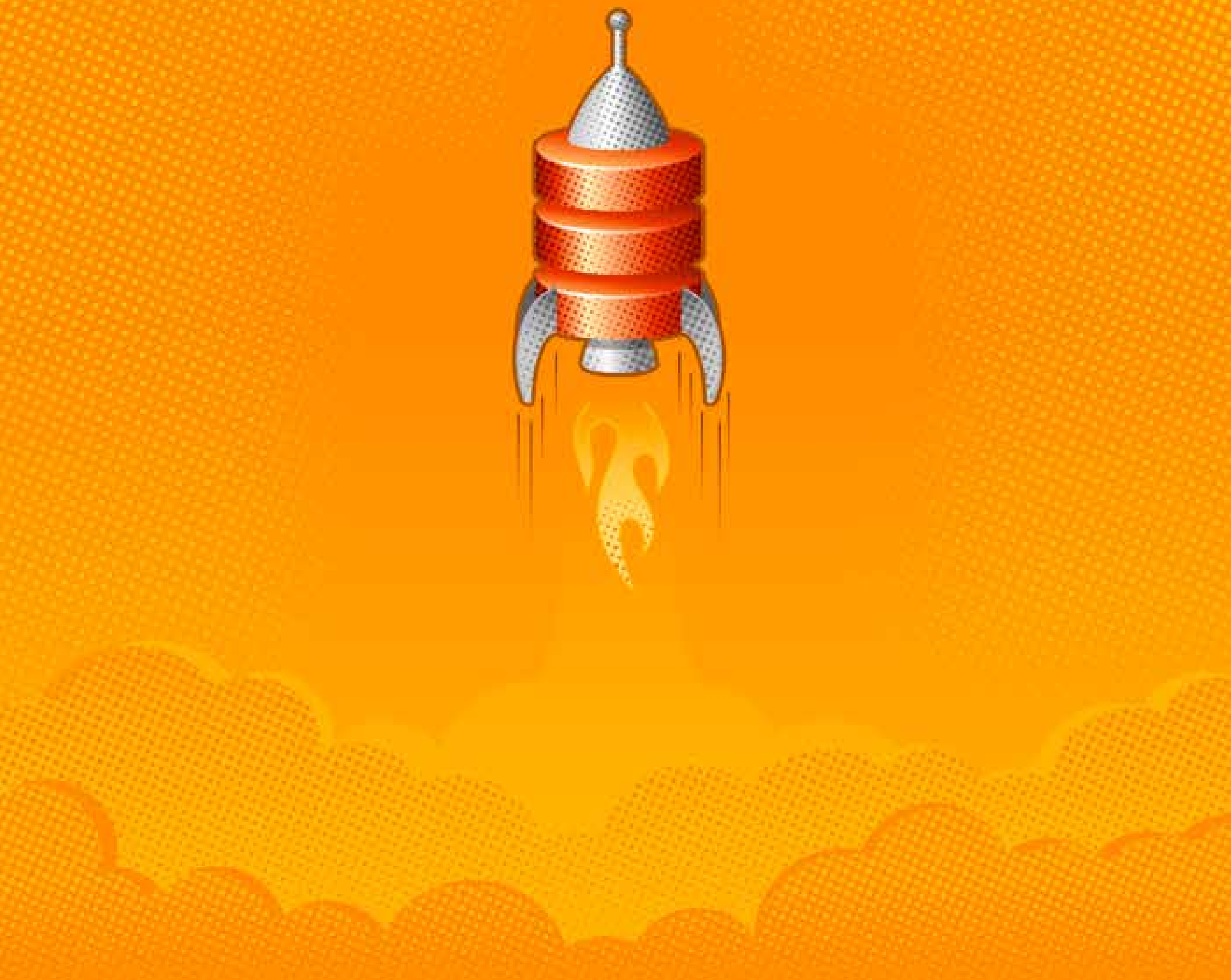


PRACTICAL

MySQL® Performance Optimization

A hands-on, business-case-driven guide to understanding MySQL® query parameter tuning and database performance optimization.

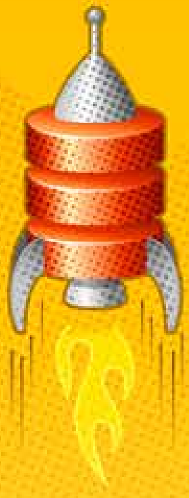
PERCONA

With the increasing importance of applications

and networks in both business and personal interconnections, performance has become one of the key metrics of successful communication. Optimizing performance is key to maintaining customers, fostering relationships, and growing business endeavors.

A central component to applications in any business system is the database, how applications query the database, and how the database responds to requests. MySQL is arguably one of the most popular ways of accessing database information. There are many methods to configuring MySQL that can help ensure your database responds to queries quickly and with a minimum amount of application performance degradation.

Percona is the only company that delivers enterprise-class software, support, consulting, and managed services solutions for both MySQL and MongoDB® across traditional and cloud-based platforms that maximize application performance while streamlining database efficiencies. Our global 24x7x365 consulting team has worked with over 3,000 clients worldwide, including the largest companies on the Internet, who use MySQL, Percona Server®, Amazon® RDS for MySQL, MariaDB® and MongoDB.



This book, co-written by Peter Zaitsev (CEO and co-founder of Percona) and Alex Rubin (Percona consultant, who has worked with MySQL since 2000 as DBA and Application Developer) provides practical hands-on technical expertise to understanding how tuning MySQL query parameters can optimize your database performance and ensure that its performance improves application response times. It also will provide a thorough grounding in the context surrounding what the business case is for optimizing performance, and how to view performance as a function of the whole system (of which the database is one part).

THIS BOOK CONTAINS THE FOLLOWING SECTIONS:

- **Section 1**: *Application Performance and User Perception*
- **Section 2** *(this section)*: *Why is Your Database Performance Poor?*
- **Section 3:** *MySQL Configuration*

For more information on Percona, and Percona's software and services, visit us at www.percona.com.

ABOUT THE AUTHORS

Peter Zaitsev, CEO

Peter Zaitsev is CEO and co-founder of Percona. A serial entrepreneur, Peter enjoys mixing business leadership with hands-on technical expertise. Previously he was an early employee at MySQL AB, one of the largest open source companies in the world, which was acquired by Sun Microsystems in 2008. Prior to joining MySQL AB, Peter was CTO at SpyLOG, which provided statistical information on website traffic.

Peter is the co-author of the popular book, High Performance MySQL. He has a Master's in Computer Science from Lomonosov Moscow State University and is one of the award-winning leaders of the world MySQL community. Peter contributes regularly to the Percona Performance Blog and speaks frequently at technical and business conferences including the Percona Live series, SouthEast LinuxFest, All Things Open, DataOps LATAM, Silicon Valley Open Doors, HighLoad++ and many more.

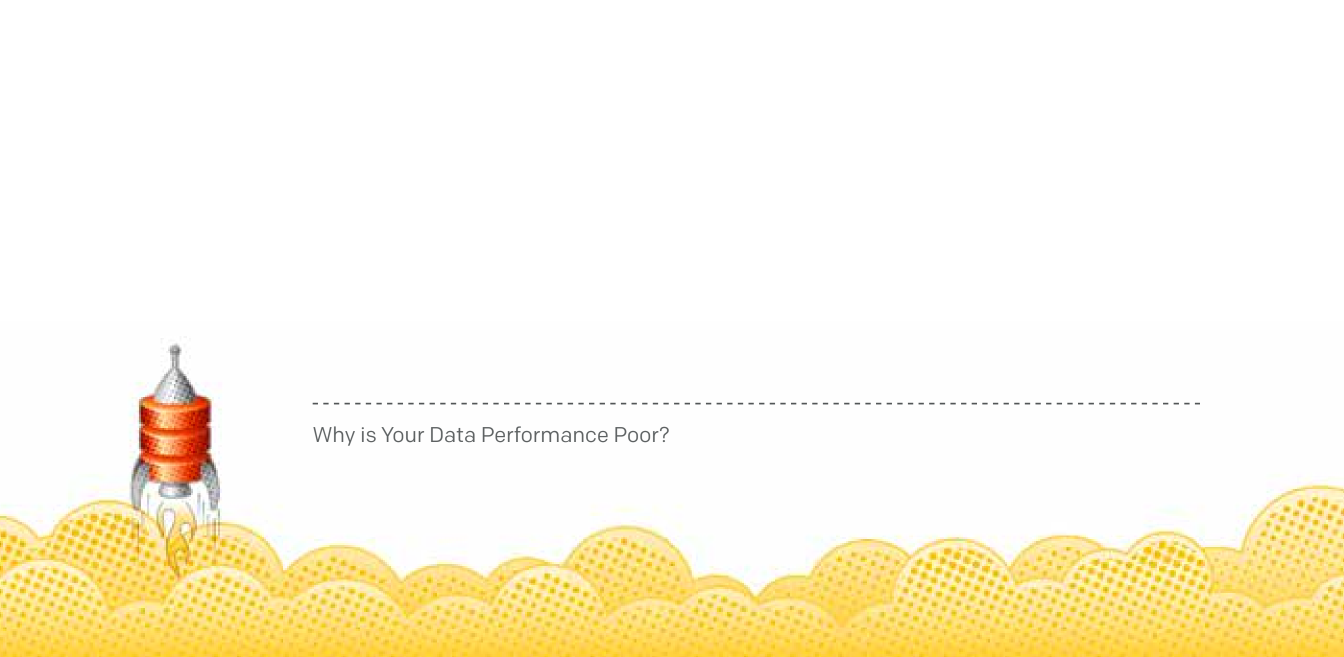
Alexander Rubin, Principal Consultant

Alexander joined Percona in 2013. Alexander worked with MySQL since 2000 as a DBA and Application Developer. Before joining Percona he was doing MySQL consulting as a principal consultant for over seven years (started with MySQL AB in 2006, then Sun Microsystems and then Oracle). He helped many customers design large, scalable and highly available MySQL systems and optimize MySQL performance. Alexander also helped customers design big data stores with Apache Hadoop and related technologies.

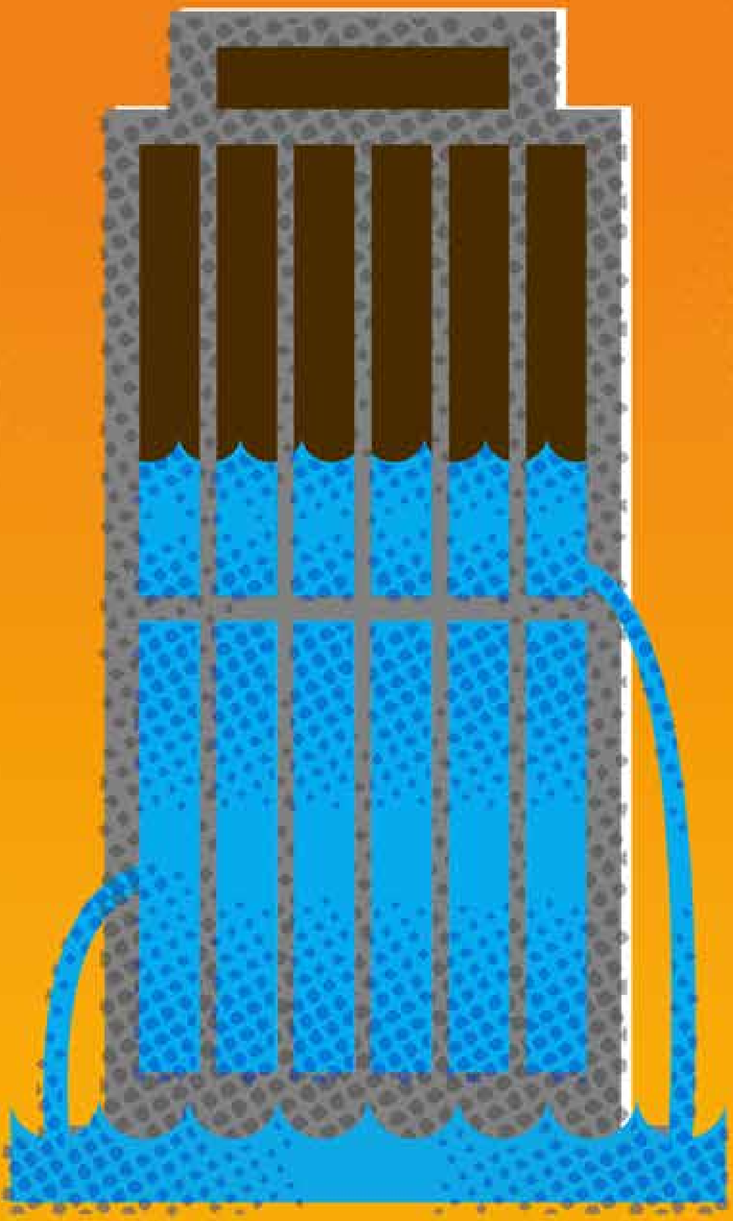
Why is Your Database Performance Poor?

WHY IS YOUR DATA PERFORMANCE POOR?

To achieve great application performance, you often must troubleshoot performance problems. To do this, you need to understand what causes poor performance in the first place. Correctly understanding the true cause of performance problems allows for a quick and efficient resolution—yet often this crucial information is lacking. Not fully understanding the root cause of poor performance means that the solution could require more time and resources than are necessary, or that the solution used to address the problem isn't the most efficient one. For example, throwing more hardware resources at a performance issue may increase performance, but it doesn't address the performance issue itself. It's important that your solution isn't just a way to mask the problem, but actually resolves it.



Poor Architecture Design and Implementation



POOR ARCHITECTURE DESIGN AND IMPLEMENTATION

Performance problems often stem from either poor architecture design, or poor design implementation. Architecture-based performance issues are a worst case scenario, because they are very expensive to change. With poor implementation issues, the architecture-level ideas might be sound but get badly implemented in the code. Implementation issues are usually easier to address.

The biggest architecture design problem is picking the wrong architecture for the scale of your application. If you're designing a small scale application, you might do very well with MySQL alone—no caching, sharding or replication needed. At "Facebook" scale however, the same functionality will require these and many more techniques. Understanding the scale of your application and choosing the right architecture can be the difference between successful performance or failed performance.

Another common problem is using MySQL for tasks it isn't good at. The world of databases, even open source databases, is a lot different now than ten years ago. You don't need to suffer anymore with MySQL's built-in, full-text search functionality—you can use ElasticSearch® or Sphinx. No more writing scripts to process portions of your log files on many replicas, and then aggregate the results—you can use Hadoop® or Vertica™ for that. Redis®, MongoDB, Cassandra™ and others all have excellent sweet spots, which is why we often see systems running at scale using more than one database system.

This isn't necessarily a cause for over excitement, however. Combining different database systems (and different components in general) means the whole system itself becomes more complicated. Each system requires skills in development and

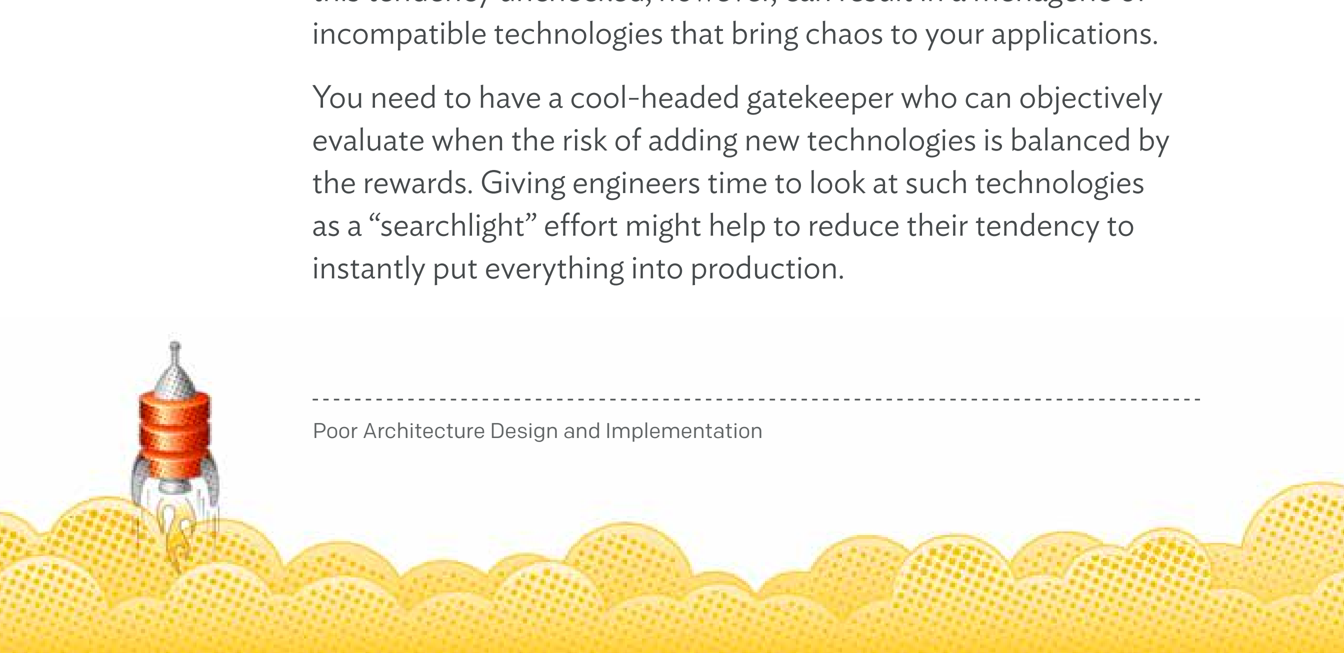
operations, which is hard for smaller teams to master and maintain at a reasonable depth. We should listen to Albert Einstein, who said “Everything should be made as simple as possible, but not simpler.”

Applying this principle, you should use MySQL (or another database system) where it works reasonably well, and add other systems (especially complicated ones) only if it helps to achieve significant gains. Following online rumors that “Technology X is better at Y” is a bad idea when designing databases.

The following example is a good illustration. I visited a customer who was storing logs in MySQL. One of their developers heard that Cassandra was much better for storing logs. I asked them how many records are they going to store, and they told me some 10,000 a day. At this small scale, any solution would work just fine. I advised them to stick with MySQL. Six months later I visited them again. We needed to examine those logs to get the history for some diagnostics. It turns out they ignored my advice and implemented Cassandra for log storage. The problem was the only developer who knew Cassandra had since left, and they had no way of retrieving those logs for us to analyze.

A special note here: it is natural for engineers, both in development and operations, to want to play with shiny new technologies (it’s part of the innovative value they bring to any organization). Leaving this tendency unchecked, however, can result in a menagerie of incompatible technologies that bring chaos to your applications.

You need to have a cool-headed gatekeeper who can objectively evaluate when the risk of adding new technologies is balanced by the rewards. Giving engineers time to look at such technologies as a “searchlight” effort might help to reduce their tendency to instantly put everything into production.



Finally, I should mention the database schema. The database often require major changes to applications. You need to ensure that the database schema does more than represent the data you need presented—it must also efficiently perform the operations your applications require. This applies to both reads and writes, either of which can become bottlenecks.

Over time, application requirements are going to change: new data means new features get added and old ones removed. Often as this evolution occurs, you are under production constraints and don't have the time for proper database design. Instead, you are often faced with "duct-taping wings on a pig" to get it to fly. Many of these ad hoc solutions can get databases off the ground, and I would even go so far as saying a very large number of applications we use daily are powered by such "flying-pig" databases. However, I wouldn't say they fly very well. It is important to take a hard look every so often at your database schema, and honestly evaluate if it is still optimal for your requirements. It never hurts to perform some cleanup and do some optimization. This will not only improve performance, but will make applications easier to develop and maintain—increasing both development and operations productivity.

Poor Implementation

We've just shown some examples of poor application design. What about implementation? There are many ways to introduce chaos by doing the right things, but in the wrong way! Even a good schema can employ bad queries that don't use indexes, use too many queries, or have queries that fetch too much data. Let's look at a couple of examples.

A problem we often encounter is developers thinking about the database like it's a "data store" inside their application: if they need

a chunk of data, they go ahead and order it from the database. This means that if an application is going to present a table, for example, it issues separate queries for every cell in the table. A much better solution typically is to have one SQL query retrieve the entire table.

An application loop is a typical example of the above. In the application, we see something like this:

```
$id = 1;

while ($id < 1000000) {

  ...select * from table where id = $id ...

}
```

This will end up running one million queries, which is significantly slower than running one query to get the whole table. The main reason for the delay is the high overhead of a single query execution: MySQL will need to receive the SQL query, parse it, then read the index, then send the result and then do it all over again for the next query. Even if each query is fast (e.g., one millisecond) the total amount will add up to be one million milliseconds, or approximately 16 minutes. It is much faster to read the whole table in one shot.

Note. Sending the whole table to the application could be a bad design as well. It is fine when the application function requires a cache, but otherwise it is probably more efficient to come up with a query that filters or processes data inside MySQL and avoids sending the results back to the application.

Another example of bad design implementation is using MySQL for things you could do in your application. We've seen things like getting the current date, time, converting the timezone and other

activities done on the database side using queries, when they could just as easily be done on the client side (without impacting performance).

Fetching too much data from the database can be another problem. I've seen people run queries like:

```
"SELECT * FROM   topscores ORDER BY score
DESC"
```

to fetch the top ten scores. This query fetches more than the top ten scores. It fetches all the scores, orders them, presents the top ten and then discards the rest. MySQL, unlike other databases, always provides full query results in cases like the above. Even worse, the result might get sent to the client in its entirety and buffered completely in the client's memory.

Don't do this! If you only need ten rows, use the LIMIT command.

I find that developers often have a hard time understanding query latency, or accounting for it properly. Many developers work on their laptops, where latency from the application to the database is much lower than in a production environment. If you are running your application in the data center with a 1Gb network, one switch between your application server and a MySQL server, the expected latency added by the network should be about 300 microseconds. If the MySQL server responds instantly, it takes the client 300 microseconds to get a response. This sounds pretty fast! And it is, as long as your page load or user interaction has a reasonable amount of queries. However, we have seen web pages that require up to 100,000 queries.

(Yes! 100,000 queries for just one page to load!)

Do the math: even if all these queries are simple, the page will take at least 30 seconds to load!

Poor Deployment

Even if it is well-designed and implemented, you must also deploy a system correctly. A lot of deployment issues are trivial: using the wrong hardware, the wrong instance type in the cloud, using an inappropriate amount of memory or spinning disks (spinning disks—yuck!) instead of flash-based storage: these types of issues are all typically caught quickly and easy to resolve.

Network-related issues are the biggest cause of most deployment-related issues. In other words, not thinking completely through network latency and network throughput.

Even in the same data center, adding multiple switches and some slow firewalls between the application server and the database can multiply latency. When you place the database server and application/web server in different availability zones, or especially in different data centers, the increase gets even more dramatic.

I remember a case where we observed very poor performance from one application. The application and MySQL servers weren't heavily loaded. Naturally we suspected the network was the culprit. When we asked about the network topology, the customer told us that "the connection between the application server and database is 1GB." As we pressed for more details, we learned this was a dedicated 1GB link between two data centers located more than 100 miles apart (not a local 1GB connection as we assumed—reminding us once again never to assume anything).

Note on the network speeds. It is important to understand the difference between network bandwidth versus network throughput versus network latency. In the above case, 1GB is the network bandwidth. This is the maximum rate of data that the channel can deliver, all things being equal. Network throughput is the actual rate of data delivered, taking into account the inhibitors to the data flow. Network latency is the delay between sending and receiving the data.

A good analogy for network bandwidth, throughput, and latency is a highway. A good interstate highway may be six lanes wide (1GB bandwidth) and have the ability to move six trucks worth of "cargo" at 75 miles per hour. But if the highway is in poor condition, that speed will be reduced (throughput), and if the distance between the two points is very far (100 miles) the delivery time may be significantly longer even if we run six big trucks at the same time (latency).

Network latency will rapidly spike, and make response times worse, both in cases of network error (such as packet loss) or network saturation. Even a 0.1 percent packet loss rate is a big deal! If your application is sensitive to latency, don't use the network at more than 50 percent of capacity.

Network performance can vary a lot, especially in the cloud and other cases of software defined networking (SDN), where the exact network topology can be very fluid. It's always best to monitor the network latency between your database server and client, so that you can quickly see any unusual latency increases.

The simplest tool to measure network latency is a ping tool:

```
root@ts140i:/var/lib/mysql# ping -c 5 -s 500
10.11.13.1
PING 10.11.13.1 (10.11.13.1) 500(528) bytes of
data.
508 bytes from 10.11.13.1: icmp_seq=1 ttl=64
time=27.1 ms
508 bytes from 10.11.13.1: icmp_seq=2 ttl=64
time=21.6 ms
508 bytes from 10.11.13.1: icmp_seq=3 ttl=64
time=21.6 ms
508 bytes from 10.11.13.1: icmp_seq=4 ttl=64
time=21.4 ms
508 bytes from 10.11.13.1: icmp_seq=5 ttl=64
time=21.4 ms
```

These results are from a wireless network, so we're observing a latency (response time) of 21ms (which would be extremely high for a data center environment).

Note. I'm using the -s parameter to specify the packet size. You can set this parameter to your typical query response size to see more relevant latency. The 64 bytes default packet size is smaller than most typical MySQL query/responses.

To identify how many routing devices stand between your application server and database, use the "traceroute" command:

```
root@ts140i:/var/lib/mysql# traceroute
10.11.13.1

traceroute to 10.11.13.188 (10.11.13.188), 30

hops max, 60 byte packets

1 test (10.11.13.188)   0.314 ms   0.322 ms

0.359 ms
```

In this case, we can see that "test" was directly accessible, without having to go through any routers. The traceroute command won't show you how many switches are in between the application server and the database, or show you more complicated details—such as if tunneling or a VPN use.

For interactive network latency observations, the "mtr" tool is very helpful. It combines the function of ping and traceroute in the same tool.

Another contributor to unexpected latency increases is the use of secure socket layer (SSL) with MySQL. SSL is wonderful for adding security, however it drastically increases latency times (especially the time it takes to connect to the MySQL server). If you enable SSL, remember to account properly for the latency impact on your database queries. Maintaining persistent connections might be an especially good idea when SSL is in use.

Now let's talk about network bandwidth. While latency issues are often clearly visible, network bandwidth issues are often not. In most cases, when it comes to database activity we see low bandwidth usage under a typical load, but a huge spike during batch jobs (which retrieve a lot of data and become network bound) or backup runs (often taking up lots of network bandwidth). The best way to spot these instances of latency due to bandwidth consumption is to set up and employ proper monitoring.

It’s important to note that it’s not just the last hop between your switch and database server that can be saturated—it could also be core links in your data center. We have often seen backups started on all hosts at the same time, which would saturate the network link to the filer responsible for storing the backups.

It’s situations like the above that demonstrate why cooperation between teams is key for capacity planning. To plan effectively, you need to understand both the objective requirements and the operational data—which is only achievable through a full transfer of information between the responsible parties. In far too many cases, we’ve found it extremely difficult for DBAs to get real information about the network topology or network statistics from the routers and switches.

If network bandwidth becomes the issue, good solutions are upgrading to faster network speeds or aggregating several existing connections. Using dedicated links for management traffic (such as backups) is another option. Both of these require planning and potential changes to network configuration.

As a DBA you have a couple of options completely in your control. First, you can control timing. Don’t run a huge number of batch jobs concurrently. Start backups at different times and with different hosts, and don’t run batch jobs and backups at the same time.

Second, consider using compression to reduce the amount of bandwidth required for backups. If you’re using streaming, you can also use the “pv” tool to set the rate limit.

Cloud platform and virtualization add their own share of deployment issues, with the first being that it is a shared

environment and the performance profile of everything ranging from the CPU, to disk IO, to the network configuration can change. Cloud and virtualization systems attempt to isolate and reduce performance variance from its neighbor's noise, but none of them are perfect. Typically, your best response is to plan for normal performance variance. In extreme cases, you might need to physically isolate some applications in virtualized environments, or move to different instances with a cloud provider.

Another issue we often see is improper technology use. For example, we had one customer who had set up Percona XtraDB™ Cluster in a virtualized environment where all the nodes ended up on the same physical host. So much for availability! In cloud environments, it's important to note that there's no guarantee you won't end up with the same issue—unless you explicitly request instances in different availability zones.

Poor Software Configuration

Poor software configuration is another issue that causes all kinds of problems ranging from application performance to downtime, and can include data corruption and data loss.

The configuration of any software components could be important to performance, and the problems caused by misconfigured software are too numerous to cover fully. Most components are going to be correctly configured most of the time, but just one misconfiguration might cause huge problems.

From the application standpoint, a misconfigured driver is a common MySQL-related issue. Configuring a Java connection pool to be too small, for example, can often cause low performance and application errors. Setting it too large, on the other hand, can cause connection errors, poor performance or even crashes. Setting

the wrong isolation mode in the connection settings, without understanding the implications, can cause the application to work incorrectly and potentially cause data corruption.

On the MySQL side, one mistake is simply running MySQL with the default settings. For historical reasons MySQL was not designed to automatically take on all resources of the physical host or virtual instance. It was in fact designed to take on very little of them on purpose. This means if you run MySQL with the default settings, you won't be able to open many MySQL connections—causing poor performance. You need to configure MySQL correctly to optimize performance. We will cover MySQL configuration advice in greater detail later.

You can also misconfigure other software components participating in MySQL interactions: we've seen issues caused by a misconfigured DNS, HProxy, load balancer, virtualization software or the Linux kernel. Unlike MySQL, the Linux kernel provides a very decent default configuration and requires extra tuning only for most unusual or demanding applications.

Poor Data Design

Data design issues are a very common cause of problems. Data design problems can be the source of both the most significant gains (we've seen up to 1000 times performance improvements in extreme cases) and the most significant cost to fix. This is why we strongly suggest getting a second opinion about data design—whether you've chosen Percona or someone else to address your setup.

Data design is a complex subject that is so closely connected to overall application architecture that the boundaries between the two are not very clear. For this discussion, we'll lump decisions related to what technologies you use—for example where to use MySQL and where to use Hadoop—and other "big picture" questions such as sharding, replication architecture questions, queries and schema as "data design questions."

When designing your schema you need to consider what kind of data you'll be getting, how much data, what kind of distribution it will need, as well as what operations you're going to be doing with it. Simply looking at your application's objects and their relationships, and mapping them to database schema, rarely produces optimal designs for the application. Quite often you need to implement data normalization or denormalization, deploy horizontal or vertical partition or use other techniques. When deciding what combination of indexes is going to work well for your mix of queries, you'll need to consider how to optimize read queries without slowing down writes too much. You also need to decide what it looks like to have "too many" indexes.

Finally, you need to examine your queries and understand in each case when running that particular query is the most efficient way to get the job done. Some optimizations are obvious: using a multi-row insert statement instead of performing 1000 single-row inserts (especially outside of the transaction). Or running one query to fetch all data from the table instead of running a query for each cell. Other decisions are more complicated—you need to understand the MySQL optimizer and execution engine abilities for the version you're using, and which queries it can run efficiently and which it can't.

The biggest mistake we see is just "following your gut," or taking approaches that worked with other databases and transporting them directly to MySQL. Learning how to use the EXPLAIN command in MySQL will help you understand which queries are effective and which aren't.

The EXPLAIN command can be hard to read on its own, especially for complicated queries. MySQL Workbench is a free tool that includes a Visual Explain feature, which helps you to easily understand how a query is executing.

Note. Query performance is highly dependent on data size, data distribution and specific constants that are present in the query. Both the execution plan and the execution time can be completely different. This means that you need to validate performance on a real data set and with a variety of data (not just perform the same action on the same data over and over again).

Many problems result from developers designing and testing applications with very small and artificial data sets, and developing approaches that don't scale at all. If you do not have access to real data (e.g., you've never launched the application), come up with a means for generating artificial data to cover some worse case scenarios.

Query performance does not necessarily change linearly with data size. Depending on your queries and indexing, multiplying the data size by ten might cause query performance to change by 10 or 1000 percent!

This is why it is very important to establish a data retention policy as soon as possible. How long you store the MySQL data is not terribly important: a range from six months to three years is fairly typical. Archiving the data does not mean that you remove the data for good. After setting the data retention policy, we can use the pt- archiver tool (part of the Percona Toolkit™) or some type of custom script to remove the data from the main MySQL server and place it into an "archive" system. You can use another MySQL instance, or even Apache Hadoop, for longer term data storage.

It is also possible to convert data to CSV files and store it on Amazon Web Services® S3 so that the data will be available for on-demand data analysis. You can also use the "archive" systems for data analysis. At the same time, if the archived data is inserted in another MySQL instance, it can be used by applications.

For example, a bank may need to show a statement to a user (a list of all bank operations). Let's say the statement normally goes back only one year. If the user needs to see older statements (beyond one year), the user can select a "view archive" button. The application will then connect to an "archived" MySQL instance.

The bottom line is that a single MySQL server is usually not intended to store terabytes of data: this is the domain of Big Data. In the Big Data world (e.g., Hadoop), data is stored in a cluster and spread across multiple machines. Having a data retention policy can significantly increase application performance, saving time and reducing the cost of maintenance tasks like backup.

If you are using sharding, you might need to address another data-related problem: shard disbalance. Typically this problem originates from poor sharding design choices: sharding on something like the first letter of name (which gives us a far from uniform distribution), or mapping so many users to a single shard that there is not enough capacity to handle the amount of per-user information growth over time.

As with other data design issues, it is good to test things with real data if possible (or as realistically-generated data as you can get).

In any case, it is typically impossible to avoid disbalance completely. While trying to reduce disbalance issues, you need to ensure you also have a solution for shard rebalancing that you can implement when needed.

Resource Saturation

Another common reason for poor performance is a lack of resources. The system has only so much memory, CPU, disk IO and network resources. It can't do more than these constraints allow. By optimizing your software configuration, schema, queries, etc., often the same application actions will require less of these types of resources. So one of the best questions to ask when improving performance is whether you're facing a lack of resources, or if the system isn't optimized enough.

In reality, there is no single answer to this question: often both schema and query size optimization and better hardware resources are possible solutions. You'll need to decide which provides a better return on investment in terms of time and money.

Here are some of the most common resource constraints to consider:

- ***Disk IO****. Many applications require a lot of database disk IO, both in terms of reads and writes. The performance characteristics of available disk IO systems can be huge, ranging from handling some 100 reads/writes for single spinning disks to being able to handle over 100,000 reads/writes for modern PCI Express® solid state storage. If you're still running rotational disks and you become disk bound, those should be the first candidates for replacement.*
- ***Memory****. The main use of the memory for MySQL is as a cache. Cache memory is much faster than even the fastest solid state block device, so if you can get the most actively accessed parts of the database (working set) in memory, it can help performance dramatically.*

- » ***CPU***. *CPU is responsible for the actual code execution. When we refer to CPU performance, we include the performance of the cache and memory subsystems too. MySQL processes single queries on essentially a single core, so you only benefit from the CPU having a large number of cores if there are enough concurrent queries running.*

There are more complicated resource-related issues as well. The network can be the issue, you might have issues with logical constraints on resources coming from virtualization levels, or you might find a solid state storage device gets slower as you fill it up.

No Single Answer

It might sound as if the cause of poor performance and the opportunity for optimization lies only in one of the areas we covered. Typically, the opposite is true. As we look at systems, we often find that there is room for performance improvement in every one of those areas—and those improvements tend to multiply. If we're able to get two times better performance from more hardware resources, two times from better architecture, two times from query optimization and two times from configuration, without getting outrageous gains in any particular area, we've still achieved sixteen times improvement.

Is it MySQL?

IS IT MYSQL?

When users start complaining about application performance, and the application is powered by MySQL, how do you know when it is MySQL and not something else causing the performance degradation? Sometimes the answer is simple: for example, you find the application's queries running for more than 30 seconds in the MySQL's process list. In other cases, it might not be so obvious.

A great way to tackle this problem is to use application-side instrumentation to pinpoint the exact response time for user requests and location of delays. Some applications have their own custom instrumentation, others use tools such as New Relic to get an inside look—both are great options. In many cases, however, there are no such tools available.

The USE Method for MySQL

One fairly simple approach to troubleshooting is called the USE Method (a term coined by Brendan Gregg). USE stands for "Utilization, Saturation and Errors." This method describes how you can look at utilization, saturation and errors in relationship to resources. This process often leads you to where the problems lie.

How and where do you apply this method? Start on the application server and then look at the other servers that are involved in the execution. (This can be difficult for very complicated environments where different requests are hitting different servers simultaneously, with potentially tens of services each.)

A good resource-based monitoring system often helps to spot servers that have issues with utilization, saturation or errors. This will help highlight the cause of the problem you're observing and when resolving issues fixes it.

As you apply this method in relation to MySQL, you can look at standard physical resources such as CPU utilization, length of run queue, disk, network utilization, etc. However, you might also want to look at the MySQL side, as MySQL's logical resource utilization also might cause the problems—even if there are no visible physical resource constraints.

The most common MySQL-related issues are due to locks: table locks or row-level locks where concurrent transactions need to wait for one another without saturating physical resources. If you're using Percona XtraDB Cluster, you also should consider transaction certification delay and flow control.

If you're using Semi-Sync replication with MySQL, waiting for acknowledgment confirmation might be the bottleneck.

In terms of errors, you should always check the MySQL error log. The log only records the most severe issues, however, so it is often helpful to look into the application error log, performance_schema or slow query log (which in Percona Server can be configured to include error codes).

Queries Tell the Story

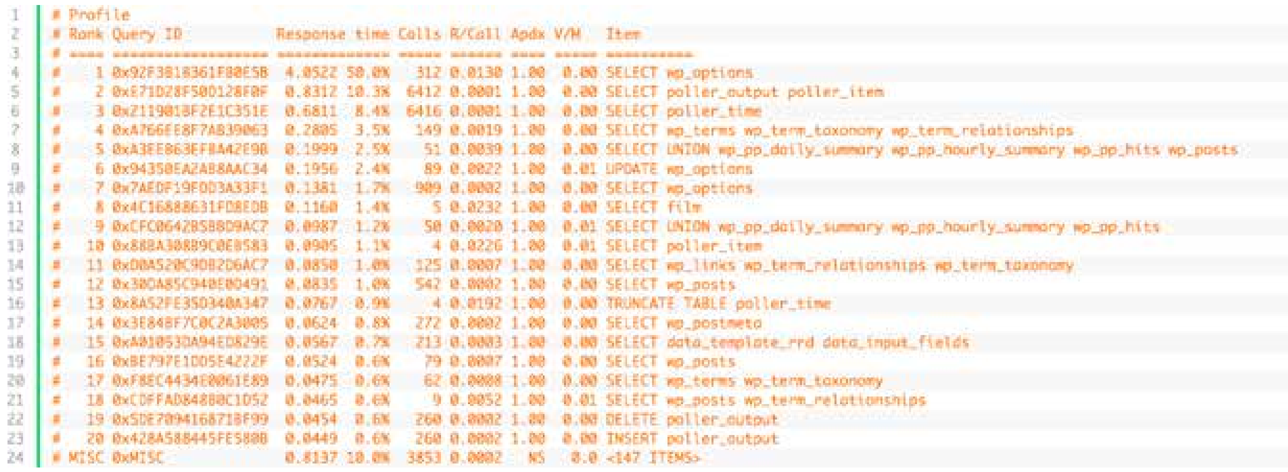
Another way to look at MySQL is through queries and their performance. What is the primary role for MySQL in terms of the application? MySQL should respond to the application queries, give the correct responses and do it fast. With this in mind, analyzing query response times is often a good way to spot problems.

From an application standpoint, query response time consists of three parts:

- ***Running a query by the MySQL server***. *This is typically the main contributor to total response time, and the main thing you should be watching. It is important to understand that it is not the only component of response time, and in some rare cases other contributors might be more important.*
- ***Network contribution***. *Queries must be sent to the server and results sent back via a network. If the server is in a different region and the query is simple, the network may be the biggest contributing factor to the response time. Knowing the network latency and watching network utilization with a fine resolution can often help to highlight this contribution.*
- ***Application processing***. *Sometimes, especially with slow scripting languages, the application's response time might take more time than MySQL's response. This could also be true for the application's MySQL driver. Poorly designed applications request much more data than they need. Application processing is CPU-bound, so you can catch this with CPU usage profiling.*

Let's look at how the MySQL server process queries in more detail. There are a number of tools that allow you to look at the query patterns the server is running: pt-query-digest from Percona Toolkit, VividCortex, MySQL Enterprise Monitor, MONYog®, SolarWinds® Database Performance Analyzer and even roll-your-own scripts using the Performance Schema will highlight this information.

Any of these tools should give you some form of query profile. Below is example from pt-query-digest:



```
# Profile
# Rank Query ID           Response time Calls R/Call Apdx V/M   Item
# ==== ================== ============= ===== ====== ==== ===== ==========
#    1 0x92F3818361F80E5B  4.0522 50.0%   312 0.0130 1.00  0.00 SELECT wp_options
#    2 0xE71D28F50D128F0F  0.8312 10.3%  6412 0.0001 1.00  0.00 SELECT poller_output poller_item
#    3 0x2119018F2E1C351E  0.6811  8.4%  6416 0.0001 1.00  0.00 SELECT poller_time
#    4 0xA766EE8F7AB39063  0.2805  3.5%   149 0.0019 1.00  0.00 SELECT wp_terms wp_term_taxonomy wp_term_relationships
#    5 0xA3EE863EF8A42E98  0.1999  2.5%    51 0.0039 1.00  0.00 SELECT UNION wp_pp_daily_summary wp_pp_hourly_summary wp_pp_hits wp_posts
#    6 0x94350EA2AB8AAC34  0.1956  2.4%    89 0.0022 1.00  0.01 UPDATE wp_options
#    7 0x7AEDF19F0D3A33F1  0.1381  1.7%   909 0.0002 1.00  0.00 SELECT wp_options
#    8 0x4C16888631FD8EDB  0.1160  1.4%     5 0.0232 1.00  0.00 SELECT film
#    9 0xCFC0642B5B8D9AC7  0.0987  1.2%    50 0.0020 1.00  0.01 SELECT UNION wp_pp_daily_summary wp_pp_hourly_summary wp_pp_hits
#   10 0x888A30889C0E8583  0.0905  1.1%     4 0.0226 1.00  0.01 SELECT poller_item
#   11 0xD0A520C9DB2D6AC7  0.0850  1.0%   125 0.0007 1.00  0.00 SELECT wp_links wp_term_relationships wp_term_taxonomy
#   12 0x38DA85C940E00491  0.0835  1.0%   542 0.0002 1.00  0.00 SELECT wp_posts
#   13 0x8A52FE350340A347  0.0767  0.9%     4 0.0192 1.00  0.00 TRUNCATE TABLE poller_time
#   14 0x3E848F7C0C2A3005  0.0624  0.8%   272 0.0002 1.00  0.00 SELECT wp_postmeta
#   15 0xA01053DA94ED829E  0.0567  0.7%   213 0.0003 1.00  0.00 SELECT data_template_rrd data_input_fields
#   16 0xBE797E1DD5E4222F  0.0524  0.6%    79 0.0007 1.00  0.00 SELECT wp_posts
#   17 0xF8EC4434E0061E89  0.0475  0.6%    62 0.0008 1.00  0.00 SELECT wp_terms wp_term_taxonomy
#   18 0xCDFFAD84880C1D52  0.0465  0.6%     9 0.0052 1.00  0.01 SELECT wp_posts wp_term_relationships
#   19 0x5DE7094168718F99  0.0454  0.6%   260 0.0002 1.00  0.00 DELETE poller_output
#   20 0x428A588445FE5808  0.0449  0.6%   260 0.0002 1.00  0.00 INSERT poller_output
# MISC 0xMISC              0.8137 10.0%  3853 0.0002   NS   0.0 <147 ITEMS>
```

Essentially we can see which queries are executed and their typical response times.

As you capture this information on regular intervals for application-side performance problems, there will be a few possibilities:

- » ***Profile looks the same***. *If the MySQL server is getting the same query mix, and the response times are the same, it is unlikely that it is a MySQL server problem.*
- » ***Some queries are much more frequent***. *If some queries are much more frequent than usual, it might be a problem on the application side. For example, a malfunctioning cache often has a major inflow of specific queries. An increase in traffic often causes a proportional increase in queries.*
- » ***There is less inflow of some (or all) queries***. *This might indicate a problem on the application side (or some other tier). For example, you might find a caching layer is overloaded, reducing the inflow of queries to MySQL. It could also be a MySQL issue. Take a look at the Processlist in this case. Severe bottlenecks, such as a locked table that is important to the process, might block a lot of queries and Processlist will show few queries passing through.*

» ***Response times for some (or all) queries are much higher**. If you see a similar number or a smaller number of queries processed in the same period of time, but with higher response times, chances are something is happening on the MySQL side and it needs attention.*

In addition to examining the "query fingerprint," it is often a good idea to look at the "concurrency" reported by pt-query-digest, or the number of threads_running as plotted by our monitoring system. If there are regularly more threads running than CPU cores available, chances are you're dealing with a bottleneck that can be related to CPU saturation, disk IO or table/row-level lock contention. In this case MySQL should be the suspect, and needs to be checked out.

Types of Problems

TYPES OF PROBLEMS

We started with the assumption that application performance is poor and needs troubleshooting. But how did we get there? Knowing how the performance issue was detected can be very helpful in isolating what is causing the issue—and result in faster and better fixes.

Quality Assurance and Catching Performance Issues

First, let's discuss the role of quality assurance (QA), benchmarking and staging environments in ensuring application performance. It is important to have performance-focused QA processes in place, along with functionality-based QA. It is important to do performance QA "correctly": in other words, using realistic data and workloads. However, you can't expect to prevent all performance problems through QA, because the real world is a much more complicated environment than the lab.

Accepting the fact that issues will crop up in your production environment and be ready to use the proper tools to troubleshoot that system is a better and more prudent expectation. Hoping your fantastic QA team ensured that nothing bad will go into production, at least not in your code, is probably less than realistic!

Slow Performance Regression Until It Hurts

One of the most common paths applications can take is that performance degrades gradually— nothing really breaks, but more and more users get annoyed until the problem becomes unacceptable and a major priority. In many cases, some business-affecting event will suddenly make performance a priority: a large

customer gets fed up and leaves, someone files a complaint based on application performance, or performance derails the boss during a demonstration for the board of directors.

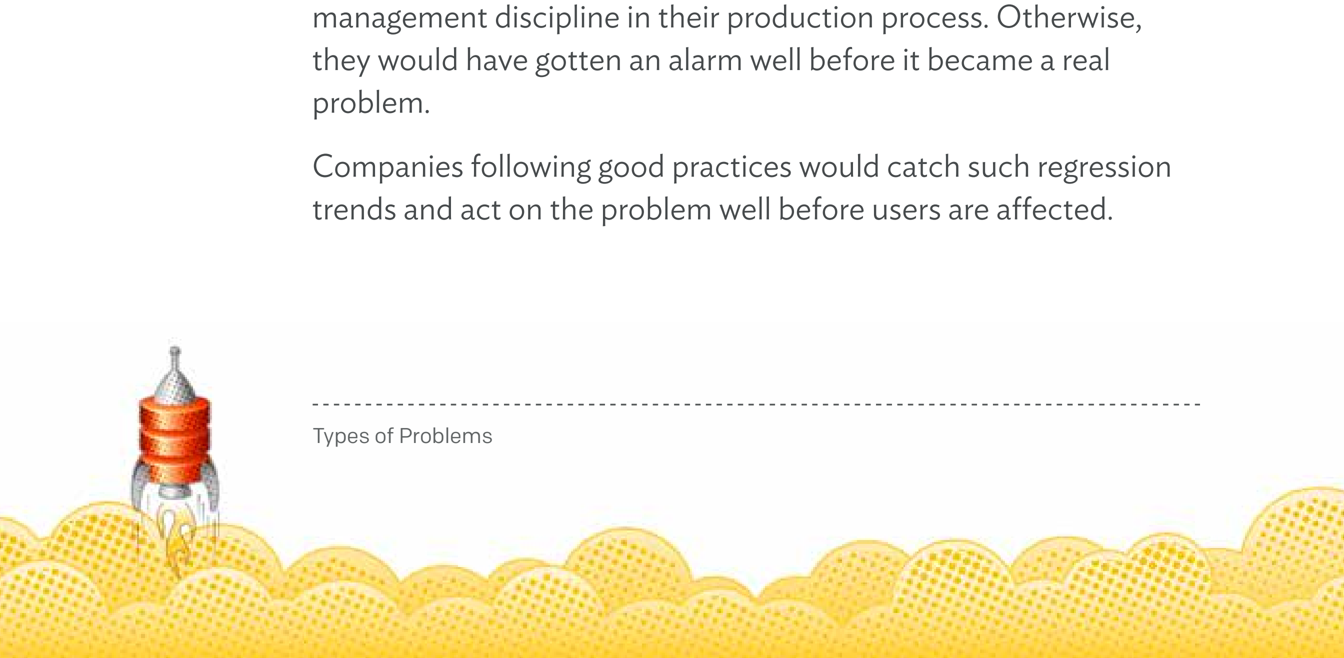
Slow regression issues have good points and bad points. The good thing is you don't often need drastic optimization to get the system back to acceptable performance levels. Often when looking at such systems, we're able to change a few configuration variables, add some indexes and solve the immediate problem.

Chances are, however, that such minor fixes won't provide a long enough runway for rapidly growing applications—major architecture changes might be needed to build long-term solutions. Way too often we see companies that don't really follow through with this second part of the equation, and we find them in much worse shape after a few months.

This is analogous to an ER patient that gets a quick fix for an immediate symptom caused by a long term condition, but afterward won't follow through on advice for lifestyle adjustments. The excuses are often the same: time, money, motivation and the hope that the quick fix will be enough to permanently address the situation.

If a company allows their application performance to slowly degrade, it also often means they lack the same performance management discipline in their production process. Otherwise, they would have gotten an alarm well before it became a real problem.

Companies following good practices would catch such regression trends and act on the problem well before users are affected.



APDEX is an excellent standard you should consider exploring.

APDEX essentially defines what response time users consider great, what is acceptable and provides a computing score that measures the performance of any given application:

```
Apdex = (Satisfactory samples + 0.5xTolerating
samples + 0xFrustrated samples)/Total samples
```

As an example, you might choose one second as a satisfactory response time for a given type of user interaction, and three seconds as a tolerable response time. If out of 100 samples you have 95 responses below one second and four responses below three seconds, your APDEX score will be .97.

When using the APDEX scoring system (or other simple percentile methods) the system should respond within three seconds 99.9 percent of the time. Setting the goals higher than users' expectations will allow you to spot performance problems and resolve them before they affect the user's experience.

APDEX works best when looking at actual user interactions. It is often handled on the Web layer, especially if you're timing performance using the browser rendering (or mobile app instrumentation). This accounts for the response time tied to JavaScript execution, network design, and page rendering. Tools like New Relic have good support for these types of technologies.

If you're responsible for database performance, you can apply many of these same practices. It doesn't take much to figure out that for the application to respond within "N" milliseconds, we need a given query to respond within "M" milliseconds. After doing the math, you can compare the computed response time to the expected 95 percent response time as computed by pt-query-digest (or other tools), and set up a warning if the response time constantly exceeds the expected threshold.

Dramatic Change in a Short Time

A second common issue is abrupt and severe performance problems. We often hear something like this: "It worked fine yesterday! Today it just doesn't work! We didn't change a thing!"

Computers are state machines: if there is no change at all they will work exactly the same way. Drastic performance regression means somewhere a drastic change happened—you just need to find it.

Finding what changed can be very easy or very hard. To this point, in many cases instead of looking for what could have changed and why the system slowed down, we prefer to look at it as an opportunity to optimize the system in general. This is especially true when we don't have a good history regarding the system operation.

This is a key reason to use a very good monitoring system, in conjunction with tight change controls. With those in place, you can make sure that you record any code changes, configuration changes or schema changes, and it is possible to review these changes—as opposed to starting logs only when a regression happened.

Let's take a look at some of the situations that can cause unfortunate system changes, and how to deal with them.

New Application Code

Installing new application code, whether to implement new features or to make changes to existing software, is perhaps the number one cause of performance problems.

We always strongly recommend that you establish a process where you review the code, database schema changes and queries for their performance and scalability before you deploy them

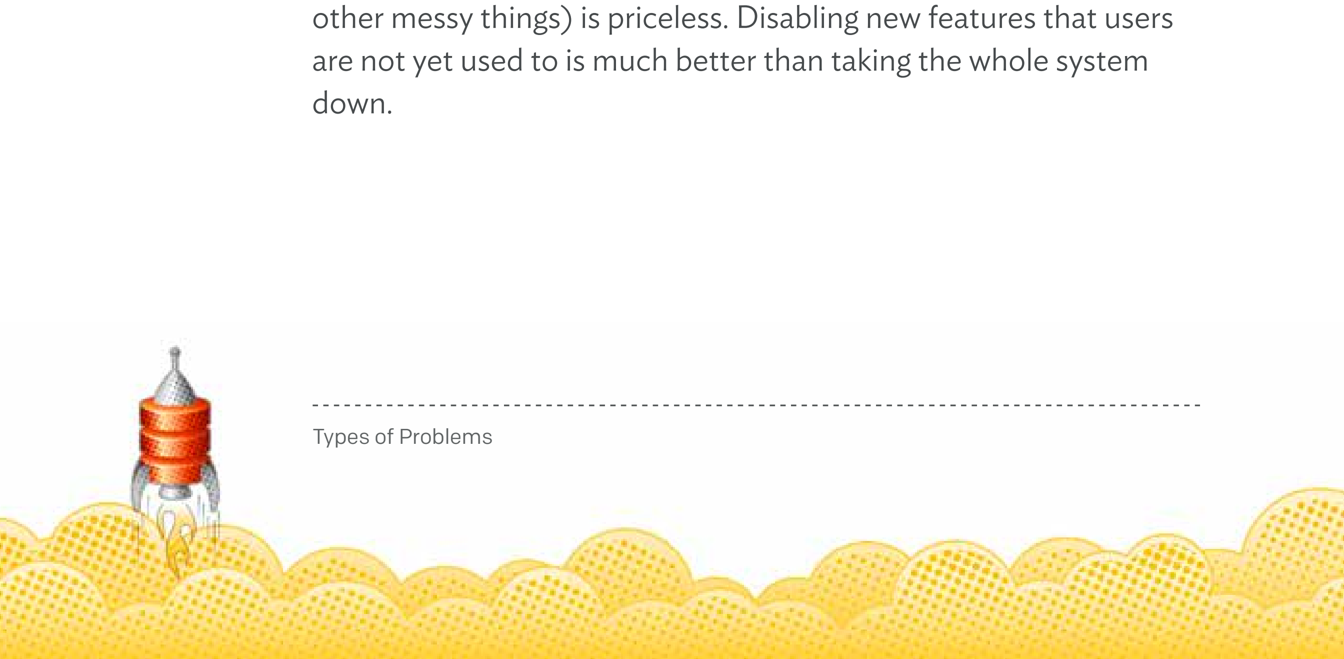
in production. Even better, perform some capacity planning to validate how the code operates under full production scale, as well as when you push it to the limit. At what scale do things start to break? The more important it is that an application stays up and performs well, the more you should invest in pre-production validation activities.

A specific problem that can occur from installing new features in large-scale systems is how quickly the new software will grow the database size. If indexing and query design is sub-par, problems can appear much faster for a new feature set than for mature ones.

Let's imagine we have a publishing site, and we decided to enable comments. When we launch the system the "comments" table is empty. Comment sections being what they are, it might quickly grow to millions of comments in just a couple of days—dramatically changing the system profile.

Besides thoroughly testing features in QA, it's often a good idea to have a switch that will easily enable/disable a feature in production. You can then move the code to production with the feature off, enable it for a period of time (or a sampling of users if feasible) and then review the statistics on how it worked live.

The ability to disable features in production without causing emergency code rollback (which can require database changes and other messy things) is priceless. Disabling new features that users are not yet used to is much better than taking the whole system down.



Database Schema Changes

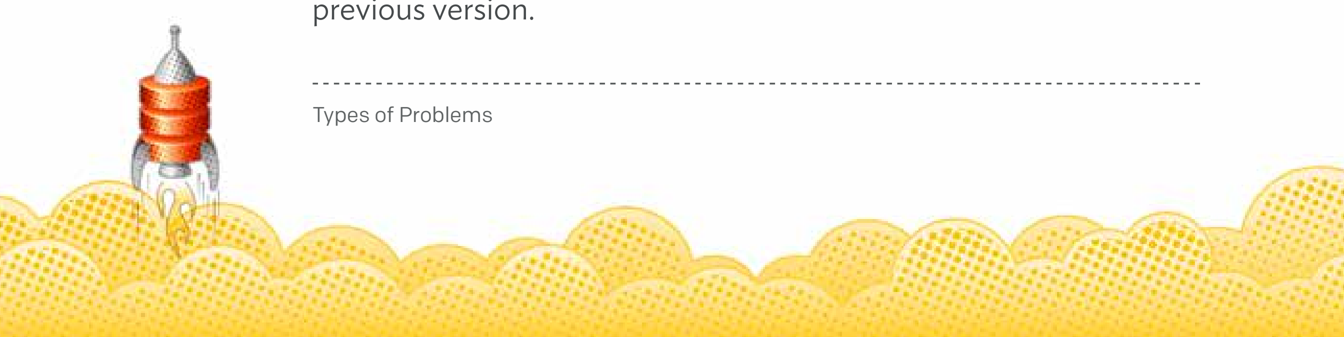
Database schema changes are often the culprits responsible for performance regressions. This shows up many different ways: dropping an "unneeded" index (causing queries to run much slower), adding indexes, adding foreign key constraints or changing column types. All of these schema modifications have been known to cause performance issues.

The solution for schema changes is the same as application code changes: test changes in QA and validate their actual impact in production by looking at the query execution times. Have the queries started running slower, or have outliers appeared that weren't there before?

Changes such as adding an index or dropping an index are easy to understand and address. The optimizer gets more or fewer choices on how to run the query, and might or might not put them to good use—or come up with an entirely different plan. Other issues, however, are less obvious.

First, there is the physical layout. When a table (or index) is rebuilt, the physical layout may change. This can give it a different performance profile, causing some access operations to go faster or slower. For example, rebuilding an index might put it in its most optimal form for fast reads, but index writes could initiate an increased number of page splits—causing a slowdown in write operations.

Second, there is the memory content. Building indexes or rebuilding a table often washes out a lot of valuable data from the database cache. This can cause queries to run slower directly after completing the change. Give the system a chance to "warm up" after changes to tables before panicking and rolling back to the previous version.



Finally, there is luck. Yes you read that correctly! There is some luck involved in the MySQL optimizer plan selection. Some of the statistical estimates MySQL depends on can be affected by both the physical layout (which can change) and rolls of the dice (for estimation purposes). The results of such dice rolls can be stored in persistent tables, permanently changing the optimizer plans. If a query is running slower, check out the EXPLAIN plan to see if it has changed substantially.

Changes to System Software or Hardware

Changes to hardware, the operating system version or its configuration, or MySQL's version or configuration can all cause performance changes too. Again, you need to make sure you are cognizant of such changes and have either validated their impact in advance—or at least assessed performance logs after rolling out the change.

Much of these are obvious, common-sense ideas for regular networking environments. It is a completely different story when things get more complicated—such as cloud environments, virtualized environments or an enterprise structured with substantially siloed teams.

In cloud deployments, many of the changes to the infrastructure, virtualization software, physical host, network topology, etc., are completely outside of your control. You won't often have insight into the how, where and why of these changes. Besides software version or configuration changes, you might be looking at changes in the environment: your neighbors might get less noisy, which can affect the CPU, IO or network resources available to you. Cloud isolation has been steadily improving over the years, but it is still not complete at this point.

Virtualization is very similar to the cloud, with the difference that it's sometimes possible to get details on the version or configuration changes. This depends heavily on the relationships you maintain between teams in your organization: I've seen cases where the team responsible for VMware provisioning was completely isolated from the DBAs.

In addition to server virtualization, storage might also have transparency issues. If your database is using network-attached storage (NAS) or storage area network (SAN) solution, it might be a black box whose performance profile could be altered by software and configuration changes. Performance can be affected by the storage system load, as well as other applications running on top of the same SAN/NAS device.

Another obvious hardware issue that negatively affects performance is hardware failure. Hardware goes down all the time, due to mechanical, electrical, or simple power supply issues. Some common examples are RAID failures, RAID battery charge drained or SSD garbage collecting activities. Sometimes somebody just accidentally unplugs a device! However it occurs, hardware can stop working and disrupt performance, or contribute to performance degradation when, by coming back online, it adds a spike to the workload. The solution is to ensure your critical systems are redundant and have backup power.

If you can't verify all the possible changes that could potentially affect your application performance, at the very least ensure that you're capturing information about the raw performance of the "physical" resources such as the disk, CPU or the network.

Software outside of MySQL can also "misbehave" as well. Software can disrupt performance in large or small ways. One example is Linux deciding to swap process memory from RAM to a file on disk. The Linux kernel is a complicated beast, using algorithms to determine when to perform a swap. The community has been very vocal about how this is disruptive to processes, and the case of database performance is no exception.

One common issue we see with software—from MySQL, Linux or anything else—is that people assume the latest version will make everything better. This is not always true. In fact, major software changes typically come with some regressions—whether you get affected or not. It could be a direct feature or performance regression or an interoperability issue. For example, we commonly seen MySQL performance actually go down when the CPUs are upgraded to a newer version, as a larger amount of cores can cause increased contention and a subsequent drop in performance.

The solution is obvious: make sure to validate your upgrade performance expectations and be ready to deal with the problems if they appear in production. This includes a plan to rollback to an older version.

Workload Changes

A system's performance profile can change due to workload changes. In the age of the Internet, the workload of connected applications is always changing. Every day is different from the last day, and won't be like the next day. Most of these workload changes are fairly insignificant, though—with some noted exceptions.

One example of suddenly increased traffic is if the website or application suddenly gets a major promotional bump—placement on a TV show, celebrity endorsement or review on a popular blog

or some heavily-trafficked media site. It can happen without warning and can magically and exponentially increase the workload.

The best defense in these cases is a good offense: have a plan for what to do during a huge traffic increase. Smaller sites typically need to plan better for "spike readiness" than larger ones. It's easy to legitimately and unpredictably increase traffic on a small site by ten or even 100 times for a short span. It is much less likely to happen on a site like Facebook.

If your system exposes an API for users, you have an additional risk of heavy API use suddenly appearing out of nowhere. Make sure you have good statistics on API use, and the ability to throttle or disable abusers.

Another source of traffic is bots. Some bots are good, such as a GoogleBot indexing your website to bring more visitors. Others, however, might be rogue bots scanning for email addresses to spam (or other nefarious purposes). The problem with bots, and rogue bots especially, is that they often don't respect robots.txt (a file that contains rules bots should follow). Rogue bots often mask themselves as a known bot or a conventional browser.

Another scenario that will increase your workload is becoming the target of a DDoS attack. This will flood your system with requests in order to take it down. A DDoS attack hits pages on your website that are expensive to generate, or generates an insane amount of traffic to saturate firewalls and Internet channels.

At this time, the largest reported DDoS attack generated 602 Gbps of traffic—something most network providers couldn't handle.

Whatever the reason, the chances are that a workload change at the application level will also cause a workload change on the database side.

Background Jobs/Maintenance

But what if we can see that the application level workload is basically the same: the same kinds of requests are coming in at the same levels? There are other reasons why the database workload can change, namely background jobs and maintenance processes. These exist on many systems, and often are added by developers or OPSE without the proper procedures or controls. These processes can cause the database to overload. In some applications, the background data processing might be the majority of the workload, and in these cases it is important not to discount them as something that affects performance. In looking at the background activity, observe not just what is happening on the MySQL side, but what happening to the resources that MySQL is using.

For example, a batch job written in Python™ and running on the database node might initiate swapping by allocating a large amount of memory. This could cause severe database performance problems.

Another common issue is poor service configuration. If the backup service is taking up too much memory, for example, it could be competing for disk IO or network resources.

It’s possible that a malfunctioning system component could generate additional traffic. If there are issues with the caching layer, you might end up having more database hits than you normally would for the same application traffic. The same thing can happen if something is wrong with your queuing system, which should be designed to queue work, aggregate it and spread it out over time.

These type of malfunctions often disproportionately increase certain types of queries.

There are many reasons why the workload can change, and the key is to make sure you can detect these changes in both the application request rate and the database query rate. Monitoring this activity often helps find the reason for the workload increase, and helps you take the appropriate action. It's necessary to look at the full system activities picture to understand your true workload!

Intermittent Problems

Perhaps hardest problems to diagnose are intermittent problems. If your performance is slow all the time, or if it happens at specific times (at peak hours, for example), it is relatively easy to take a look at the system and find what is going on. However, what if the problem happens randomly and performance is bad only for a couple of minutes (or seconds), or if only a slice of your users is affected?

Intermittent problems are difficult, but quite typical. In fact, a lot of problems start as intermittent problems, gradually increase in magnitude, and—if undetected and unfixed—become persistent problems.

It is important to have a good monitoring process, with high resolution, in place to deal with intermittent problems. If you're only capturing data every minute, you might miss the majority of intermittent problems—at least at their start, when they can last for only a couple of seconds. Good monitoring systems capture at least some data at intervals of one second. One of the systems designed with high resolution in mind is VividCortex. VividCortex can be very helpful in diagnosing intermittent problems.

Another tool that is very helpful, even if you're running a high-resolution data capture monitoring system, is the pt-stalk tool from Percona Toolkit. You can configure it to trigger at the beginning of

a suspected incident. It will capture and gather detailed diagnostic information such as disk IO, InnoDB engine status, processlist and many others.

Before discussing intermittent problems further, it makes sense to examine how computer systems (including web and database servers) process requests. Requests come in from users based on something called "random arrivals." Real system don't get a static average request rate—it changes dramatically between peak and low traffic hours. If we look at some specific time, however, we can figure out an average rate of requests (let's say 100 requests/ sec. for this example).

It's important to understand that this is an average number—it does not mean that every second exactly 100 requests come in. For some seconds it might be 80 requests, others 120, and rarely will there be 200 requests per second. With more granularity, more variance will be observed. For example, if we zoom down to milliseconds with only 100 requests/sec, most milliseconds won't have any requests. Other milliseconds, however, will have one, two or even five requests.

When requests come, they must be processed and responses sent back. The distribution of responses does not always match the distribution of arrivals. So using the case above, we might need 5ms of CPU time to serve the request (and nothing else). If we have only one CPU, for five requests that came in at the same 1ms slot, we might queue them and process them one after another: responding at 5, 10, 20, 25 and 30ms.

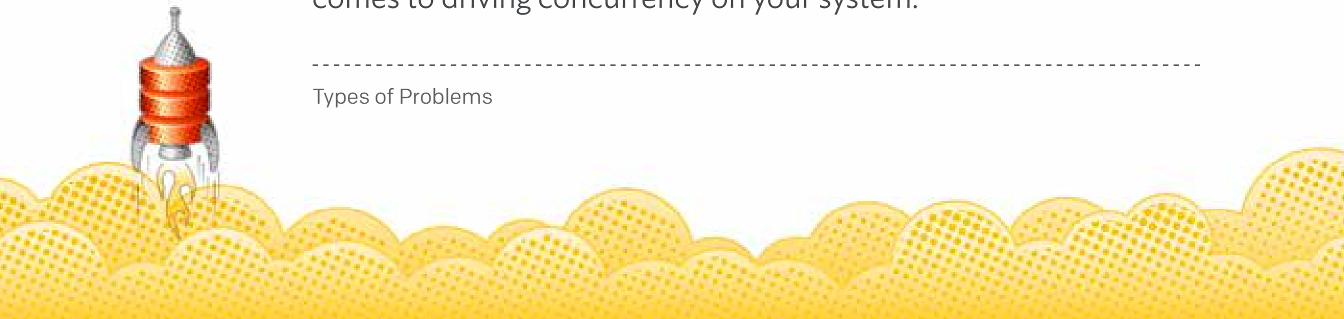
This example shows us an interesting request-handling property: requests are queued and serviced at different times. This queuing and servicing happens all the time, and frankly it's not possible to observe it at all levels. For example, if we go very deep into the CPU core the executing thread might stall on the CPU cache while

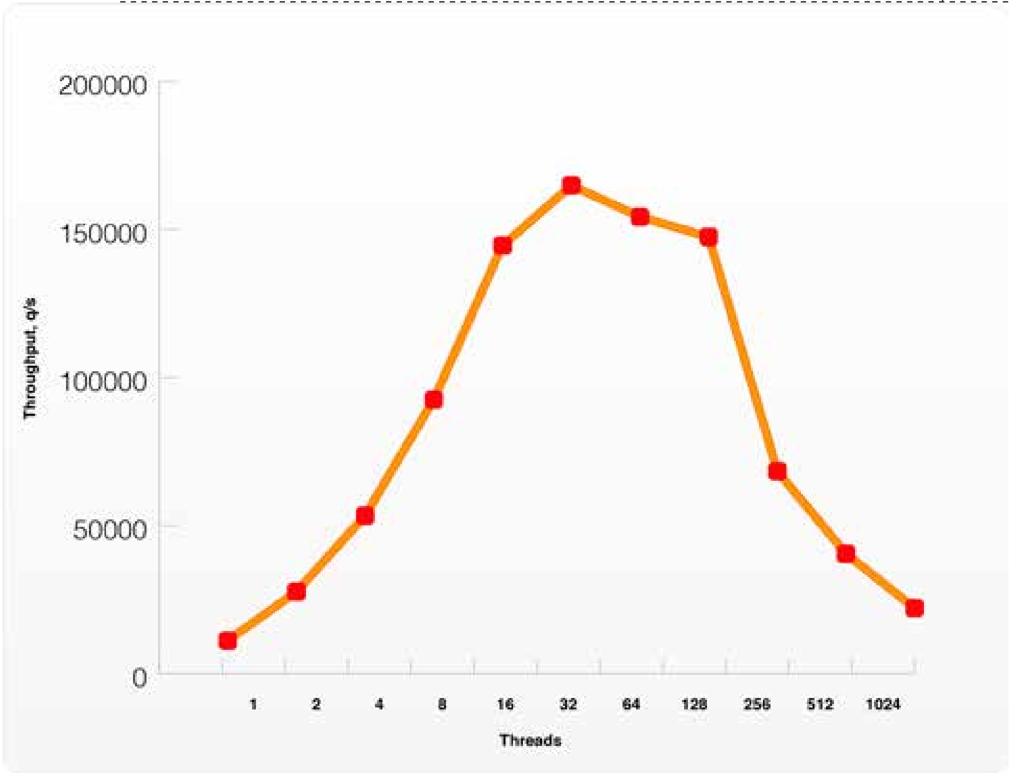
waiting for data to come from memory. A multi-threaded CPU can only switch to executing another thread when putting this thread back in the queue. This extremely low-level switching between requests waiting in the queue and those actually being serviced is for the most part invisible to the application.

Requests can also be processed concurrently rather than sequentially. In the example above, what would likely happen is that five threads could process the five requests concurrently (assuming there are enough CPU cores available), and in approximately 5ms.

Typically there are many concurrency controls in a system that define how requests are executed. For example you might be running a web server such as Nginx®, and you can configure how many requests to process at the same time. You could also configure when a request is sent to a different network process, such as a Java application server (which also has its own set of controls determining how many requests to truly run in parallel). The size of the MySQL connection pool will also control how many concurrent connections (and as such, queries) are sent to MySQL. MySQL has explicit configuration settings for both the thread pool (innodb_thread_pool) and implicit limiting factors such as row level locks that define how it handles concurrent requests.

It is worth mentioning that every system has a certain level of concurrency, at which it achieves peak throughput. The Universal Scalability Law determines peak throughput. Pushing systems well beyond their optimal concurrency throughput can cause them to perform extremely poorly. For example, certain MySQL versions running 1000 queries from 1000 connections at the same time can lower throughput exponentially, compared to running just 32 queries at the same time. So more is not always better when it comes to driving concurrency on your system.





Threads	Throughput (qps)
1	11178.34
2	27741.06
4	53364.52
8	92546.73
16	144619.58
32	164884.03
64	154235.73
128	147456.33
256	68369.02
512	40509.67
1024	22166.94

As such, configuring all relevant components to their optimal concurrency and relying on queuing to service more concurrent requests is one of the most common performance optimization strategies.

So let's put all of this together. The system will generally have a variable inflow of requests every second, where the number of requests will be in flights at the different execution stages. Some are processed by the application server, others by the database server CPU, others processed by the storage system, etc.

Now that we've reviewed how a system processes requests, let's return to look at the details of different types of intermittent problems.

The first type of intermittent problem is caused by a load spike hitting the system. If your system typically gets and services 100 requests/sec, and 1000 requests come in, the system will have to either handle more requests concurrently (which as we discussed can mean worse throughput) or queue them up and gradually work through the queue. In either case, we're going to get a few seconds of poor performance while the spike is worked off.

There are many reasons for such spikes that are completely external, ranging from some aggressive API user to some Internet problem being resolved (causing a lot of queued connections to hit at the same time).

If we look at MySQL, many application-level issues might cause such an inflow of requests. For example, some "lock" on the cache level might get thousands of requests queued up, and releasing them at the same time could cause an avalanche of MySQL requests. Problems of this nature are often described as a "stampede."

In this class of intermittent problems, we have a high spike of system requests—well beyond the system's ability. The solution is typically to figure out the cause and see how to prevent it.

In another class of intermittent problems you won't see the request inflow spike, but you will see a slowdown in the system's attempts at processing them. As request processing slows down, we often get increased concurrency that exceeds the optimal system level, compounding the original problem. For this issue, you need to find the root cause of the slowdown.

In many cases, this slowdown could be related to database server locks (row level locks or table level locks), a very bad plan for one of the queries, a query taking up too many system resources, errors in the storage system or the network, or some internal MySQL or OS (i.e., filesystem) issues.

It's wrong to assume that any errors from the network or storage will be propagated to the application. If a frame is lost or corrupted at the Ethernet level, it will be retransmitted through TCP/IP error correction—but only after a significant timeout. If the disk has a hard time reading the data, often there will be a number of retries and other recovery actions done. This can translate to longer response times rather than an error.

For these types of intermittent problems, you need to focus on finding what caused the processing slowdown and resolve it.

It's not easy to differentiate between problems from requests started and requests completed. One way to tell is that if there are a lot of "starts" in the given second, it is likely an issue with the level up. If, with the same number of starts, we don't see a lot of resolutions (i.e., row level lock stalls), then it is an internal system issue.

Examining system concurrency details (threads_running in MySQL) might be very helpful in this case: did it jump from a normal value of 20 to 1000 threads in one second, or did it take several seconds or minutes for it to build up? The problem, of course, is that for high-volume systems even one-second resolution is not enough to see what is really happening.

Stalls at the MySQL or OS level are a very common cause of these types of problems. A "stall" is defined as normal database operations blocked for all users (or a large portion of users) for a period of time. This is different than the other examples above (like disk IO requested by the query) because such operations only impact one client. Facebook's team (and Mark Callaghan specifically) did a lot of work cleaning up stalls that happen in MySQL code under various circumstances, and MySQL has greatly improved in this area over last several years.

The stalls referred to above are not long stalls that last for seconds—typically these are micro stalls that only take a few milliseconds—but they can cause very serious issues for database performance.

Let's look at a realistic case: the database handles 50,000 queries a second from many applications, and connections are created and destroyed as needed with no additional concurrency controls. Typically the system works in the optimal range of 20 to 40 queries running (threads_running), close to the number available and allowing for some network and disk IO. 99.9 percent of queries are handled in less than 1ms.

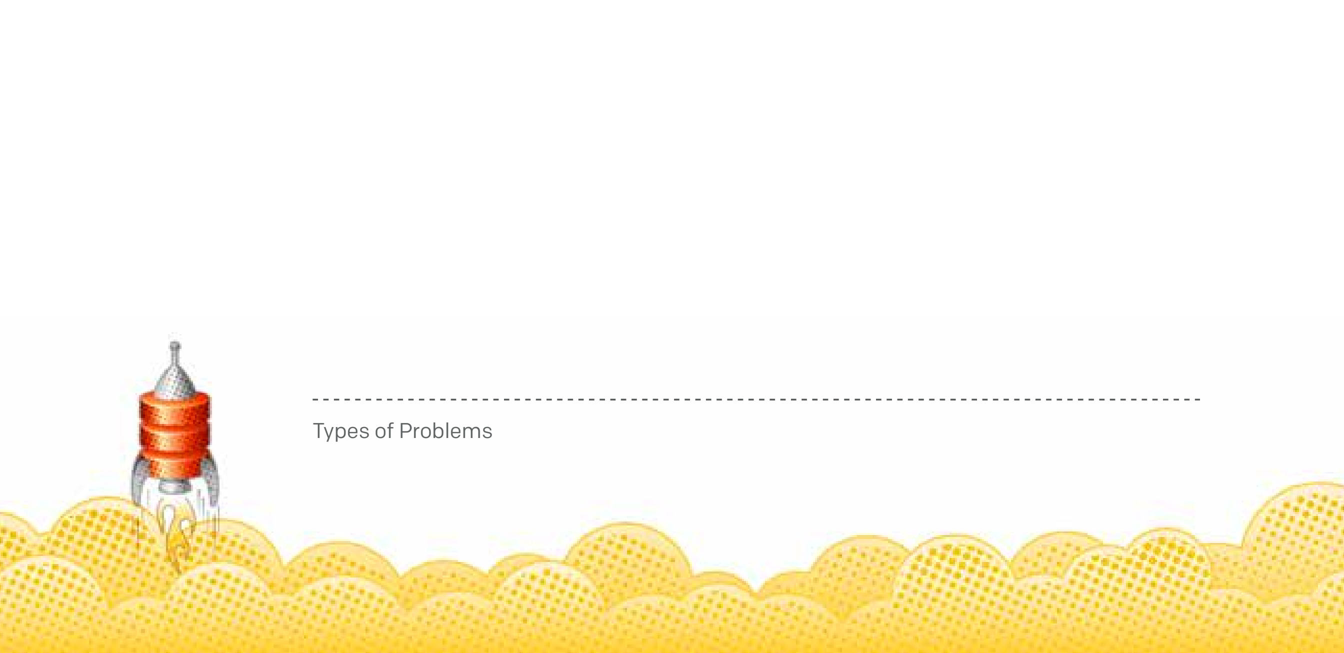
Now imagine a complete stall happens in this system for 100ms: during that 100ms, five thousand new queries come in, and as no queries are completed, five thousand more threads are created (and queries from those threads start executing) until blocked by the lock or other cause of the stall. When the stall stops, we have more than 5000 queries that start running at the same time (only the number of queries matching the number of CPU threads will start running at once, but it does not matter here). This will quite likely push the system into a concurrency mode that is less than optimal, and as such become unable to handle the normal 50000 queries/sec while matching the inflow rate.

Would the system become more and more overloaded and spiral down? Possibly. But what usually happens in systems involving humans is that as the system becomes slower, users send in fewer requests. As such, the request inflow gradually reduces to the point where the system can handle all requests at a faster rate than they arrive and the system gradually recovers its optimal state. It then starts to serve users faster, so they start sending a normal rate of requests again. It also "helps" that the number of MySQL connections is limited, so after a certain period of time the application will start getting errors instead of pushing a bigger load onto MySQL. This assists recovery (at the cost of a partial downtime).

Analyzing such short terms effects is very tricky, especially as very high-resolution data captured in production (think millisecond level) is often hard to accomplish without the overhead cost getting too high. MySQL's performance schema and the Linux kernel instrumentation frameworks (Perf Events, DTrace, etc.) can be quite useful in these situations.

When diagnosing stalls, one of the most difficult things to see is cause and effect. Often by the time data capture tools like pt-stalk trigger, you are already looking at the effects and not the cause, because the cause has already occurred. Imagine, for example, some process locks a table for 100ms (causing the stall described above). If you start your examination just one second into the incident, you will see thousands of queries running without any idea why so many are running in the first place.

Diagnosing intermittent problems is definitely an art. But I still believe that understanding why they happen, and what exactly is going on inside MySQL when they happen, can help you address them much more efficiently.



How Percona Can Help

How Percona Can Help

Potential performance killers are easy to miss when you are busy with daily database administration. Once these problems are discovered and corrected, you will see noticeable improvements in data performance and resilience.

Percona Consulting can help you maximize the performance of your database deployment with our MySQL performance audit, tuning and optimization services. The first step is usually a Performance Audit. As part of a Performance Audit, we will methodically and analytically review your servers and provide a detailed report of their current health, as well as detail potential areas for improvement. Our analysis encompasses the full stack and provides you with detailed metrics and recommendations that go beyond the performance of your software to enable true performance optimization. Most audits lead to substantial performance gains.

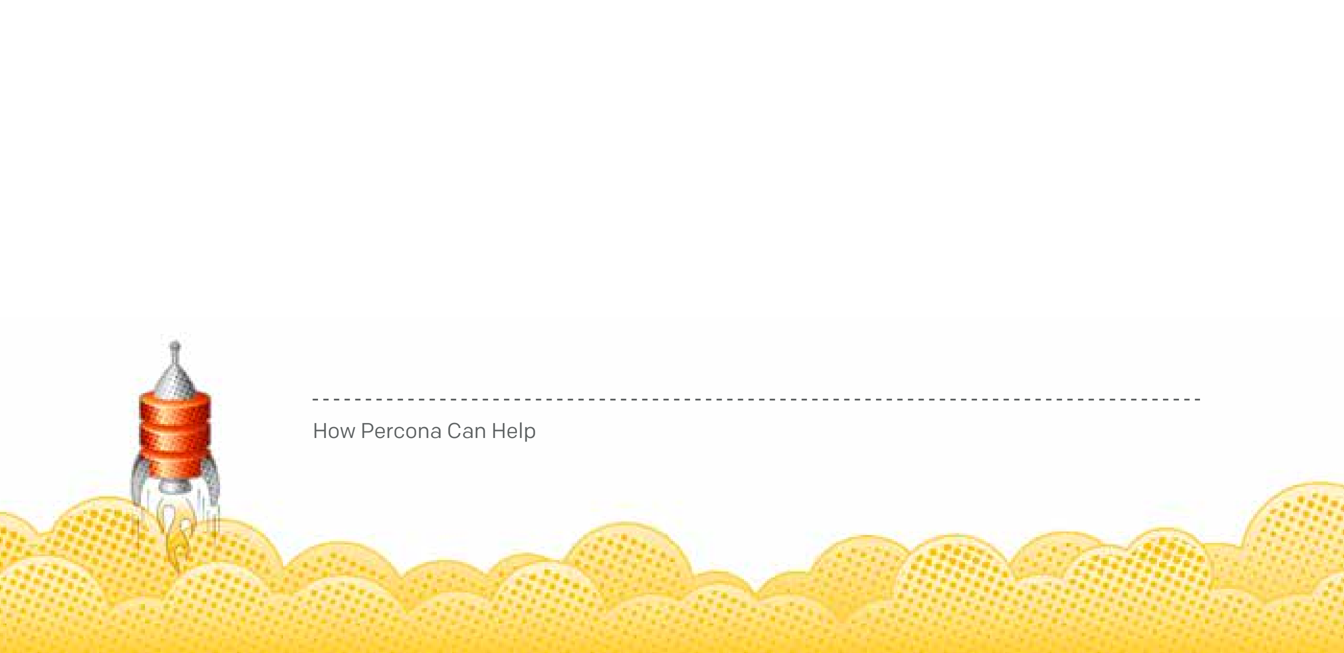
If you are ready to proactively improve the performance of your system, we can help with approaches such as offloading workload-intensive operations to Memcached. If your user base is growing rapidly and you need optimal performance on a large scale, we can help you evaluate solutions. If performance problems lie outside of MySQL or NoSQL, such as in your web server, we can usually diagnose and report on that as well.

Percona Support can provide developers and members of your operation team the 24x7x365 resources they need to both build high-performance applications and fix potential performance issues. Percona Support is a highly responsive, effective, affordable option to ensure the continuous performance of your deployment.

Our user-friendly Support team is accessible 24x7 online or by phone to ensure that your databases are running optimally. We can help you increase your uptime, be more productive, reduce your support budget, and implement fixes for performance issues faster.

If you want to put your time and focus on your business, but still have peace of mind knowing that your data and database will be fast, available, resilient and secure, Percona Database Managed Services may be ideal for you. With Percona Managed Services, your application performance can be proactively managed by our team of experts so that problems are identified and resolved before they impact your business. When you work with Percona's Managed Service team you can leverage our deep operational knowledge of Percona Server, MySQL, MongoDB, MariaDB®, OpenStack Trove and Amazon® RDS to ensure your data (and your database) is performing at the highest levels.

Our experts are available to help, if you need additional manpower or expertise to improve and ensure the performance of your system. To discuss your performance optimization needs, please call us at +1-888-316-9775 (USA), +44 (203) 6086727 (Europe), visit http://learn.percona.com/contact-me or have us contact you.



ABOUT PERCONA

Percona is the only company that delivers enterprise-class software, support, consulting, and managed services solutions for both MySQL and MongoDB across traditional and cloud-based platforms that maximize application performance while streamlining database efficiencies. Our global 24x7x365 consulting team has worked with over 3,000 clients worldwide, including the largest companies on the Internet, who use MySQL, Percona Server, Amazon RDS for MySQL, MariaDB and MongoDB.

Percona consultants have decades of experience solving complex database and data performance issues and design challenges. We consult on the full LAMP stack, from hardware to operating systems and right up through the database and web tiers. Because we are both broadly and deeply experienced, we can help build complete solutions. Our consultants work both remotely and on site. We can also provide full-time or part-time interim staff to cover employee absences or provide extra help on big projects.

Percona was founded in August 2006 by Peter Zaitsev and Vadim Tkachenko and now employs a global network of experts with a staff of over 125 people. Our customer list is large and diverse and we have one of the highest renewal rates in the business. Our expertise is visible in our widely read Percona Data Performance blog and our book High Performance MySQL.

Visit Percona at www.percona.com

Percona Corporate Headquarters

8081 Arco Corporate Drive, Suite 330
Raleigh, NC 27617, USA

usa +1 (888) 401-3401
eur +44 (203) 6086727

www.percona.com
info@percona.com