The Road to Direct Lookups is Paved with Good Intentions

A MySQL performance case study

Alex Boyd and Davide Romani
Demonware Ltd., Activision Publishing, Inc.

At Demonware we provide the backend technology and services platform for the Call of Duty franchise and other popular Activision titles. Hundreds of million gamers generate a considerable amount of data that we store in a variety of databases.

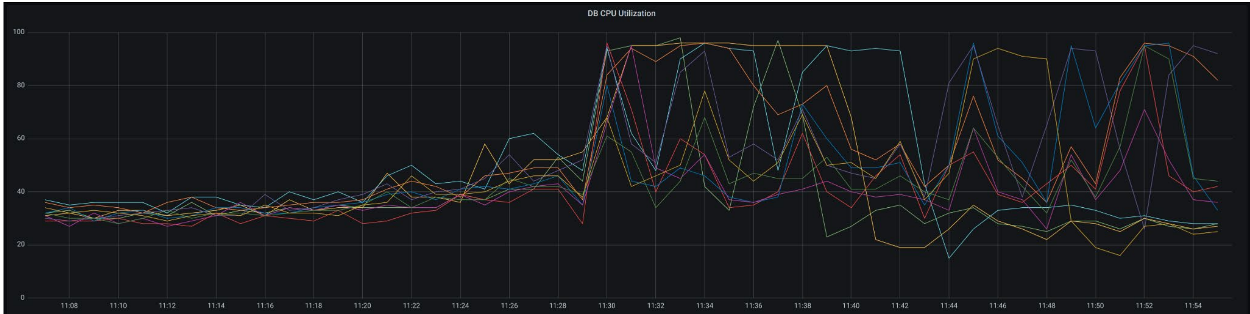
While certain backend transactions involve sagas (sets of transactions and compensating actions) and eventually consistent updates, others are easily implemented by using a single database transaction. For cases like this, some of our services leverage Percona MySQL to guarantee that their operations are atomic, consistent, isolated and durable.

Over the years, the number of MySQL instances in our fleet has grown substantially, and we pride ourselves on having developed a highly respectable level of expertise in fine-tuning our databases. However, every now and then, a humbling experience reminds us that favoring consistency makes it harder to guarantee availability, and that there is always something new to learn.

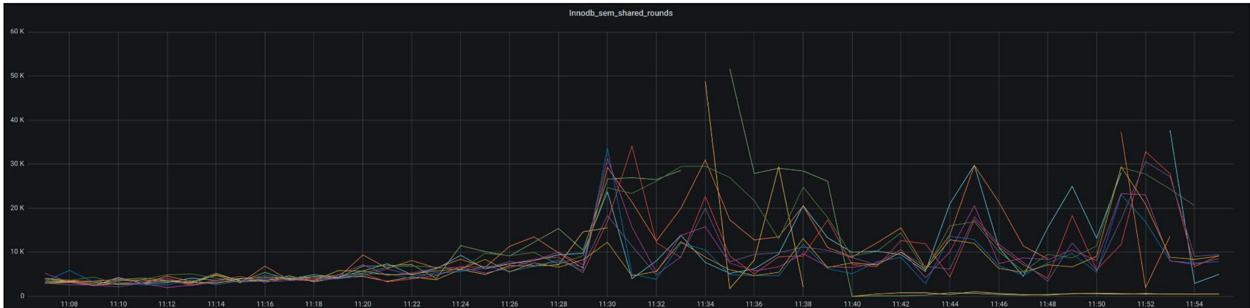
One such learning experience happened recently. We ran into a performance issue with the MySQL data layer in one of our services that perplexed us. At first, understanding the issue seemed almost intractable, but by using hypothesis testing experiments, we were able to reproduce the issues in synthetic tests, and eventually find a simple change to make that cleared the issue up entirely. The following is a full account of that story.

The incident

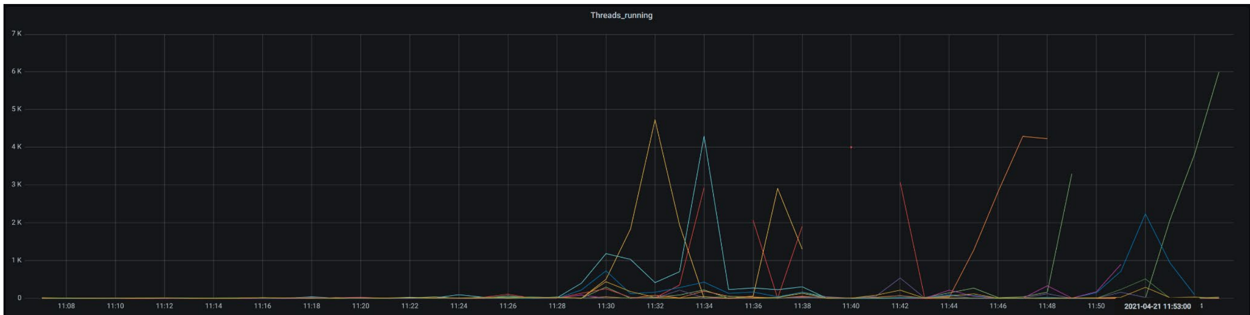
One day, one of our services started struggling, despite having sustained higher loads in the past. Some of its MySQL instances unexpectedly started saturating their CPUs:



Another metric revealed that, for those hosts, lots of CPU cycles were being wasted in "spin-rounds", actively waiting for locks:

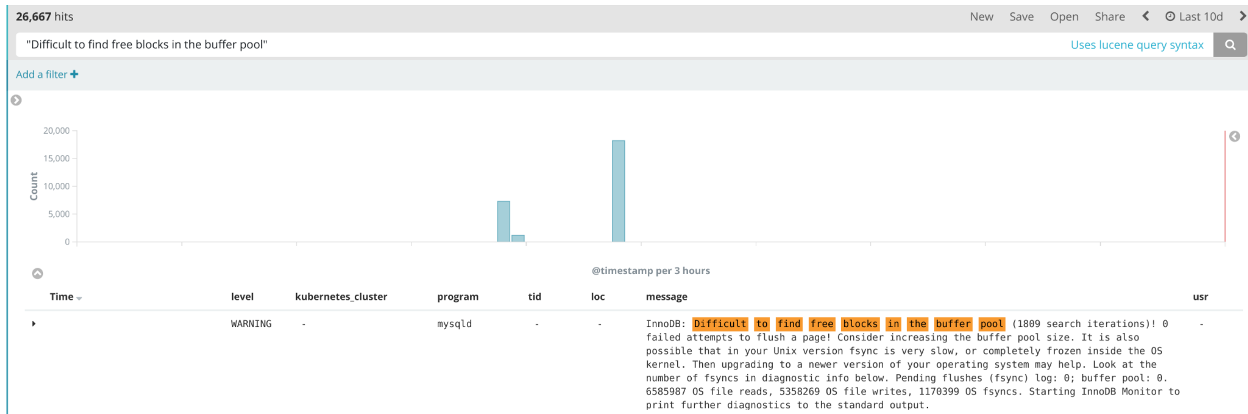


Connections started piling up, and those databases were eventually overwhelmed:



Restarting a “hot” MySQL instance relieved that shard for one to two hours, but then the same problem occurred again and again. This behavior was new, something we had not experienced during our load test sessions. Another thing we had not seen before were thousands of logs reporting the following error:

```
InnoDB: Difficult to find free blocks in the buffer pool
```



Investigation

The service that was encountering those errors was a Python application running in Kubernetes, with about **8 thousand single-process workers**. Its backend consists of about **100 shards** using Percona Server for MySQL 5.7 on primary and replica nodes. This service currently uses a MySQL thread per connection model. Our idea was to iterate on load testing until we could **reproduce** the issue seen in production.

As a starting point, there were some **clear symptoms**:

- Everything is fine until it isn't
- Within one minute, a single shard will jump from 50% CPU utilization to 100%, with a corresponding increase in query latency from ~150 ms to several seconds
- MySQL hosts will start emitting warning logs about being unable to allocate a free buffer pool page

There was also a long tail of **less obvious symptoms**:

- Available free buffer pool pages dropping below the 1k per buffer pool instances
- A spike in connections / threads running
- Spikes in many lock spins / lock waits metrics
- A spike in row lock metrics
- A spike in aborted client connections and new connections being established
- Some InnoDB status dumps from a MySQL host that we captured during the incident showed that a large number of threads were briefly stalling for less than 1 second over and over again, hitting some form of semaphore locking contention

With the above points in mind, we formulated the following **hypotheses**:

1. Some unknown performance impact caused a spike of client reconnections, and these reconnections added enough load to the system to cause further overload and connection cycling. Once this reconnect storm started, the database was never stable enough to recover at high load.

2. There was a bottleneck in one IO-bound subsystem in MySQL, such as buffer pool pages allocation / deallocation / cleanup and reuse, and once that got overloaded and backed up, all threads started spinlock waiting on that subsystem.
3. There was a bottleneck in a single threaded / non parallelized subsystem in MySQL. Once these threads were at their limits of performance, other threads all started busy waiting without making progress on the locks they were trying to acquire.

Then we started asking ourselves:

- Can we artificially trigger a reconnect storm by restarting our MySQL processes while they are under load in a load test?
- Can we simply increase the load until we see failures and hit the same bottleneck that caused the issue in production?
- Can we run a load test that puts extra pressure on the buffer pool to see if this behavior is reproducible with that kind of workload?

Probing hypothesis 1

As a first pass, we wanted to design some experiments to see if this was purely a symptom of how the system fails when the database or some of its subsystems is overloaded. Our intention was to put enough load on the system and check whether that alone would trigger this complex failure.

Initially we pushed the “raw read” and “raw write” load, as measured both by the volume of end-to-end requests, but also by a variety of internal InnoDB metrics, like number of rows read, updated and inserted. Even though **the tests reached throughputs well above the breaking points observed in the incident**, we were unable to reproduce any issues here, and the app tier in front of the databases became a bottleneck in every case before the MySQL databases themselves showed any symptoms of overload.

Later we manually ran a few computationally expensive table scan queries while the system was put under pressure by a load test. The goal was to **forcibly overload the database** and see how things failed when it had to sustain a higher workload than it was able to handle.

The combined load was able to cause timeouts in the queries, and make connections from the application to MySQL fail, but **the failures looked very different** from how things had been in production in the scenario we were trying to reproduce. Although this test was able to generate high volumes of aborted clients and reconnects, which we thought might be enough to cause self-sustained thrashing, the databases and application layer were always able to recover on their own once we stopped sending the heavy queries, we had used to destabilize the system. This led us to **drop our confidence in hypothesis 1**.

Probing hypothesis 2

The previous two tests led us to believe that the issue wasn't purely due to overloading the database, but rather to something more specific that failed and thrashed in a particular way.

We started to try and design some experiments to **see if we could put pressure on the buffer pool**. One symptom we had seen in production was that the number of free pages dropped below the configured 1k reserved pages per buffer pool instance, and also the message "`InnoDB: Difficult to find free blocks in the buffer pool`" appearing in the logs.

Our next few tests were all along these lines. The actual user data in these databases was a lot more artificial and uniform in our tests as compared to production. We hypothesized that our synthetic tests data might be less fragmented and packed into fewer buffer pool pages, even for the same total data size. We confirmed that by comparing the result of the following query in the production environment and the load test one:

```
SELECT
    table_name,
    index_name,
    COUNT(*) as page_count,
    SUM(data_size)/1024/1024 as data_size_in_MB
FROM information_schema.innodb_buffer_page
GROUP BY table_name, index_name
ORDER BY data_size_in_MB DESC;
```

To better simulate actual user data, we created a handful of different shards with various buffer pool tunings. We reduced the overall **buffer pool size** on some and overlaid a **backup of production data** mapped onto the user ID space of our load test users on another. With some of these changes, we were able to see more data being paged in and out to disk, in a volume that was more in line with the hosts in production. We went even further and used what we learned from these changes to try and intentionally create a setup where we put massive pressure on the buffer pools to see if we could hit any sort of bottleneck where the buffer pool page cleaner wouldn't be able to keep up.

We found that, even after pushing past the high-water marks of the paging in production prior to impact, the load test setups could keep up just fine. It didn't appear as if this paging mechanism by itself was the bottleneck we were looking for. Between this and the earlier tests pushing to very high read and write volumes, this dropped our confidence in hypothesis 2; that the root cause was that we simply hit some bottleneck in buffer pool management or raw IO.

Probing hypothesis 3

The failure of the first two attempts left us with hypothesis 3, that we were dealing with an issue related to **lock contention**. One thing that stood out to us was that in production we had a continuous background rate of **spin-lock rounds**, and even some **OS lock waits**, and that those increased to very high levels when we saw impact. This was one of the symptoms that we had not managed to reproduce in any of our tests so far. Even the background rate of semaphore lock contention was next to nothing in all our synthetic tests, while it was always high even at low loads in production.

We figured that there must have been some queries run in production that were highly impactful at triggering some sort of lock contention, which were missing in our synthetic tests. We were a bit stumped and set the test cluster aside for a few days to be used for another person while we used tools to go through all the queries on the production systems with a fine-toothed comb to try and spot any major differences to what was hitting the databases in our tests.

The smoking gun

Luckily for us, another team started to load test some new features that were in development that put some load on this service, and they **started to reproduce the issues** we found in production even at much lower scales: the same surge in CPU usage, client errors, number of threads running, and short-lived semaphore waits. This gave us the evidence we needed, and we observed the queries this service generated on MySQL. We found one similar query that was present in production environments and during these tests, but that had been absent in the earlier load test reproduction attempts. The query looked like this:

```
SELECT entity_id, player_id, account_type, quantity, …
FROM my_table
WHERE account_type = ?
AND player_id = ?
AND quantity > ?
AND entity_id >= ?
AND entity_id IN (?)
```

This query was pulling down pages of entities associated with a player, using pagination and filtering. The final “IN” clause to filter over a set of entity IDs was the crucial difference that was missing in our earlier tests. This was a call pattern done relatively frequently in our production workloads, but our particular load test simulations always just pulled down all the pages. The entity id column was part of the composite primary key, so in theory filtering down for fewer pages of just the subset of rows we cared about should be an optimization. We went

back and updated the load test simulation to add in this extra filtering, and we were immediately able to reproduce the symptoms.

This solidly pointed to our **hypothesis 3 as the culprit**: there was some subsystem that was bottlenecking everything else, and as far as we could tell it only showed up when we hit this particular pattern of reads. Our next step from there was to **dig into these InnoDB status dumps**, see if we could make any sense of the contention on these semaphore locking patterns, and determine where in the internals of MySQL we were hitting the bottleneck. To do this, we followed roughly the same process as others have done and blogged about before, such as this Percona post:

https://www.percona.com/blog/2019/12/20/contention-in-mysql-innodb-useful-info-from-the-semaphores-section/

Chains of locks

In some of our very worst dumps we had close to 1000 threads all waiting for under 1 second. The range of location in code with stalls, and the type of lock they were waiting on, were quite varied. Each entry in an engine innodb db dump looked like the following:

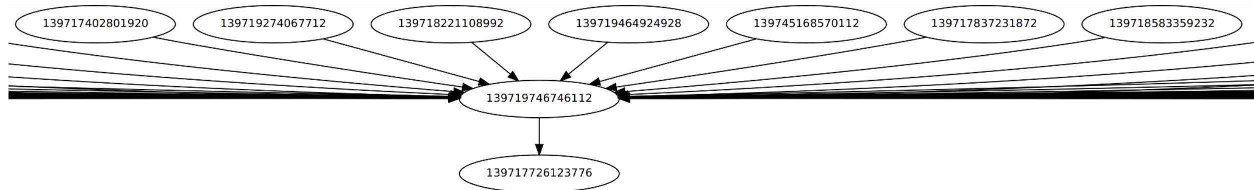
```
----------
SEMAPHORES
----------
...
--Thread 139719055636224 has waited at row0ins.cc line 3023 for 0  seconds
the semaphore:
...
```

To demonstrate the complexity of this locking pile up, here's the aggregated counts of hundreds of such entries from one particular dump:

```
count   lock
-----   ----------------------------
  1     btr0cur.cc line 329 X-lock
  1     btr0sea.ic line 136 S-lock
  1     mtr0mtr.ic line 161 X-lock
  1     trx0purge.cc line 163 Mutex
  2     btr0cur.cc line 4068 X-lock
  2     row0ins.cc line 2520 S-lock
  2     trx0roll.cc line 1007 Mutex
  4     trx0undo.cc line 1804 Mutex
  5     btr0sea.cc line 1291 S-lock
  5     btr0sea.cc line 1388 X-lock
  5     trx0trx.cc line 2029 Mutex
  8     fsp0fsp.cc line 3539 X-lock
 12     btr0sea.ic line 98 X-lock
 24     trx0trx.cc line 1201 Mutex
```

```
 48     btr0cur.cc line 2084 S-lock
221     btr0cur.cc line 1019 SX-lock
222     row0sel.cc line 5025 S-lock
948     row0ins.cc line 3023 S-lock
```

We also found that if we ran a **read-only workload** with these filter queries, that it was sufficient to hit issues with similar symptoms, though with a much smaller set of semaphores under contention. So, we focused on those ones first. It seemed that the main bottleneck was due to **various select and buffer pool operations all being blocked on a thread** holding the following exclusive lock:




```
--Thread 139717726123776 has waited at btr0sea.ic line 98 for 0  seconds the semaphore:
X-lock on RW-latch at 0x2db49f8 created in file btr0sea.cc line 238
```

On close investigation of the chain of locks reproduced with a mix of read and write load, we discovered that the vast majority of blocked threads was linked to this `btr0sea` lock.

The threads were not always linked directly, but we were seeing other threads that were holding onto other sets of locks, and then were stalling out waiting on this lock as well. This seemed to lead to quite **a massive amplification of contention.** This single lock could block other threads that are holding locks, which in turn could prevent progress on a wide variety of operations that don't directly need that `btr0sea` lock.

What is this btr0sea lock?

Looking at btr0sea.ic line 98 in Percona's source code, we found:

```
/** X-Lock the search latch (corresponding to given index)
@param[in]  index index handler */
UNIV_INLINE
void
btr_search_x_lock(const dict_index_t* index)
{
      rw_lock_x_lock(btr_get_search_latch(index));
}
```

The function definition for btr_get_search_latch is in the same file and it states:

```
/** Get the adaptive hash search index latch for a b-tree.
@param[in]  index b-tree index
@return latch */
UNIV_INLINE
rw_lock_t*
btr_get_search_latch(const dict_index_t* index)
{
...
}
```

That's how we discovered that `btr0sea lock` is used as part of the adaptive hash index subsystem. The **adaptive hash index (AHI)** is MySQL's attempt to build a direct lookup table of the most common query patterns, allowing for extremely fast lookups that bypass the need to traverse the tree-based indexes entirely. Why would this be a problem when we have these read queries using the "IN" filtering in their WHERE clause?

As far as we can tell, this pattern of calls was triggering AHI to constantly rebuild itself to try and produce direct lookups for the individual rows that matched the entity id filter and bypass traversing that b-tree index. However, **the cardinality on the number of different rows we were hitting with those lookups was massive**, and we would have never been able to fit the entire set of lookups into the adaptive hash index, so the operation wasn't providing much value, but it was **constantly triggering rebuilds**. It turns out that when this rebuild process is happening, the `btr0sea locks` show up, and that they can block many other high frequency buffer pool and index management operations.

Our best understanding of the situation is that a rebuild would kick off and trigger short lived stalls across many other processes on the MySQL host. This would in turn cause a huge spike up of threads, all spin-locking, waiting for this lock or others held by threads that were themselves waiting for this lock.

As this setup uses one thread per connection, rather than a pool of threads, the number of threads all spin-locking can pile up quickly from dozens of threads running during normal operation, all the way up to multiple hundreds of threads. This in turn maxes out the CPU on the box, and further leads to the slowdown of the threads which are holding the locks as CPU time becomes a scarce resource. If the host ever gets to that bad of a state, it freezes up and never recovers, as the backlog of work becomes so large that the system continues to hit the locking contention constantly while making little progress due to a lack of CPU.

We believe that symptoms such as lack of free buffer pool blocks are collateral damage from this lock contention. We think the page cleaner was making little progress due to lock contention stalls. Meanwhile, there appeared to still be other code paths that kept consuming more blocks until the free pool was exhausted.

This also appears to be in-line with some other findings elsewhere in the MySQL community around performance impacts of the adaptive hash index. We found that the MariaDB fork has swapped to the default of having the AHI disabled. While this was a very different workload they had used in their own testing to ours, many of the reasons they have cited in their decision to disable it by default could potentially apply to our situation:

https://jira.mariadb.org/browse/MDEV-20487

Disabling the adaptive hash index

Our new hypothesis is that the adaptive hash index rebuilds are the first domino that falls. To test this, we simply took our now reproducible test case and re-ran the same load test simulation, but with the adaptive hash index turned off on the MySQL host. The MySQL hosts were barely breaking a sweat once AHI was disabled. The locking contention and pile up did not reproduce, and even the background rate of spin-locking we saw under healthy operations was at near zero with a rather large reduction on CPU usage for the entire workload.

With the AHI off, we also tried to push the service to its limits to find the new breaking point. As far as we could tell with our load testing simulation, we were able to push MySQL to over 80% CPU usage on each shard without seeing any user impacting failures. Instead of the previous freezes we were seeing at sub 50% CPU usage, and as our background CPU usage appears to be much lower with less spin-lock contention, this meant that we were able to push to more than triple the rates where we had previously been hitting issues. A single setting that we can flip to triple our throughput and to remove a thrashing feedback loop from our system is just about the best outcome we could have hoped for at the start of this process.

We examined a handful of metrics to determine how much the AHI might be helping and similar to how it was measured in this Percona blog post:
https://www.percona.com/blog/2016/04/12/is-adaptive-hash-index-in-innodb-right-for-my-workload/.

As a result, we slowly pushed this change across our production hosts for this service and are very happy with the results now that it's fully rolled out. Since then, we have passed those high-water marks that were previously causing the system to buckle, without any signs of issues.

Lessons learned

Investigating distributed systems at this scale can be closer to science than to engineering. In some cases, there are simply too many variables and interrelated complex systems to work your way up from first principles by understanding every single line of code and what every process is doing at all times. We found it much more valuable to use a scientific approach where we formulated testable hypotheses and used these to design experiments to learn about the nature of the system. This investigation eventually helped us track down an issue in production all the

way to the behavior of the AHI, which was a great example of this style of investigation leading to concrete results.

One last oddity we found in the analysis

Here's a final quirk we don't understand that showed up during our analysis of the locking relationships. We found this a few times in our innodb engine dumps, but by itself, it didn't seem to be an issue. Nevertheless, this seems to suggest a case where a thread is blocking itself due to a lock it already holds. If you can explain what is happening here, we want to hear from you. Contact us at careersDW@demonware.net.:

```
--Thread 139718536591104 has waited at mtr0mtr.ic line 161 for 0  seconds
the semaphore:
X-lock (wait_ex) on RW-latch at 0x7f2162eaed30 created in file buf0buf.cc
line 1433
a writer (thread id 139718536591104) has reserved it in mode  wait exclusive
number of readers 1, waiters flag 1, lock_word: ffffffffffffffff
Last time read locked in file row0ins.cc line 2520
Last time write locked in file /mnt/workspace/percona-server-5.7-redhat-
binary-rocks-new/label_exp/min-centos-7-x64/test/rpmbuild/BUILD/percona-
server-5.7.29-32/percona-server-5.7.29-32/storage/innobase/btr/btr0btr.cc
line 187
```

© 2021 Activision Publishing, Inc.