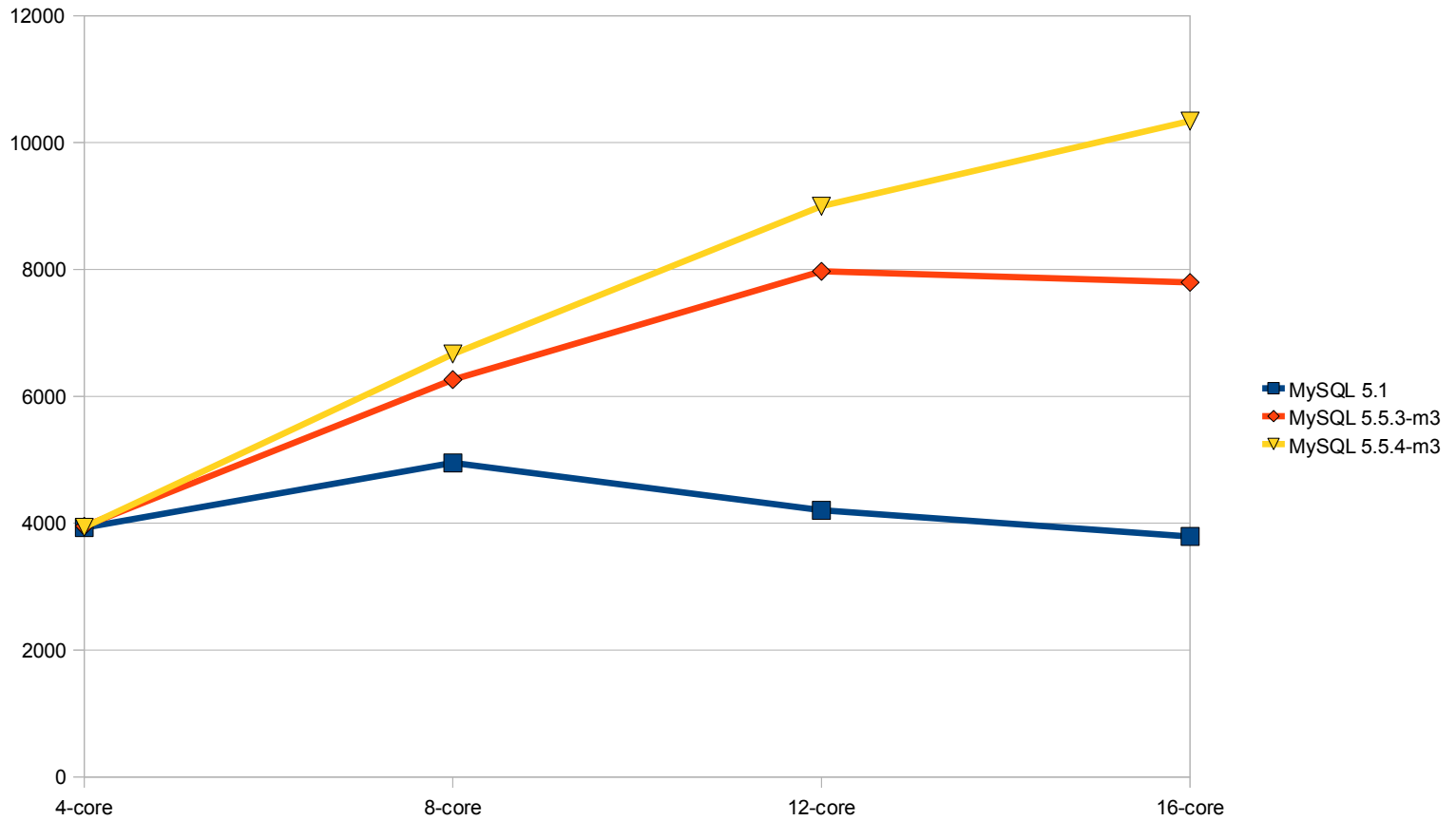
Outline

- Scalability Improvements
- Scalability bottlenecks and their elimination in MySQL Server
- Scalability bottlenecks and their elimination in InnoDB
- Recovery improvements in InnoDB
- Impact of solutions

Multi-Core CPU Development

- Intel recently released 8-core CPU's with 2 threads per core, 2-socket and 4-socket solutions with this CPU will be common and even 8+ sockets will be available
- AMD recently released a 12-core CPU available in 2, 4 and 8 socket servers

OLTP RW Less Read



Scalability bottlenecks in MySQL Server in MySQL 5.1

- LOCK_open mutex (main bottleneck for Read-Only workloads)
- LOCK_alarm mutex
- LOCK_thread_count
- LOCK_grant (protects GRANT tables)
- THR_LOCK_lock (used for TABLE lock handling)
- THR_LOCK_charset
- Query cache mutexes + Binlog mutexes

LOCK_open mutex in MySQL 5.1 (1)

- In normal query workloads LOCK_open is used in open_table and close_open_tables
- Protects refresh_version (incremented through FLUSH TABLE)
- Held while calculating hash value for table name
- Held while searching for a free TABLE object in cache of open_tables
- Held while searching for TABLE objects to free due to too many TABLE objects in cache
- Held while opening table from FRM file if needed
- Held while creating new TABLE object if needed

LOCK_open in MySQL 5.1 (2)

- Held while putting back TABLE object to free list
- Held while resetting Handler object
- Held while releasing BLOB objects allocated in query
- Held while freeing TABLE objects

LOCK_open in MySQL 5.5 (1)

- New subsystem MDL (MetaData Locking) introduced in MySQL 5.5
- MDL has hash from table name to table MDL lock object
- MDL lock object contains cached TABLE_SHARE object
- TABLE_SHARE object has list of free TABLE objects and list of used TABLE objects

LOCK_open in MySQL 5.5 (2)

- MDL hash protected by mutex, hash function executed outside of mutex, easy to split into multiple hashes in future versions
- MDL table lock object protected by separate mutex
- refresh_version now atomic variable outside of LOCK_open
- Removed release of BLOB memory from LOCK_open
- Removed reset of Handler object from LOCK_open
- Removed hash tables from LOCK_open (moved to MDL hash mutex)

LOCK_open in MySQL 5.5 (3)

- Removed search for free object on TABLE_SHARE, now separate free list and used lists (this is one of the reasons of worsening performance of Read-Only workloads with increasing number of connections in MySQL 5.1)
- Creation of TABLE object from TABLE_SHARE no longer protected by LOCK_open => Handler::open not protected by LOCK_open (reported as an issue since InnoDB grabs its kernel_mutex here)
- Creation of TABLE_SHARE from FRM file still under LOCK_open protection, work is ongoing to fix this as well (usually avoided by ensuring config is properly set)

LOCK_open split into MDL hash, MDL lock, LOCK_open, atomic refresh_version

- So effectively we have split LOCK_open mutex into 3 mutexes + an atomic variable
- All these mutexes are acquired and released one at a time
- Special "tricks" needed to separate MDL hash from MDL table lock object

Special trick for separation of MDL hash mutex and MDL table lock mutex (1)

- MDL hash mutex held when inserting, deleting and searching in MDL hash
- MDL table lock object is the objects found in MDL hash
- MDL hash search retrieves MDL table lock object
- MDL table lock object can be deleted
- Two reference counters, one incremented when read from hash (protected by MDL hash mutex), one incremented when acquired MDL table lock mutex (protected by this mutex)

Special trick for separation of MDL hash mutex and MDL table lock mutex (1)

- When deleting both mutexes are held, ok to delete if both reference counters have same value, otherwise delete_flag is set
- When acquiring MDL table lock mutex check if delete_flag is set, if set, delete object and retry search in hash table

LOCK_alarm mutex

- Protects alarm handling which is used by MySQL in network handling
- Removed on platforms that support SO_SNDTIMEO and SO_RCVTIMEO (timeouts on socket reads) => no alarm handling needed for socket reads

LOCK_thread_count in MySQL 5.1 => MySQL 5.5

- Protects list of connections
- Protects global Query Id variable
- Protects global count of Running Threads
- Global Query Id variable made atomic variable
- #Running Threads made atomic variable
- Makes LOCK_thread_count mostly used at connect/disconnect and various SHOW commands, previously used 1-2 times per query

THR_LOCK_charset mutex

- In some cases in MySQL 5.1 this mutex was held unnecessarily during normal charset set-up for various charsets
- Removing this unnecessary mutex lock/unlock had dramatic effect on effected workloads

Scalability bottlenecks in MySQL 5.5

- LOCK_open mutex
- MDL hash mutex
- The above two are still the main MySQL bottlenecks although now much less of a bottleneck, can fairly easily be split into different mutexes for different tables
- MDL table lock mutex
- LOCK_grant (RW-lock almost always using R-lock, can be fixed by Read-lock optimized RW-locks)
- THR_LOCK_lock
- Query cache mutexes + Binlog mutexes

Scalability Bottlenecks in MySQL 5.1/InnoDB 1.0.6

- Buffer Pool mutex (Very hot mutex)
- Rollback Segment Mutex (Hot mutex)
- Log Mutex (Very hot mutex)
- Index RW-lock (Hot in some cases)
- Dictionary Mutex (Hot in some cases)
- Block mutex (1 per page) (can be hot for some pages)
- Page RW-lock (1 per page)
- Kernel Mutex (gets hot as number of trx's increase)
- InnoDB Thread Concurrency Mutex (can be shut off)
- Adaptive Hash Mutex (can be shut off)

InnoDB Mutex analysis on Sysbench RW using InnoDB 1.0.6

- Log Mutex acquired around 350k/sec, held about 75% of time
- Buffer Pool mutex acquired around 700k/sec, held about 50% of time
- Rollback Segment Mutex acquired around 20k/sec, held about 25% of time

Log Mutex Analysis

- Protects Log Data Struct (LSN number and other Log related data)
- Protects Log Memory Buffer
- Protects Writing of Mini-transactions to buffer pages (updates LSN number of pages among other things)

Log Mutex Analysis (2)

- Activity to write to buffer pages is completely independent of the rest of the log mutex activity
- However it is necessary to ensure that the buffer pages are updated in LSN order
- To handle this we introduce a new mutex `log_flush_order` mutex which is taken before writing to the buffer pages, it is taken while still holding the log mutex to maintain LSN order, immediately after acquiring it we release the log mutex

Log Mutex Analysis (3)

- Split of Log Mutex has the advantage that the Log Mutex and the Buffer Pool mutex are separated from each other
- To have to acquire the hot buffer pool mutex while holding the hot log mutex isn't a good idea, so this design removes this need

Buffer Pool Mutex Analysis

- Buffer Pool mutex protects many data structures in the Buffer Pool: LRU, Flush List, Free List, Page Hash Table
- To decrease pressure of the buffer pool there are essentially two ways:
- 1) Split the buffer pool mutex and have different mutexes protect different parts of the buffer pool, e.g. one mutex to protect LRU, another to protect Flush List and another to protect Page Hash
- 2) Split the buffer pool into multiple instances, each buffer pool with its own buffer pool mutex

Buffer Pool Mutex Analysis (2)

- I did a lot of analysis of splitting out the Flush List from the buffer pool and splitting out the page hash from the buffer pool
- The results were very promising in the context of the “old” InnoDB, but the introduction of compressed pages into the buffer pool in the InnoDB plugin made it very hard to get good results using these versions due to too many cases of having to hold multiple buffer pool mutexes

Buffer Pool Mutex Analysis (3)

- To access a page within a buffer pool requires 3 locks to be taken, the buffer pool mutex, the block mutex and the Page RW-lock.
- Introducing a Page Hash mutex requires yet one more mutex to be acquired as part of accessing a page, the buffer pool and page hash mutex are possible to separate but increasing number of mutexes to acquire also increases code pathlength in a critical part of the system

Buffer Pool Mutex Analysis (4)

- So effectively to split the buffer pool into multiple buffer pool instances makes a whole lot of sense
- Analysing this split it turned out that we could avoid holding more than one buffer pool mutex instance in all query execution code, only in some code executed rarely was it necessary to hold all mutexes at the same time

Buffer Pool Mutex Analysis (5)

- Benchmarks showed that the major performance enhancements was realised by either splitting out an array of page hash mutexes or by splitting the buffer pool into multiple instances
- The best results came by splitting the buffer pool into multiple instances
- Also multiple buffer pools is very likely to be better in situations with high IO load on the system
- A configurable number of buffer pool instances means we can tune the system for optimum number of instances as well

Buffer Pool Mutex Analysis (6)

- To further the decoupling of the log subsystem from the buffer subsystem we also split out the Flush List from the buffer pool mutex, to ensure that while holding the new log_buf mutex we need not hold the still fairly hot buffer pool mutex
- “hot”-ness of buffer pool mutex instances is very different, accesses to pages in a database is rarely fair, in particular root pages in indexes cause fairly high variance

Rollback Segment Mutex Analysis

- Accessing the rollback segment mutex is one of the main cause of degrading performance with many connections (together with kernel mutex and LOCK_open in MySQL 5.1)
- As the number of updates to a record by different transactions increases, the path to get to the proper record increases
- To get to the proper record is protected to some extent by the rollback segment and can even at times include IO while holding this mutex
- Mutex analysis shows the variance of hold times of this mutex to be very high and hold times are also very high

Rollback Segment Mutex Analysis (2)

- The hold times of this mutex effectively means that it doesn't really fit the InnoDB mutex implementation which is more geared towards mutexes like the buffer pool mutex with very short duration and many concurrent accesses to it
- However it is possible to split this mutex, and so have been done, it's now 128 mutexes
- This removes this mutex from the picture
- It is possible to reuse old InnoDB installations even with this split of rollback segments

InnoDB Purge Activity

- Purge activity comes about as support for consistent reads requires old records to be kept in data and indexes until no transaction will anymore access them
- Previously purge activity was performed as part of master thread
- Higher transaction rates means that quite a lot of energy needs to be spent on purging
- Thus master thread can be blocked to execute only purge activities for a long time
- This leads to master thread not properly flushing dirty pages, not doing checkpoints regularly as it should

InnoDB Purge Activity (2)

- When master thread isn't taking care of flushing dirty pages regularly eventually the execution threads will take over this task
- This will drastically lower performance for a short time until flushing is under control again
- So the problem with master thread leads to very high variance in throughput

InnoDB Purge Activity (3)

- Solution is to separate Purge activity into its own thread
- This means that master thread will always properly handle flush activities and checkpointing activities
- This will lead to steady performance
- Separation of purge into its own thread can lead to short-term performance to be lower since avoiding purging means higher performance for a short time, but if situation persists the purge is needed and can then kill performance of the MySQL Server

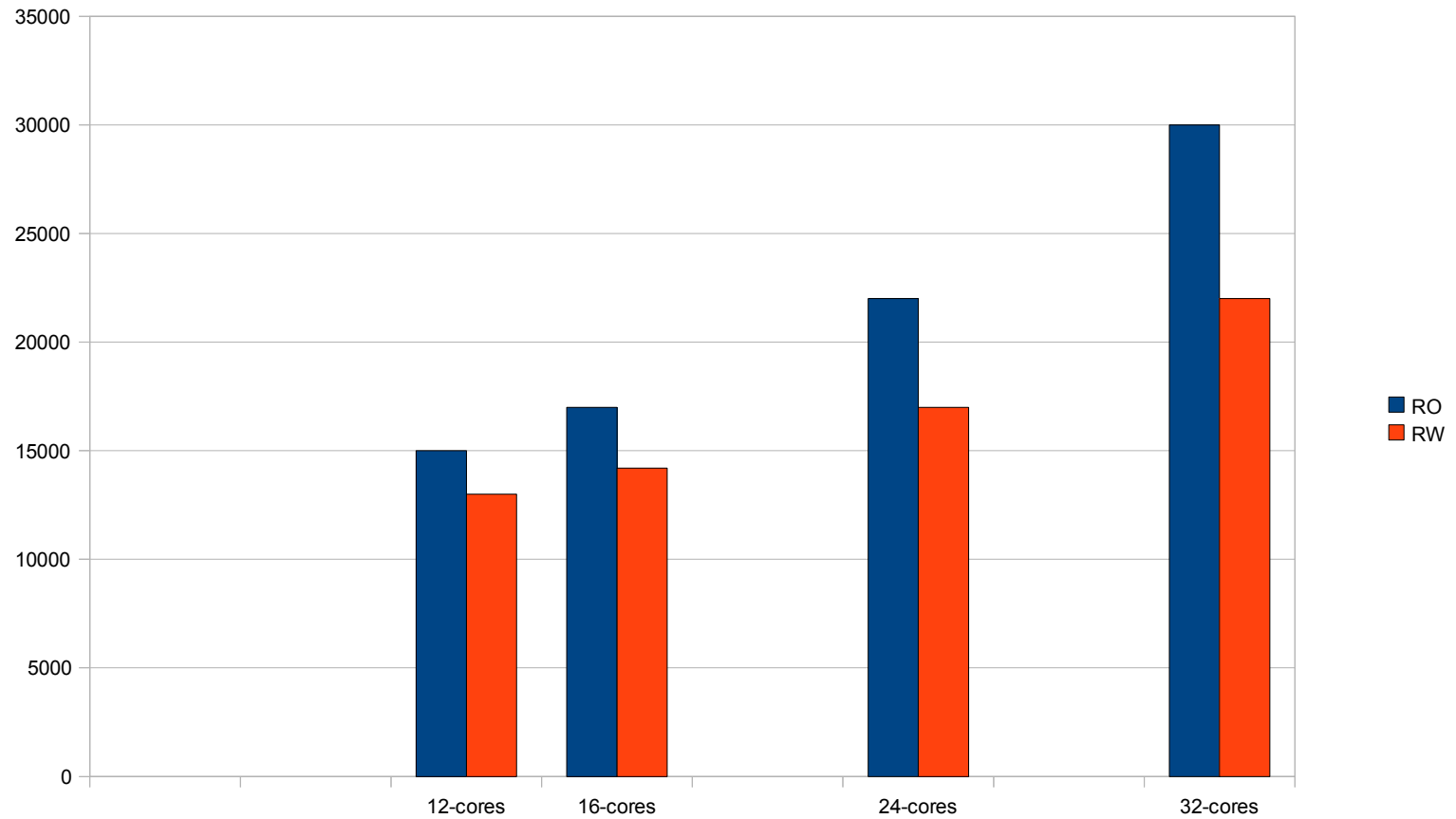
InnoDB Recovery (in 1.0.7 and in MySQL 5.5.4-m3)

- Phase 1: Redo Log Scan
- Phase 2: Redo Log application
- Phase 3: Rollback of uncommitted trx's
- Redo Log Scan contained operation to calculate size of heap which was made $O(1)$ from $O(n)$
- Redo Log application improved insert into buffer pool flush list by changing from linear search to binary search => $O(n*n)$ changed to $O(n*\log n)$
- Standard sysbench recovery improved from 7 hours to 14 minutes

Extending Change Buffering to also support Delete + Purges

- Buffers deletes and purges and perform those in background
- For Delete only benchmark on IO bound workload the delete rate can increase from 50/sec to 8000/sec

dbStress scalability 1 thread per core 12->32 cores





Thank you for your attention

Questions?

Welcome to the next session where we will present more detailed benchmark results