# Yh-Mysqld-exporter

192.168.0.179:9104/metrics http://192.168.0.179:9104/metrics

docker run -d -p 9104:9104 --restart=always --name=me \

-e DATA\_SOURCE\_NAME="root:Admin@123@(192.168.0.173:3306)/paasos" \

docker.dev.yihecloud.com/base/mysqld-exporter

#查询MySQL每小时接受到的字节数

increase(mysql\_global\_status\_bytes\_received[1h])

#查询吞吐量

increase(mysql\_global\_status\_questions[1h])

SHOW GLOBAL STATUS LIKE "Questions";

increase(mysql\_global\_status\_commands\_total{command="select"}[1h])

#慢查询

mysql\_global\_variables\_long\_query\_time

#获得行的锁定次数

SHOW GLOBAL STATUS

increase(mysql\_global\_status\_innodb\_row\_lock\_waits[1h])

#连接数

mysql\_global\_status\_threads\_connected

#最大连接数

mysql\_global\_variables\_max\_connections

#空闲连接数

mysql\_global\_variables\_max\_connections-mysql\_global\_status\_threads\_connected

#连接失败用户数

mysql\_global\_status\_aborted\_clients

#缓冲池大小

mysql\_global\_variables\_innodb\_buffer\_pool\_size

#空闲

mysql\_global\_status\_buffer\_pool\_pages{state="free"}

## 查询语言

#查询MySQL每小时接受到的字节数

increase(mysql\_global\_status\_bytes\_received[1h])

#查询吞吐量

increase(mysql\_global\_status\_questions[1h])

increase(mysql\_global\_status\_commands\_total{command="select"}[1h])

#慢查询

mysql\_global\_status\_slow\_queries

mysql\_global\_variables\_long\_query\_time

#获得行的锁定次数

increase(mysql\_global\_status\_innodb\_row\_lock\_waits[1h])

#连接数

mysql\_global\_status\_threads\_connected

#最大连接数

mysql\_global\_variables\_max\_connections

#空闲连接数

mysql\_global\_variables\_max\_connections-mysql\_global\_status\_threads\_connected

#连接失败用户数

mysql\_global\_status\_aborted\_clients

#缓冲池大小

mysql\_global\_variables\_innodb\_buffer\_pool\_size

#空闲

mysql\_global\_status\_buffer\_pool\_pages{state="free"}

# prometheus/mysqld\_exporter:

Exporter for MySQL server metrics

https://github.com/prometheus/mysqld\_exporter

MySQL Server Exporter

Prometheus exporter for MySQL server metrics. Supported MySQL versions: 5.1 and up. NOTE: Not all collection methods are supported on MySQL < 5.6

Building and running

Required Grants

CREATE USER 'exporter'@'localhost' IDENTIFIED BY 'XXXXXXXX' WITH MAX\_USER\_CONNECTIONS 3;

GRANT PROCESS, REPLICATION CLIENT, SELECT ON \*.\* TO 'exporter'@'localhost';

NOTE: It is recommended to set a max connection limit for the user to avoid overloading the server with monitoring scrapes under heavy load.

Build

make

Running

Running using an environment variable:

export DATA\_SOURCE\_NAME='login:password@(hostname:port)/'

./mysqld\_exporter <flags>

Running using ~/.my.cnf:

./mysqld\_exporter <flags>

Collector Flags

| **Name** | **MySQL Version** | **Description** |
| --- | --- | --- |
| collect.auto\_increment.columns | 5.1 | Collect auto\_increment columns and max values from information\_schema. |
| collect.binlog\_size | 5.1 | Collect the current size of all registered binlog files |
| collect.engine\_innodb\_status | 5.1 | Collect from SHOW ENGINE INNODB STATUS. |
| collect.engine\_tokudb\_status | 5.6 | Collect from SHOW ENGINE TOKUDB STATUS. |
| collect.global\_status | 5.1 | Collect from SHOW GLOBAL STATUS (Enabled by default) |
| collect.global\_variables | 5.1 | Collect from SHOW GLOBAL VARIABLES (Enabled by default) |
| collect.info\_schema.clientstats | 5.5 | If running with userstat=1, set to true to collect client statistics. |
| collect.info\_schema.innodb\_metrics | 5.6 | Collect metrics from information\_schema.innodb\_metrics. |
| collect.info\_schema.innodb\_tablespaces | 5.7 | Collect metrics from information\_schema.innodb\_sys\_tablespaces. |
| collect.info\_schema.processlist | 5.1 | Collect thread state counts from information\_schema.processlist. |
| collect.info\_schema.processlist.min\_time | 5.1 | Minimum time a thread must be in each state to be counted. (default: 0) |
| collect.info\_schema.query\_response\_time | 5.5 | Collect query response time distribution if query\_response\_time\_stats is ON. |
| collect.info\_schema.tables | 5.1 | Collect metrics from information\_schema.tables (Enabled by default) |
| collect.info\_schema.tables.databases | 5.1 | The list of databases to collect table stats for, or '\*' for all. |
| collect.info\_schema.tablestats | 5.1 | If running with userstat=1, set to true to collect table statistics. |
| collect.info\_schema.userstats | 5.1 | If running with userstat=1, set to true to collect user statistics. |
| collect.perf\_schema.eventsstatements | 5.6 | Collect metrics from performance\_schema.events\_statements\_summary\_by\_digest. |
| collect.perf\_schema.eventsstatements.digest\_text\_limit | 5.6 | Maximum length of the normalized statement text. (default: 120) |
| collect.perf\_schema.eventsstatements.limit | 5.6 | Limit the number of events statements digests by response time. (default: 250) |
| collect.perf\_schema.eventsstatements.timelimit | 5.6 | Limit how old the 'last\_seen' events statements can be, in seconds. (default: 86400) |
| collect.perf\_schema.eventswaits | 5.5 | Collect metrics from performance\_schema.events\_waits\_summary\_global\_by\_event\_name. |
| collect.perf\_schema.file\_events | 5.6 | Collect metrics from performance\_schema.file\_summary\_by\_event\_name. |
| collect.perf\_schema.file\_instances | 5.5 | Collect metrics from performance\_schema.file\_summary\_by\_instance. |
| collect.perf\_schema.indexiowaits | 5.6 | Collect metrics from performance\_schema.table\_io\_waits\_summary\_by\_index\_usage. |
| collect.perf\_schema.tableiowaits | 5.6 | Collect metrics from performance\_schema.table\_io\_waits\_summary\_by\_table. |
| collect.perf\_schema.tablelocks | 5.6 | Collect metrics from performance\_schema.table\_lock\_waits\_summary\_by\_table. |
| collect.slave\_status | 5.1 | Collect from SHOW SLAVE STATUS (Enabled by default) |
| collect.heartbeat | 5.1 | Collect from [heartbeat](https://github.com/prometheus/mysqld_exporter#heartbeat). |
| collect.heartbeat.database | 5.1 | Database from where to collect heartbeat data. (default: heartbeat) |
| collect.heartbeat.table | 5.1 | Table from where to collect heartbeat data. (default: heartbeat) |

General Flags

| **Name** | **Description** |
| --- | --- |
| config.my-cnf | Path to .my.cnf file to read MySQL credentials from. (default: ~/.my.cnf) |
| log.level | Logging verbosity (default: info) |
| log\_slow\_filter | Add a log\_slow\_filter to avoid exessive MySQL slow logging. NOTE: Not supported by Oracle MySQL. |
| web.listen-address | Address to listen on for web interface and telemetry. |
| web.telemetry-path | Path under which to expose metrics. |
| version | Print the version information. |

Setting the MySQL server's data source name

The MySQL server's [data source name](http://en.wikipedia.org/wiki/Data_source_name) must be set via the DATA\_SOURCE\_NAME environment variable. The format of this variable is described at <https://github.com/go-sql-driver/mysql#dsn-data-source-name>.

Using Docker

You can deploy this exporter using the [prom/mysqld-exporter](https://registry.hub.docker.com/u/prom/mysqld-exporter/) Docker image.

For example:

docker pull prom/mysqld-exporter

docker run -d -p 9104:9104 --link=my\_mysql\_container:bdd \

-e DATA\_SOURCE\_NAME="user:password@(bdd:3306)/database" prom/mysqld-exporter

heartbeat

With collect.heartbeat enabled, mysqld\_exporter will scrape replication delay measured by heartbeat mechanisms. [Pt-heartbeat](https://www.percona.com/doc/percona-toolkit/2.2/pt-heartbeat.html) is the reference heartbeat implementation supported.

Example Rules

There are some sample rules available in [example.rules](https://github.com/prometheus/mysqld_exporter/blob/master/example.rules)

# 3@基于 Prometheus 的数据库监控 - 推酷

http://www.tuicool.com/articles/JfUNVff

作者 金 戈

沃趣科技技术专家

传统监控系统面临的问题

Prometheus的前身：Borgmon

应用埋点

服务发现

指标采集与堆叠

指标数据存储

指标

指标的查询

规则计算

介绍

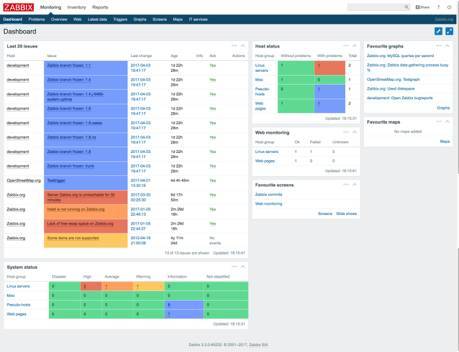
架构

数据库监控

部署服务端

传统监控系统面临的问题

传统监控系统，会面临哪些问题？



初次使用需要大量配置，随着服务器和业务的增长会发现zabbix等传统监控面临很多问题：

1. DB性能瓶颈，由于zabbix会将采集到的性能指标都存储到数据库中，当服务器数量和业务增长快速扩张时数据库性能首先成为瓶颈。
2. 多套部署，管理成本高，当数据库性能成为瓶颈时首先想到的办法可能时分多套zabbix部署，但是又会带来管理很维护成本很高的问题。
3. 易用性差，zabbix的配置和管理非常复杂，很难精通。
4. 邮件风暴，邮件配置各种规则相当复杂，一不小心可能就容易造成邮件风暴的问题。

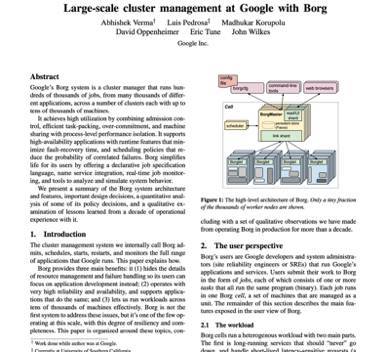
随着容器技术的发展，传统监控系统面临更多问题

1. 容器如何监控?
2. 微服务如何监控?
3. 集群性能如何进行分析计算?
4. 如何管理agent端大量配置脚本?

我们可以看到传统监控系统无法满足，当前IT环境下的监控需求

## Prometheus的前身：Borgmon

2015年Google发表了一篇论文《Google使用Borg进行大规模集群的管理》



这篇论文也描述了Google集群的规模和面临的挑战

1. 单集群上万服务器
2. 几千个不同的应用
3. 几十万个以上的jobs，而且动态增加或者减少
4. 每个数据中心数百个集群

基于这样一个规模，Google的监控系统也面临巨大挑战，而Borg中的Borgmon监控系统就是为了应对这些挑战而生。

## Borgmon介绍

那么我们来看一下Google如何做大规模集群的监控系统

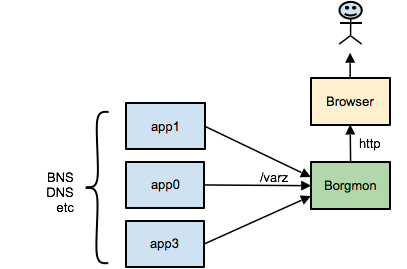
应用埋点

首先，Borg集群中运行的所有应用都需要暴露出特定的URL， http://<app>:80/varz 通过这个URL我们就可以获取到应用所暴露的全部监控指标。



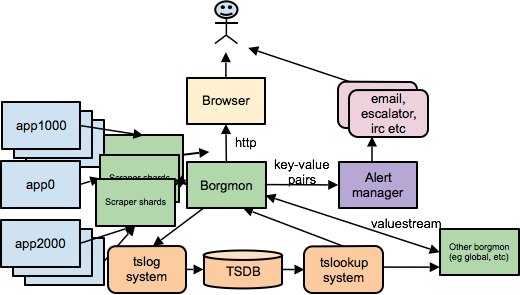
### 服务发现

然而这样的应用有数千万个，而且可能会动态增加或者减少，Borgmon中如何发现这些应用呢？Borg中的应用启动时会自动注册到Borg内部的域名服务器BNS中，Borgmon通过读取BNS中应用列表信息，收集到应用列表，从而发现有哪些应用服务需要监控。当获取到应用列表后，就会将应用的全部监控变量值拉取到Borgmon系统中。



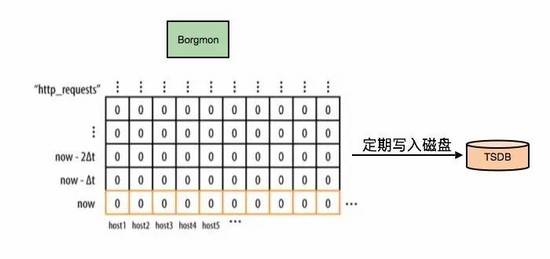
### 指标采集与堆叠

当监控指标收集到Borgmon中，就可以进行展现或者提供给告警使用，另外由于一个集群实在是太过庞大了，一个Borgmon可能无法满足整个集群的监控采集和展现需求，所以一个数据中心可能部署多个Borgmon，分为数据收集层和汇总层，数据收集层会有多个Borgmon专门用来到应用中收集数据，汇总层Borgmon则从数据收集层Borgmon中获取数据。



指标数据存储

Borgmon收集到了性能指标数据后，会把所有的数据存储在内存数据库里，定时checkpoint到磁盘上，并且会周期性的打包到外部的系统TSDB。通常情况下，数据中心和全局Borgmon中一般至少会存放12小时左右的数据量，以便渲染图表使用。每个数据点大概占用24字节的内存，所以存放100万个time-series，每个time-series每分钟一个数据点，同时保存12小时数据，仅需17GB内存。



指标

指标的查询

Borgmon中通过标签的方式查询指标，基于标签过滤我们可以查询到某个应用的具体指标，也可以查询更高维度的信息

基于标签过滤信息，比如我们基于一组过滤信息查询到host0:80这个app的http\_requests指标

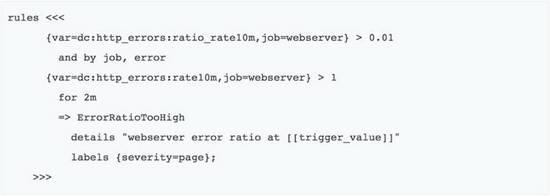
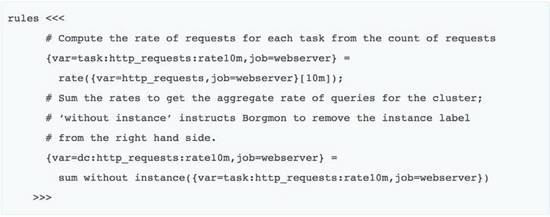
我们也可以查询到整个美国西部，job为webserver的http\_requests指标

那么这个时候拿到的就是所有符合条件的实例的http\_requests指标

规则计算

在数据收集和存储的基础之上，我们可以通过规则计算得到进一步的数据。

比如，我们想在web server报错超过一定比例的时候报警，或者说在非200返回码，占总请求的比例超过某个值的时候报警。



Prometheus

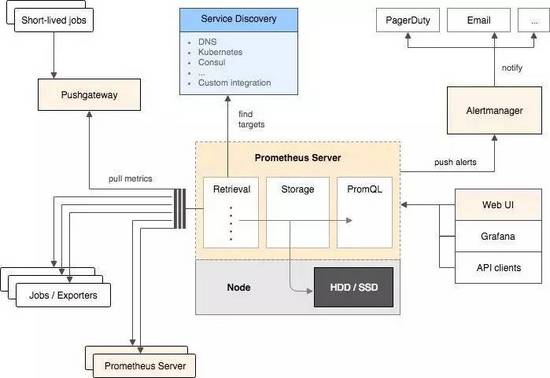
介绍

Borgmon是Google内部的系统，那么在Google之外如何使用它呢？这里就提到我们所描述的Prometheus这套监控系统。Google内部SRE工程师的著作《Google SRE》这本书中，直接就提到了Prometheus相当于就是开源版本的Borgmon。目前Prometheus在开源社区也是相当火爆，由Google发起Linux基金会旗下的原生云基金会（CNCF）就将Prometheus纳入其下第二大开源项目（第一项目为Kubernetes，为Borg的开源版本）。

架构

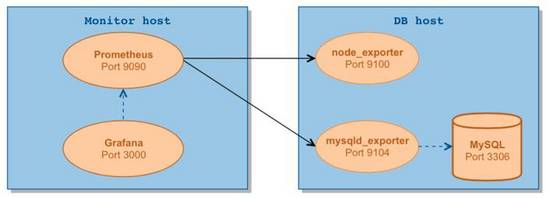
Prometheus整体架构和Borgmon类似，组件如下，有些组件是可选的：

* Prometheus主服务器，用来收集和存储时间序列数据
* 应用程序client代码库
* 短时jobs的push gateway
* 特殊用途的exporter（包括HAProxy、StatsD、Ganglia等）
* 用于报警的alertmanager
* 命令行工具查询
* 另外Grafana是作为Prometheus Dashboard展现的绝佳工具



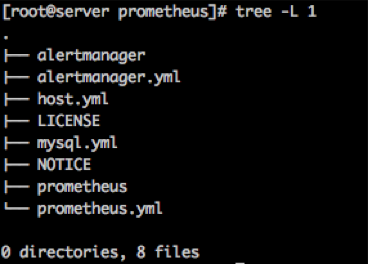
数据库监控

基于Prometheus的数据库指标采集，我们以MySQL为例，由于MySQL没有暴露采集性能指标的接口，我们可以单独启动一个mysql\_exporter，通过mysql\_exporter到MySQL数据库上抓去性能指标，并暴露出性能采集接口提供给Prometheus，另外我们可以启动node\_exporter用于抓取主机的性能指标。

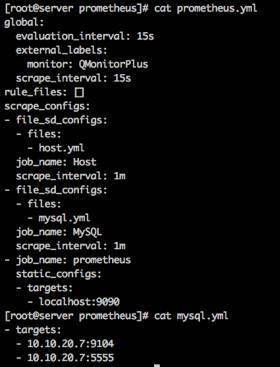


## 部署服务端

对于服务端配置非常简单，由于Prometheus全部基于Go语言开发，而Go语言程序在安装方面非常方便，安装服务端程序只需要下载，解压并运行即可。可以看到服务端常用程序也比较少，只需要包含prometheus这个主服务程序和alertmanager这个告警系统程序。



服务端配置也非常简单，常用配置包含拉取时间和具体采集方式，就我们监控mysql数据库来讲，只需要填入mysql\_exporter地址即可。



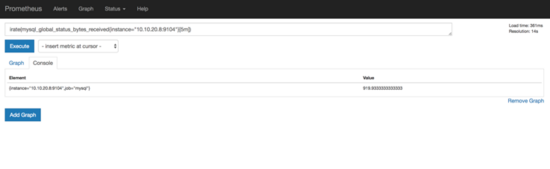
## 部署exporter端

对于mysql采集只需要配置连接信息，并启动mysql\_exporter即可

完成配置之后即可通过mysql\_exporter采集mysql性能指标

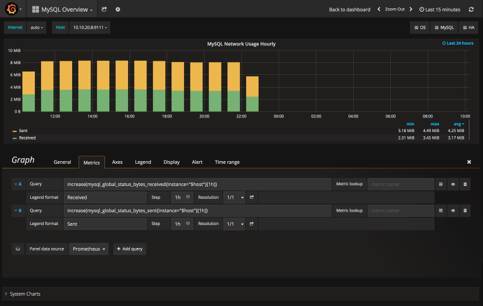


然后我们在prometheus服务端也可以查询到采集的mysql性能指标

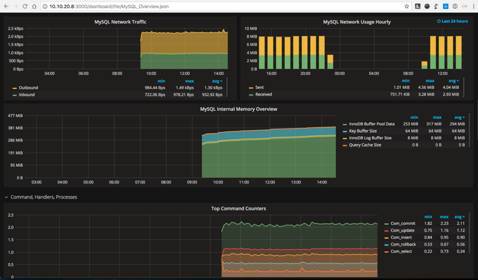


基于这些采集指标和Prometheus提供的规则计算语句，我们可以实现一些高纬度的查询需求，比如用这个语句， increase(mysql\_global\_status\_bytes\_received{instance="$host"}[1h])

我们可以查询MySQL每小时接受到的字节数，然后我们将这个查询放到Grafana中，就可以展现出非常酷炫的性能图表。



而目前结合Prometheus和Grafana的MySQL监控方案已经有开源实现，我们很轻松可以搭建一套基于Prometheus的监控系统



对于告警方面我们也可以基于Prometheus丰富的查询语句实现复杂告警逻辑

比如我们要对MySQL备库进行监控，如果复制IO线程未运行或者复制SQL线程未运行并且持续2分钟就发送告警我们可以使用如下这条告警规则。

# Alert: The replication IO or SQL threads are stopped.

**ALERT** MySQLReplicationNotRunning

IF mysql\_slave\_status\_slave\_io\_running == 0 OR mysql\_slave\_status\_slave\_sql\_running == 0

FOR 2m

LABELS {

**severity** = "critical"

}

ANNOTATIONS {

**summary** = "Slave replication is not running",

description = "Slave replication (IO or SQL) has been down for more than 2 minutes.",

}

在比如，我们要监控MySQL备库延迟大于30秒并且预测在未来2分钟之后大于0秒持续1分钟，则告警

# Alert: The replicaiton lag **is** non-zero **and** it predicted **to** **not** recover within

# 2 minutes. This allows **for** a small amount of replication lag.

ALERT MySQLReplicationLag

**IF**

(mysql\_slave\_lag\_seconds > 30)

**AND** **on** (instance)

(predict\_linear(mysql\_slave\_lag\_seconds[5m], 60\*2) > 0)

**FOR** 1m

LABELS {

severity = "critical"

}

ANNOTATIONS {

summary = "MySQL slave replication is lagging",

description = "The mysql slave replication has fallen behind and is not recovering",

}

当然在数据库方面不只是有MySQL的监控实现，目前业界也有很多其他开源实现，所以在数据库监控方面也能实现开箱即用的效果

* mysql\_exporter

https://github.com/prometheus/mysqld\_exporter

* redis\_exporter

https://github.com/oliver006/redis\_exporter

* postgres\_exporter

https://github.com/wrouesnel/postgres\_exporter

* mongodb\_exporter

https://github.com/percona/mongodb\_exporter

# 2@使用Prometheus和Grafana监控Mysql服务器性能

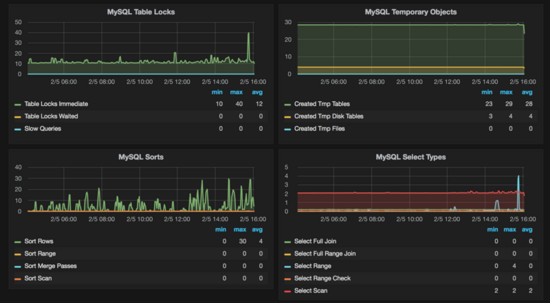
- 推酷 http://www.tuicool.com/articles/fiYZriE

时间 2016-09-29 15:12:36 [SegmentFault](http://www.tuicool.com/sites/3uEjYv" \t "_blank)

原文 [https://segmentfault.com/a/1190000007040144](https://segmentfault.com/a/1190000007040144?utm_source=tuicool&utm_medium=referral)

主题 [Grafana](http://www.tuicool.com/topics/11040110) [MySQL](http://www.tuicool.com/topics/11030000)

这是一篇快速入门文章，介绍了如何使用 Prometheus 和 Grafana 对Mysql服务器性能进行监控。内容基于 [***这篇***](https://www.percona.com/blog/2016/02/29/graphing-mysql-performance-with-prometheus-and-grafana/) 文章，结合了自己的实际实践并根据最新版本的应用进行了调整。下面是两张效果图：



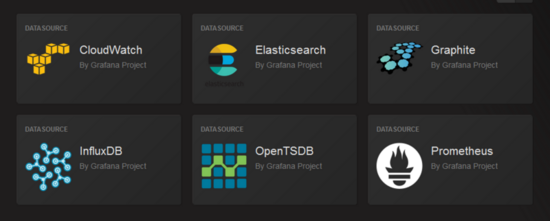


概述

[***Prometheus***](https://prometheus.io/) 是一个开源的服务监控系统，它通过HTTP协议从远程的机器收集数据并存储在本地的时序数据库上。它提供了一个简单的网页界面、一个功能强大的查询语言以及HTTP接口等等。Prometheus通过安装在远程机器上的 exporter 来收集监控数据，我们用到了以下两个exporter：

* node\_exporter – 用于机器系统数据
* mysqld\_exporter – 用于Mysql服务器数据

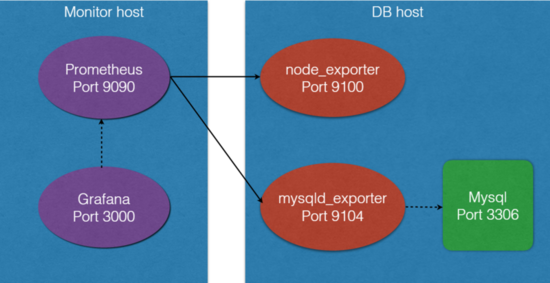
[***Grafana***](http://grafana.org/) 是一个开源的功能丰富的数据可视化平台，通常用于时序数据的可视化。它内置了以下数据源的支持：



并可以通过插件扩展支持的数据源。

架构图

下面是我们安装时用到的架构图：



安装和运行Prometheus

首先我们安装Prometheus：

$ wget https://github.com/prometheus/prometheus/releases/download/v1.1.3/prometheus-1.1.3.linux-amd64.tar.gz -O prometheus-1.1.3.linux-amd64.tar.gz

$ mkdir /usr/local/services/prometheus

$ tar zxf prometheus-1.1.3.linux-amd64.tar.gz -C /usr/local/services/prometheus --strip-components=1

然后在安装目下编辑配置文件 prometheus.yml ：

**global**:

**scrape\_interval**: 15s

**evaluation\_interval**: 15s

**scrape\_configs**:

- **job\_name**: linux

**static\_configs**:

- **targets**: ['192.168.204.63:9100']

**labels**:

**instance**: db1

- **job\_name**: mysql

**static\_configs**:

- **targets**: ['192.168.204.63:9104']

**labels**:

**instance**: db1

192.168.204.63 是我们数据库主机的IP，端口则是对应的 exporter 的监听端口。

然后我们启动Prometheus：

$ ./prometheus

INFO[0000] Starting prometheus (**version**=1.1.3, branch=master, revision=ac374aa6748e1382dbeb72a00abf47d982ee8fff) **source**=main.**go**:73

INFO[0000] Build context (**go**=go1.6.3, user=root@3e392b8b8b44, date=20160916-11:36:30) **source**=main.**go**:74

INFO[0000] Loading configuration **file** prometheus.yml **source**=main.**go**:221

INFO[0000] Loading series **map** and head chunks... **source**=storage.**go**:358

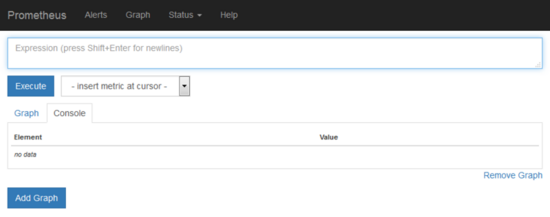
INFO[0000] 4 series loaded. **source**=storage.**go**:363

INFO[0000] Starting target manager... **source**=targetmanager.**go**:76

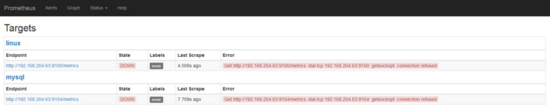
WARN[0000] No AlertManagers configured, not dispatching any alerts **source**=notifier.**go**:176

INFO[0000] Listening **on** :9090 **source**=web.**go**:235

Prometheus内置了一个web界面，我们可通过 http://monitor\_host:9090 进行访问：



在 Status -> Targets 页面下，我们可以看到我们配置的两个Target，它们的 State 为 DOWN 。



下一步我们需要安装并运行exporter。下载exporters并解压：

$ wget https://github.com/prometheus/node\_exporter/releases/download/0.12.0/node\_exporter-0.12.0.linux-amd64.tar.gz -O node\_exporter-0.12.0.linux-amd64.tar.gz

$ wget https://github.com/prometheus/mysqld\_exporter/releases/download/v0.9.0/mysqld\_exporter-0.9.0.linux-amd64.tar.gz -O mysqld\_exporter-0.9.0.linux-amd64.tar.gz

$ mkdir /usr/local/services/prometheus\_exporters

$ tar zxf node\_exporter-0.12.0.linux-amd64.tar.gz -C /usr/local/services/prometheus\_exporters --strip-components=1

$ tar zxf mysqld\_exporter-0.9.0.linux-amd64.tar.gz -C /usr/local/services/prometheus\_exporters --strip-components=1

运行node\_exporter ：

$ **cd** /usr/local/services/prometheus\_exporters

$ ./node\_exporter

INFO[0000] Starting node\_exporter (**version**=0.12.0, branch=master, revision=df8dcd2) **source**=node\_exporter.**go**:135

INFO[0000] Build context (**go**=go1.6.2, user=root@ff68505a5469, date=20160505-22:14:18) **source**=node\_exporter.**go**:136

INFO[0000] No directory specified, see --collector.textfile.directory **source**=textfile.**go**:57

INFO[0000] Enabled collectors: **source**=node\_exporter.**go**:155

INFO[0000] - vmstat **source**=node\_exporter.**go**:157

INFO[0000] - conntrack **source**=node\_exporter.**go**:157

INFO[0000] - filesystem **source**=node\_exporter.**go**:157

INFO[0000] - meminfo **source**=node\_exporter.**go**:157

INFO[0000] - netdev **source**=node\_exporter.**go**:157

INFO[0000] - stat **source**=node\_exporter.**go**:157

INFO[0000] - entropy **source**=node\_exporter.**go**:157

INFO[0000] - mdadm **source**=node\_exporter.**go**:157

INFO[0000] - sockstat **source**=node\_exporter.**go**:157

INFO[0000] - time **source**=node\_exporter.**go**:157

INFO[0000] - uname **source**=node\_exporter.**go**:157

INFO[0000] - diskstats **source**=node\_exporter.**go**:157

INFO[0000] - filefd **source**=node\_exporter.**go**:157

INFO[0000] - loadavg **source**=node\_exporter.**go**:157

INFO[0000] - netstat **source**=node\_exporter.**go**:157

INFO[0000] - textfile **source**=node\_exporter.**go**:157

INFO[0000] Listening **on** :9100 **source**=node\_exporter.**go**:176

mysqld\_exporter需要连接到Mysql，所以需要Mysql的权限，我们先为它创建用户并赋予所需的权限：

mysql> GRANT REPLICATION CLIENT, PROCESS **ON** \*.\* **TO** 'prom'@'localhost' identified by 'abc123';

mysql> GRANT SELECT **ON** performance\_schema.\* **TO** 'prom'@'localhost';

创建 .my.cnf 文件并运行mysqld\_exporter：

$ **cd** /usr/local/services/prometheus\_exporters

$ **cat** << EOF > .my.**cnf**

[client]

user=prom

password=abc123

EOF

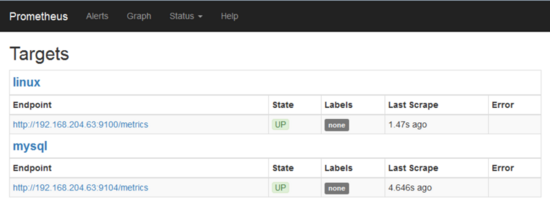
$ ./mysqld\_exporter -config.my-**cnf**=".my.cnf"

INFO[0000] Starting mysqld\_exporter (**version**=0.9.0, branch=master, revision=8400af20ccdbf6b5e0faa2c925c56c48cd78d70b) **source**=mysqld\_exporter.**go**:432

INFO[0000] Build context (**go**=go1.6.3, user=root@2c131c66ca20, date=20160926-18:28:09) **source**=mysqld\_exporter.**go**:433

INFO[0000] Listening **on** :9104 **source**=mysqld\_exporter.**go**:451

我们再次回到 Status -> Targets 页面，可以看到两个Target的状态已经变成 UP 了：



安装和运行Grafana

下载并解压Grafana：

$ wget https://grafanarel.s3.amazonaws.com/builds/grafana-3.1.1-1470047149.linux-x64.tar.gz

$ mkdir /usr/local/services/grafana

$ tar zxvf grafana-3.1.1-1470047149.linux-x64.tar.gz -C /usr/local/services/grafana --strip-components=1

编辑配置文件 /usr/local/services/grafana/conf/defaults.ini ，修改 dashboards.json 段落下两个参数的值：

**[dashboards.json]**

enabled = true

path = /var/lib/grafana/dashboards

安装仪表盘:

$ git clone https://github.com/percona/grafana-dashboards.git

$ cp -r grafana-dashboards/dashboards /var/lib/grafana

运行以下命令为Grafana打个补丁，不然图表不能正常显示：

$ sed -i 's/expr=\(.\)\.replace(\(.\)\.expr,\(.\)\.scopedVars\(.\*\)var \(.\)=\(.\)\.interval/expr=\1.replace(\2.expr,\3.scopedVars\4var \5=\1.replace(\6.interval, \3.scopedVars)/' /usr/local/services/grafana/public/app/plugins/datasource/prometheus/datasource.js

$ sed -i 's/,range\_input/.replace(\/"{\/g,"\\"").replace(\/}"\/g,"\\""),range\_input/; s/step\_input:""/step\_input:this.target.step/' /usr/local/services/grafana/public/app/plugins/datasource/prometheus/query\_ctrl.js

最后我们运行Grafana服务：

$ cd /usr/local/services/grafana/bin/

$ ./grafana-server

INFO[09-28|12:23:33] Starting Grafana logger=main version=3.1.1 **commit**=a4d2708 **compiled**=2016-08-01T18:20:16+0800

INFO[09-28|12:23:33] Config loaded **from** logger=**settings** **file**=/usr/**local**/services/grafana/conf/defaults.ini

INFO[09-28|12:23:33] **Path** Home logger=**settings** **path**=/usr/**local**/services/grafana

INFO[09-28|12:23:33] **Path** **Data** logger=**settings** **path**=/usr/**local**/services/grafana/**data**

INFO[09-28|12:23:33] **Path** **Logs** logger=**settings** **path**=/usr/**local**/services/grafana/**data**/**log**

INFO[09-28|12:23:33] **Path** Plugins logger=**settings** **path**=**data**/plugins

INFO[09-28|12:23:33] Initializing DB logger=sqlstore dbtype=mysql

INFO[09-28|12:23:33] **Starting** DB **migration** logger=migrator

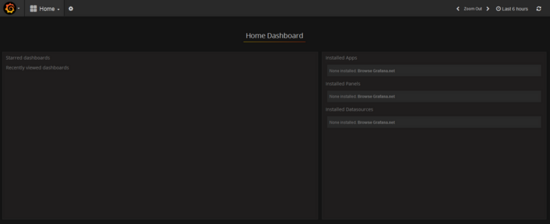
INFO[09-28|12:23:33] Creating **json** dashboard **index** **for** **path**: [/**var**/lib/grafana/dashboards]

INFO[09-28|12:23:33] **Starting** **plugin** **search** logger=plugins

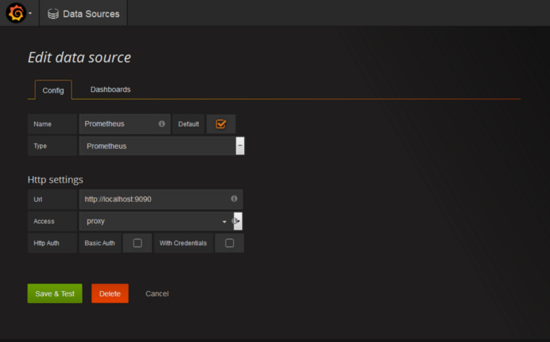
INFO[09-28|12:23:33] **Server** Listening logger=**server** address=0.0.0.0:3000 protocol=**http** subUrl=

我们可通过 http://monitor\_host:3000 访问Grafana网页界面（缺省的帐号/密码为admin/admin）：

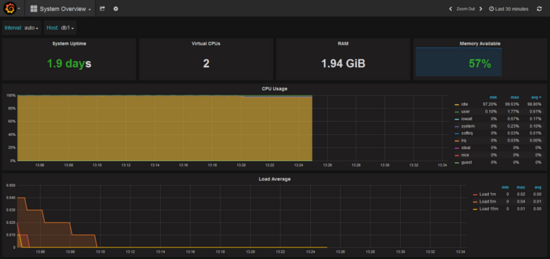




然后我们到 Data Sources 页面添加数据源：



最后我们就可以通过选择不同的仪表盘（左上角）和时间段（右上角）来呈现图表了：







参考资料

[***https://prometheus.io/***](https://prometheus.io/)

[***http://grafana.org/***](http://grafana.org/)

[***https://github.com/percona/gr...***](https://github.com/percona/grafana-dashboards)

[***https://www.percona.com/blog/...***](https://www.percona.com/blog/2016/02/29/graphing-mysql-performance-with-prometheus-and-grafana/)

# 3@MySQL 性能监控4大指标——第一部分 - OneAPM 博客

http://blog.oneapm.com/apm-tech/754.html

**【编者按】本文作者为 John Matson，主要介绍** [**mysql**](http://blog.oneapm.com/tags-mysql.html) **性能监控应该关注的4大指标。 第一部分将详细介绍前两个指标： 查询吞吐量与查询执行性能。文章系国内** [**ITOM**](http://www.oneapm.com/solutions/value.html) **管理平台** [**OneAPM**](http://www.oneapm.com/?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%B8%80%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplkd) **编译呈现。**

**MySQL 是什么？**

[MySQL](http://www.oracle.com/us/products/mysql/overview/index.html) 是现而今最流行的开源关系型数据库服务器。由 [Oracle](http://blog.oneapm.com/tags-Oracle.html) 所有，MySQL 提供了可以免费下载的社区版及包含更多特性与支持的商业版。从1995年首发以来，MySQL 衍生出多款备受瞩目的分支，诸如具有相当竞争力的 MariaDB 及 Percona。

**关键 MySQL 统计指标**

如果你的数据库运行缓慢，或者出于某种原因无法响应查询，[技术栈](http://blog.oneapm.com/tags-tech-stack.html)中每个依赖数据库的组件都会遭受性能问题。为了保证数据库的平稳运行，你可以主动监控以下四个与性能及资源利用率相关的指标：

* 查询吞吐量
* 查询执行性能
* 连接情况
* 缓冲池使用情况

MySQL 用户可以接触到数百个数据库指标，因此，在本文中，笔者将专注于能帮助我们实时了解数据库健康与性能的关键指标。

本文参考了我们在[监控入门系列文章](https://www.datadoghq.com/blog/monitoring-101-collecting-data/)中介绍的指标术语，后者为指标收集与告警提供了基础框架。

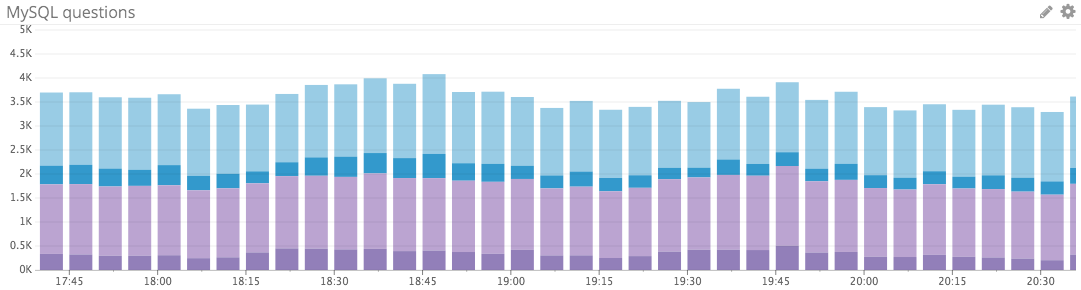
**不同版本与技术的兼容性**

本系列文章讨论的一些监控策略只适用于 MySQL 5.6与5.7版本。这些版本间的差异将在后文中提及。

本文列出的大多数指标与监控策略同样适用于与 MySQL 兼容的技术，诸如 MariaDB 与 Percona 服务器，不过带有一些明显的差别。例如，MySQL Workbench(工作台)中的一些特性（在本系列第二篇中有详细介绍）就与当下的一些 MariaDB 版本不兼容。

Amazon RDS 用户应该查看我们专门制作的 [MySQL 在 RDS](https://www.datadoghq.com/blog/monitoring-rds-mysql-performance-metrics/) 以及与 MySQL 兼容的 [Amazon Aurora](https://www.datadoghq.com/blog/monitoring-amazon-aurora-performance-metrics/) 监控手册。

## 查询吞吐量



| **名称** | **描述** | **指标类型** | **可用性** |
| --- | --- | --- | --- |
| Questions | 已执行语句（由客户端发出）计数 | Work：吞吐量 | 服务器状态变量 |
| Com\_select | SELECT 语句 | Work：吞吐量 | 服务器状态变量 |
| Writes | 插入，更新或删除 | Work：吞吐量 | 根据服务器状态变量计算得到 |

在监控任何系统时，你最关心的应该是确保系统能够高效地完成工作。数据库的工作是运行查询，因此在本例中，你的首要任务是确保 MySQL 能够如期执行查询。

MySQL 有一个名为 Questions 的内部计数器（根据 MySQL 用语，这是一个服务器状态变量），客户端每发送一个查询语句，其值就会加一。由 Questions 指标带来的以客户端为中心的视角常常比相关的Queries 计数器更容易解释。作为[存储程序](https://dev.mysql.com/doc/refman/5.6/en/stored-programs-views.html)的一部分，后者也会计算已执行语句的数量，以及诸如PREPARE 和 DEALLOCATE PREPARE 指令运行的次数，作为服务器端[预处理语句](https://dev.mysql.com/doc/refman/5.6/en/sql-syntax-prepared-statements.html)的一部分。

通过以下指令，查询诸如 Questions 或 Com\_select 服务器状态变量的值：

SHOW GLOBAL STATUS LIKE "Questions";

+---------------+--------+

| Variable\_name | Value |

+---------------+--------+

| Questions | 254408 |

+---------------+--------+

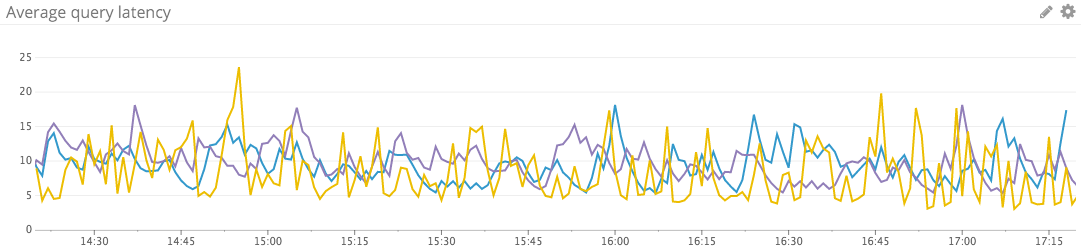
你也可以监控读、写指令的分解情况，从而更好地理解数据库的工作负载、找到可能的瓶颈。通常，读取查询会由 Com\_select 指标抓取，而写入查询则可能增加三个状态变量中某一个的值，这取决于具体的指令：

Writes = Com\_insert + Com\_update + Com\_delete

### **应该设置告警的指标：Questions**

当前的查询速率通常会有起伏，因此，如果基于固定的临界值，查询速率常常不是一个可操作的指标。但是，对于查询数量的突变设置告警非常重要——尤其是查询量的骤降，可能暗示着某个严重的问题。

## 查询性能



| **名称** | **描述** | **指标类型** | **可用性** |
| --- | --- | --- | --- |
| 查询运行时间 | 每种模式下的平均运行时间 | Work：性能 | 性能模式查询 |
| 查询错误 | 出现错误的 SQL 语句数量 | Work：错误 | 性能模式查询 |
| Slow\_queries | 超过可配置的long\_query\_time 限制的查询数量 | Work：性能 | 服务器状态变量 |

MySQL 用户监控查询延迟的方式有很多，既可以通过 MySQL 内置的指标，也可以通过查询性能模式。从[MySQL 5.6.6](https://dev.mysql.com/doc/refman/5.6/en/performance-schema-quick-start.html) 版本开始默认启用，MySQL 的 performance\_schema 数据库中的表格存储着服务器事件与查询执行的低水平统计数据。

### **性能模式语句摘要**

性能模式的 events\_statements\_summary\_by\_digest 表格中保存着许多关键指标，抓取了与每条标准化语句有关的延迟、错误和查询量信息。从该表截取的一行样例显示，某条语句被执行了两次，平均执行用时为 325 毫秒（所有计时器的测量值都以微微秒为单位）：

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SCHEMA\_NAME: employees

DIGEST: 0c6318da9de53353a3a1bacea70b4fce

DIGEST\_TEXT: SELECT \* FROM `employees` WHERE `emp\_no` > ?

COUNT\_STAR: 2

SUM\_TIMER\_WAIT: 650358383000

MIN\_TIMER\_WAIT: 292045159000

AVG\_TIMER\_WAIT: 325179191000

MAX\_TIMER\_WAIT: 358313224000

SUM\_LOCK\_TIME: 520000000

SUM\_ERRORS: 0

SUM\_WARNINGS: 0

SUM\_ROWS\_AFFECTED: 0

SUM\_ROWS\_SENT: 520048

SUM\_ROWS\_EXAMINED: 520048

...

SUM\_NO\_INDEX\_USED: 0

SUM\_NO\_GOOD\_INDEX\_USED: 0

FIRST\_SEEN: 2016-03-24 14:25:32

LAST\_SEEN: 2016-03-24 14:25:55

[摘要表](https://dev.mysql.com/doc/refman/5.7/en/performance-schema-statement-digests.html)会标准化所有语句（如上面的 DIGEST\_TEXT 一栏所示），忽略数据值，规范化空格与大小写，因此，下面的两条查询会被认为是相同的：

select \* from employees where emp\_no >200;SELECT \* FROM employees WHERE emp\_no > 80000;

想要按模式抽取出以微秒为单位的平均运行时间，你可以这样查询性能模式：

SELECT schema\_name

, SUM(count\_star) count

, ROUND( (SUM(sum\_timer\_wait) / SUM(count\_star))

/ 1000000) AS avg\_microsec

FROM performance\_schema.events\_statements\_summary\_by\_digest

WHERE schema\_name IS NOT NULL

GROUP BY schema\_name;

+--------------------+-------+--------------+

| schema\_name | count | avg\_microsec |

+--------------------+-------+--------------+

| employees | 223 | 171940 |

| performance\_schema | 37 | 20761 |

| sys | 4 | 748 |

+--------------------+-------+--------------+

相似地，按模式计算出现错误的语句总数，可以这么做：

SELECT schema\_name

, SUM(sum\_errors) err\_count

FROM performance\_schema.events\_statements\_summary\_by\_digest

WHERE schema\_name IS NOT NULL

GROUP BY schema\_name;

+--------------------+-----------+

| schema\_name | err\_count |

+--------------------+-----------+

| employees | 8 |

| performance\_schema | 1 |

| sys | 3 |

+--------------------+-----------+

### **sys 模式**

用上面的方式查询性能模式能以编程方式有效地从数据库中检索出指标。然而，对于特别查询或调查，使用 MySQL 的 [sys 模式](https://github.com/mysql/mysql-sys)通常更为简单。sys 模式以人们更易读的格式提供了一个有条理的指标集合，使得对应的查询更加简单。例如，想要找出最慢的语句（运行时间在95名开外）：

SELECT \* FROM sys.statements\_with\_runtimes\_in\_95th\_percentile;

或者查看哪些标准化语句出现了错误：

SELECT \* FROM sys.statements\_with\_errors\_or\_warnings;

在 sys 模式的[文档](https://github.com/mysql/mysql-sys)中，详细介绍了许多有用的例子。sys 模式在 MySQL 5.7.7 版本中是默认包含的。不过，MySQL 5.6 用户通过简单的几个指令就能安装它。

### **慢查询**

除了性能模式与 sys 模式中丰富的性能数据，MySQL 还提供了一个 Slow\_queries 计数器，每当查询的执行时间超过 long\_query\_time 参数指定的值之后，该计数器就会增加。默认情况下，该临界值设置为10秒。

SHOW VARIABLES LIKE 'long\_query\_time';

+-----------------+-----------+

| Variable\_name | Value |

+-----------------+-----------+

| long\_query\_time | 10.000000 |

+-----------------+-----------+

long\_query\_time 参数的值可通过一条指令进行调整。例如，将慢查询临界值设置为5秒：

SET GLOBAL long\_query\_time = 5;

（请注意，你可能要关闭会话，再重新连接至数据库，这些更改才能在会话层生效。）

### **调查查询性能问题**

如果你的查询运行得比预期要慢，很可能是某条最近修改的查询在捣鬼。如果没有发现特别缓慢的查询，接下来就该评估系统级指标，寻找核心资源（CPU，磁盘 I/O，内存以及网络）的限制。CPU 饱和与 I/O 瓶颈是常见的问题根源。你可能还想检查 Innodb\_row\_lock\_waits 指标，该指标记录着 InnoDB 存储引擎不得不停下来获得某行的锁定的次数。从 MySQL 5.5 版本起，InnoDB 就是默认的存储引擎，MySQL 对 InnoDB 表使用行级锁定。

为了提高读取与写入操作的速度，许多用户会想通过调整 InnoDB 使用的[缓冲池](https://dev.mysql.com/doc/refman/5.6/en/innodb-buffer-pool.html)大小来缓存表与索引数据。本文的第二部分会对监控与调整缓冲池大小做详细解读。

### **应该设置告警的指标**

* 查询运行时间：管理关键数据库的延迟至关重要。如果生产环境中数据库的平均查询运行时间开始下降，应该寻找数据库实例的资源限制，行锁或表锁间可能的争夺，以及客户端查询模式的变化情况。
* 查询错误：查询错误的猛增可能暗示着客户端应用或数据库本身的问题。你可以使用 sys 模式快速查找可能导致问题的查询。例如，列举出返回错误数最多的10条标准化语句：

SELECT \* FROM sys.statements\_with\_errors\_or\_warnings

ORDER BY errors DESC LIMIT 10;

* Slow\_queries：如何定义慢查询（并由此设置 long\_query\_time 参数）取决于你的用户案例。但是，无论你如何定义“慢”，你都会想知道慢查询的数量是否超出了基准水平。为了找出真正执行缓慢的查询，你可以询问 sys 模式，或深入了解 MySQL 提供的慢查询日志（该功能默认是禁用的）。有关启用并读取慢查询日志的更多信心，请参考 [MySQL 文档](http://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html)。

敬请期待本文第二部分，主要介绍 MySQL 连接与缓冲池。

本文系 [OneAPM](http://www.oneapm.com/?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%B8%80%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplkd) 工程师编译整理。[OneAPM Cloud Insight](http://www.oneapm.com/ci/feature.html?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%B8%80%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplkd) 集监控、管理、计算、协作、可视化于一身，帮助所有 IT 公司，减少在系统监控上的人力和时间成本投入，让运维工作更加高效、简单。想阅读更多技术文章，请访问 [OneAPM 官方技术博客](http://blog.oneapm.com/?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%B8%80%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplkd)。

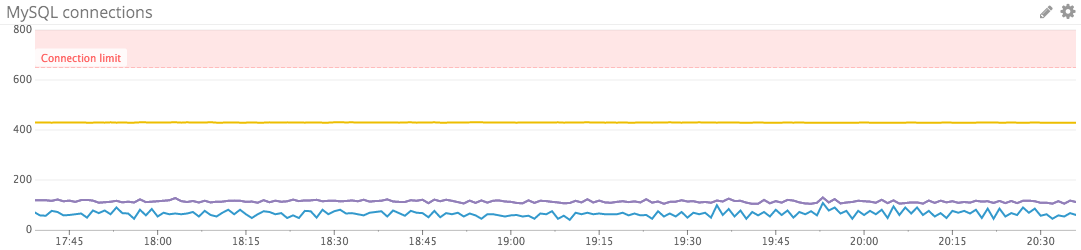
原文地址：https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/

# MySQL 性能监控4大指标——第二部分 - OneAPM 博客

http://blog.oneapm.com/apm-tech/755.html

【编者按】本文作者为 John Matson，主要介绍 [mysql](http://blog.oneapm.com/tags-mysql.html) 性能监控应该关注的4大指标。 [第一部分](http://blog.oneapm.com/apm-tech/754.html?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%BA%8C%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplnf)介绍了前两个指标：查询吞吐量与查询执行性能。本文将继续介绍另两个指标：MySQL 连接与缓冲池。文章系国内[ITOM](http://www.oneapm.com/solutions/value.html) 管理平台 [OneAPM](http://www.oneapm.com/?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%BA%8C%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplnf) 编译呈现。

## 连接



| **名称** | **描述** | **指标类型** | **可用性** |
| --- | --- | --- | --- |
| Threads\_connected | 当前开放的连接 | 资源: 利用率 | 服务器状态变量 |
| Threads\_running | 当前运行的连接 | 资源: 利用率 | 服务器状态变量 |
| Connection\_errors\_internal | 由服务器错误导致的失败连接数 | 资源: 错误 | 服务器状态变量 |
| Aborted\_connects | 尝试与服务器进行连接结果失败的次数 | 资源: 错误 | 服务器状态变量 |
| Connection\_errors\_max\_connections | 由 max\_connections 限制导致的失败连接数 | 资源: 错误 | 服务器状态变量 |

### **检查并设置连接限制**

监控客户端连接情况相当重要，因为一旦可用连接耗尽，新的客户端连接就会遭到拒绝。MySQL 默认的连接数限制为 151，可通过下面的查询加以验证：

SHOW VARIABLES LIKE 'max\_connections';

+-----------------+-------+

| Variable\_name | Value |

+-----------------+-------+

| max\_connections | 151 |

+-----------------+-------+

[MySQL 的文档](https://dev.mysql.com/doc/refman/5.7/en/too-many-connections.html)指出，健壮的服务器应该能够处理成百上千的连接数。

“常规情况下，Linux 或 Solaris 应该能够支持500到1000个同时连接。如果可用的 RAM 较大，且每个连接的工作量较低或目标响应时间较为宽松，则最多可处理10000个连接。而 Windows 能处理的连接数一般不超过2048个，这是由于该平台上使用的 Posix 兼容层。”

连接数限制可以在系统运行时进行调整：

SET GLOBAL max\_connections = 200;

然而，此设置会在服务器重启时恢复为默认值。想要永久地改变连接数限制，可以在 my.cnf 配置文件中添加如下配置（查看[本文](http://stackoverflow.com/questions/2482234/how-to-know-mysql-my-cnf-location)了解如何定位配置文件）：

max\_connections = 200

### **监控连接使用率**

MySQL 提供了 Threads\_connected 指标以记录连接的线程数——[每个连接对应一个线程](https://dev.mysql.com/doc/refman/5.7/en/connection-threads.html)。通过监控该指标与先前设置的连接限制，你可以确保服务器拥有足够的容量处理新的连接。MySQL 还提供了Threads\_running 指标，帮助你分隔在任意时间正在积极处理查询的线程与那些虽然可用但是闲置的连接。

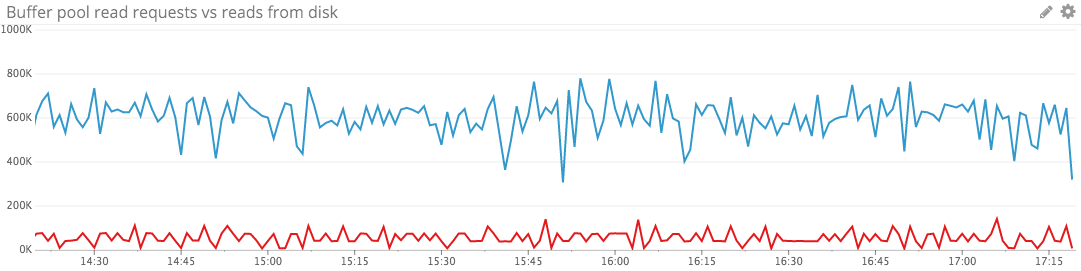
如果服务器真的达到 max\_connections 限制，它就会开始拒绝新的连接。在这种情况下，Connection\_errors\_max\_connections 指标就会开始增加，同时，追踪所有失败连接尝试的Aborted\_connects 指标也会开始增加。

MySQL 提供了许多有关连接错误的指标，帮助你调查连接问题。Connection\_errors\_internal 是个很值得关注的指标，因为该指标只会在错误源自服务器本身时增加。内部错误可能反映了内存不足状况，或者服务器无法开启新的线程。

### **应该设置告警的指标**

* Threads\_connected：当所有可用连接都被占用时，如果一个客户端试图连接至 MySQL，后者会返回 “Too many connections(连接数过多)”错误，同时将 Connection\_errors\_max\_connections 的值增加。为了防止出现此类情况，你应该监控可用连接的数量，并确保其值保持在 max\_connections 限制以内。
* Aborted\_connects：如果该计数器在不断增长，意味着用户尝试连接到数据库的努力全都失败了。此时，应该借助 Connection\_errors\_max\_connections 与 Connection\_errors\_internal 之类细粒度高的指标调查该问题的根源。

## 缓冲池使用情况



| **名称** | **描述** | **指标类型** | **可用性** |
| --- | --- | --- | --- |
| Innodb\_buffer\_pool\_pages\_total | 缓冲池中的总页数 | 资源: 利用率 | 服务器状态变量 |
| 缓冲池使用率 | 缓冲池中已使用页数所占的比率 | 资源: 利用率 | 根据服务器状态变量计算得到 |
| Innodb\_buffer\_pool\_read\_requests | 向缓冲池发送的请求 | 资源: 利用率 | 服务器状态变量 |
| Innodb\_buffer\_pool\_reads | 缓冲池无法满足的请求 | 资源: 饱和度 | 服务器状态变量 |

MySQL 默认的存储引擎 InnoDB 使用了一片称为缓冲池的内存区域，用于缓存数据表与索引的数据。缓冲池指标属于资源指标，而非工作指标，前者更多地用于调查（而非检测）性能问题。如果数据库性能开始下滑，而磁盘 I/O 在不断攀升，扩大缓冲池往往能带来性能回升。

### **检查缓冲池的大小**

默认设置下，缓冲池的大小通常相对较小，为 128MiB。不过，MySQL 建议可将其扩大至[专用数据库服务器物理内存的80%大小](http://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size)。然而，MySQL 也指出了一些注意事项：InnoDB 的内存开销可能提高超过缓冲池大小10%的内存占用。并且，如果你耗尽了物理内存，系统会求助于分页，导致数据库性能严重受损。

缓冲池也可以划分为不同的区域，称为实例。使用多个实例可以[提高大容量(多 GiB)缓冲池的并发性](https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_buffer_pool_instances)。

缓冲池大小调整操作是分块进行的，缓冲池的大小必须为块的大小乘以实例的数目再乘以某个倍数。

innodb\_buffer\_pool\_size = N \* innodb\_buffer\_pool\_chunk\_size

\* innodb\_buffer\_pool\_instances

块的默认大小为 128 MiB，但是从 MySQL 5.7.5 开始可以[自行配置](https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_buffer_pool_chunk_size)。以上两个参数的值都可以通过如下方式进行检查：

SHOW GLOBAL VARIABLES LIKE "innodb\_buffer\_pool\_chunk\_size";

SHOW GLOBAL VARIABLES LIKE "innodb\_buffer\_pool\_instances";

如果 innodb\_buffer\_pool\_chunk\_size 查询没有返回结果，则表示在你使用的 MySQL 版本中此参数无法更改，其值为 128 MiB。

在服务器启动时，你可以这样设置缓冲池的大小以及实例的数量：

$ mysqld --innodb\_buffer\_pool\_size=8G --innodb\_buffer\_pool\_instances=16

在 MySQL 5.7.5 版本，你可以通过 SET 指令[在系统运行时修改缓冲池的大小](https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool-resize.html#innodb-buffer-pool-online-resize)，并精确到字节数。例如，假设有两个缓冲池实例，你可以将其总大小设置为 8 GiB，这样每个实例的大小即为 4 GiB。

SET GLOBAL innodb\_buffer\_pool\_size=8589934592;

### **关键的 InnoDB 缓冲池指标**

MySQL 提供了许多关于缓冲池及其利用率的指标。其中一些有用的指标能够追踪缓冲池的总大小，缓冲池的使用量，以及其处理读取操作的效率。

指标 Innodb\_buffer\_pool\_read\_requests 及 Innodb\_buffer\_pool\_reads 对于理解缓冲池利用率都非常关键。Innodb\_buffer\_pool\_read\_requests 追踪合理读取请求的数量，而 Innodb\_buffer\_pool\_reads 追踪缓冲池无法满足，因而只能从磁盘读取的请求数量。我们知道，从内存读取的速度比从磁盘读取通常要快好几个数量级，因此，如果 Innodb\_buffer\_pool\_reads 的值开始增加，意味着数据库性能大有问题。

缓冲池利用率是在考虑扩大缓冲池之前应该检查的重要指标。利用率指标无法直接读取，但是可以通过下面的方式简单地计算得到：

(Innodb\_buffer\_pool\_pages\_total - Innodb\_buffer\_pool\_pages\_free) /

Innodb\_buffer\_pool\_pages\_total

如果你的数据库从磁盘进行大量读取，而缓冲池还有许多闲置空间，这可能是因为缓存最近才清理过，还处于热身阶段。如果你的缓冲池并未填满，但能有效处理读取请求，则说明你的数据工作集相当适应目前的内存配置。

然而，较高的缓冲池利用率并不一定意味着坏消息，因为旧数据或不常使用的数据会根据 [LRU 算法](https://en.wikipedia.org/wiki/Cache_algorithms#LRU) 自动从缓存中清理出去。但是，如果缓冲池无法有效满足你的读取工作量，这可能说明扩大缓存的时机已至。

### **将缓冲池指标转化为字节**

大多数缓冲池指标都以内存页面为单位进行记录，但是这些指标也可以转化为字节，从而使其更容易与缓冲池的实际大小相关联。例如，你可以使用追踪缓冲池中内存页面总数的服务器状态变量找出缓冲池的总大小（以字节为单位）：

Innodb\_buffer\_pool\_pages\_total \* innodb\_page\_size

InnoDB 页面大小是可调整的，但是默认设置为 16 KiB，或 16,384 字节。你可以使用 SHOW VARIABLES 查询了解其当前值：

SHOW VARIABLES LIKE "innodb\_page\_size";

**结论**

在本文中，我们介绍了许多你应该加以监控从而了解 MySQL 活动与性能表现的重要指标。如果你正在踌躇 MySQL 监控方案，抓取下面列出的指标能让你真正理解数据库的使用模式与可能的限制情况。这些指标也能帮助你发现，何时扩展服务器内存或将数据库移至更为强大的主机，从而保持良好的应用性能。

* 查询吞吐量
* 查询延迟与错误
* 客户端连接与错误
* 缓冲池利用率

**鸣谢**

非常感谢来自 [Oracle](http://blog.oneapm.com/tags-Oracle.html) 的 Dave Stokes 与 VividCortex 的 Ewen Fortune，他们在本文发布之前提供了许多宝贵的反馈意见。

本文系 [OneAPM](http://www.oneapm.com/?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%BA%8C%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplnf)工程师编译整理。[OneAPM Cloud Insight](http://www.oneapm.com/ci/feature.html?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%BA%8C%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplnf) 集监控、管理、计算、协作、可视化于一身，帮助所有 IT 公司，减少在系统监控上的人力和时间成本投入，让运维工作更加高效、简单。想阅读更多技术文章，请访问 [OneAPM 官方技术博客](http://blog.oneapm.com/?utm_source=Community&utm_medium=Article&utm_term=MySQL%20%E6%80%A7%E8%83%BD%E7%9B%91%E6%8E%A74%E5%A4%A7%E6%8C%87%E6%A0%87%E2%80%94%E2%80%94%E7%AC%AC%E4%BA%8C%E9%83%A8%E5%88%86&utm_content=wk613-619&utm_campaign=CiArti&from=jscwplnf)。

原文地址：<https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/>

# 3@percona/grafana-dashboards:

Grafana dashboards for MySQL and MongoDB monitoring using Prometheus

https://github.com/percona/grafana-dashboards

Grafana dashboards for MySQL and MongoDB monitoring using Prometheus

This is a set of Grafana dashboards for database and system monitoring using Prometheus datasource.

* Amazon RDS OS metrics (CloudWatch datasource)
* Cross Server Graphs
* Disk Performance
* Disk Space
* MariaDB
* MongoDB Cluster Summary
* MongoDB Overview
* MongoDB ReplSet
* MongoDB RocksDB
* MongoDB WiredTiger
* MongoDB MMAPv1
* MySQL InnoDB Metrics
* MySQL InnoDB Metrics Advanced
* MySQL MyISAM Metrics
* MySQL Overview
* MySQL Performance Schema
* MySQL Query Response Time
* MySQL Replication
* MySQL Table Statistics
* MySQL TokuDB Graphs
* MySQL User Statistics
* PXC/Galera Cluster Overview
* PXC/Galera Graphs
* Prometheus
* ProxySQL Overview
* Summary Dashboard
* System Overview
* Trends Dashboard

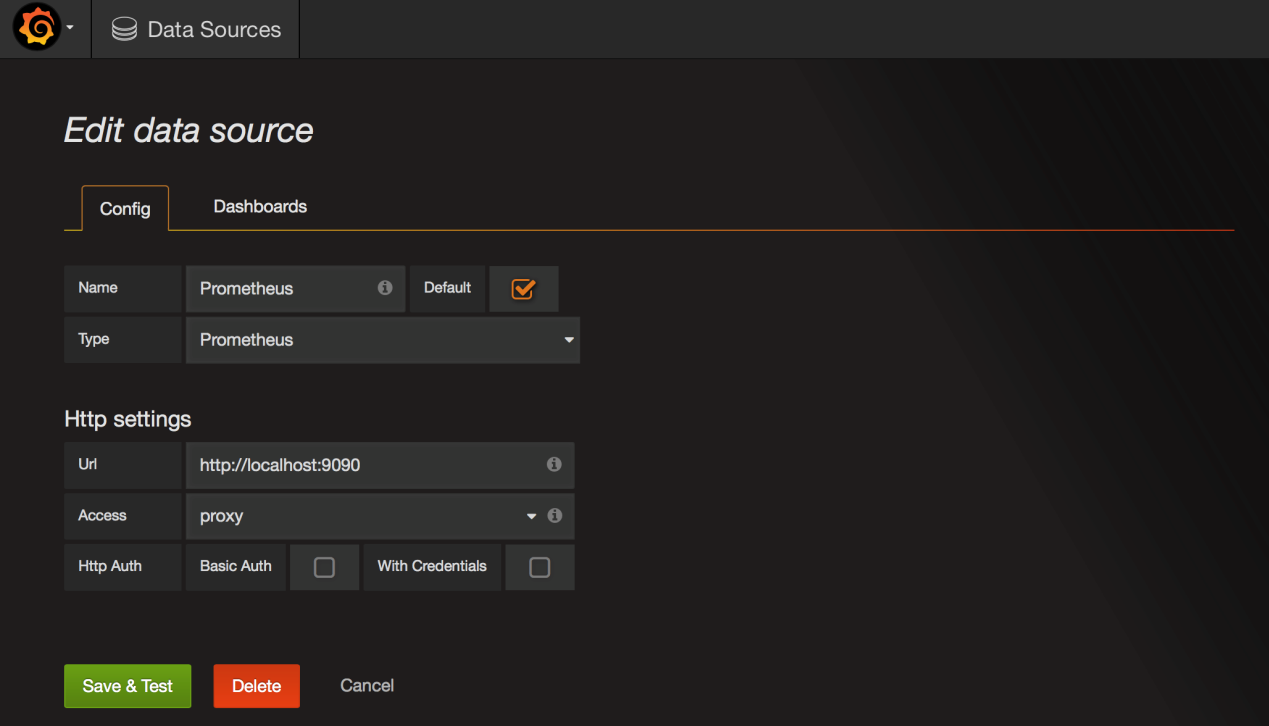
These dashboards are also a part of [Percona Monitoring and Management](https://www.percona.com/doc/percona-monitoring-and-management/index.html) project.

Live demo is available at <https://pmmdemo.percona.com/graph/>

Setup instructions

Add datasource in Grafana

The datasource should be named Prometheus so it is automatically picked up by the graphs.

[](https://github.com/percona/grafana-dashboards/blob/master/assets/datasource.png)

Prometheus config

The dashboards use built-in instance label to filter on individual hosts. It is recommended you give the good names to your instances. Here is some example:

scrape\_configs:

- job\_name: prometheus

static\_configs:

- targets: ['localhost:9090']

labels:

instance: prometheus

- job\_name: linux

static\_configs:

- targets: ['192.168.1.7:9100']

labels:

instance: db1

- job\_name: mysql

static\_configs:

- targets: ['192.168.1.7:9104']

labels:

instance: db1

How you name jobs is not important. However, "Prometheus" dashboard assumes the job name is prometheus.

Exporter options

Here is the minimal set of options for the exporters:

* node\_exporter: -collectors.enabled="diskstats,filefd,filesystem,loadavg,meminfo,netdev,stat,time,uname,vmstat"
* mysqld\_exporter: -collect.binlog\_size=true -collect.info\_schema.processlist=true
* mongodb\_exporter: the defaults are fine.

Edit Grafana config

Enable JSON dashboards by uncommenting those lines in grafana.ini:

[dashboards.json]

enabled = true

path = /var/lib/grafana/dashboards

If you wish you may import the individual dashboards via UI and ignore this and the next two steps.

Install dashboards

git clone https://github.com/percona/grafana-dashboards.git

cp -r grafana-dashboards/dashboards /var/lib/grafana/

Restart Grafana

service grafana-server restart

Apply patch (only Grafana 3.x)

If you are using Grafana 3.x you need to apply a small patch on your installation to allow the interval template variable in Stepfield of graph editor page to get the good zoomable graphs. For more information, take a look at [PR#5839](https://github.com/grafana/grafana/pull/5839).

sed -i 's/expr=\(.\)\.replace(\(.\)\.expr,\(.\)\.scopedVars\(.\*\)var \(.\)=\(.\)\.interval/expr=\1.replace(\2.expr,\3.scopedVars\4var \5=\1.replace(\6.interval, \3.scopedVars)/' /usr/share/grafana/public/app/plugins/datasource/prometheus/datasource.js

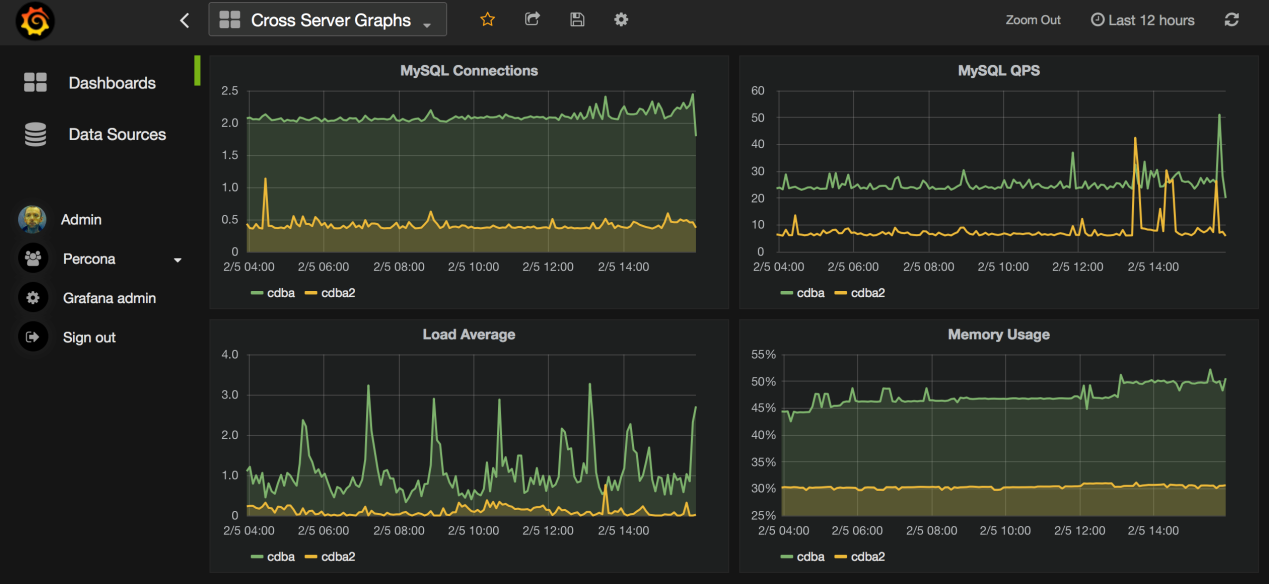
sed -i 's/,range\_input/.replace(\/"{\/g,"\\"").replace(\/}"\/g,"\\""),range\_input/; s/step\_input:""/step\_input:this.target.step/' /usr/share/grafana/public/app/plugins/datasource/prometheus/query\_ctrl.js

Update instructions

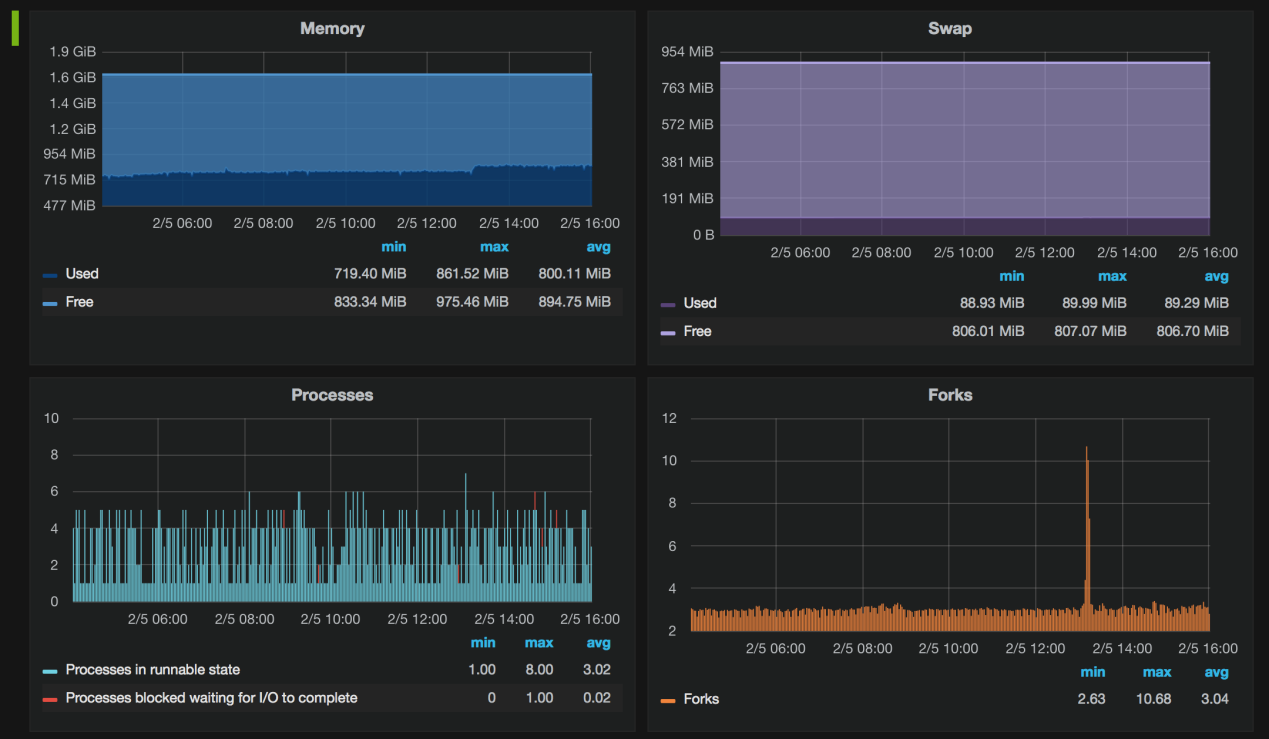
Simply copy the new dashboards to /var/lib/grafana/dashboards and restart Grafana or re-import them.

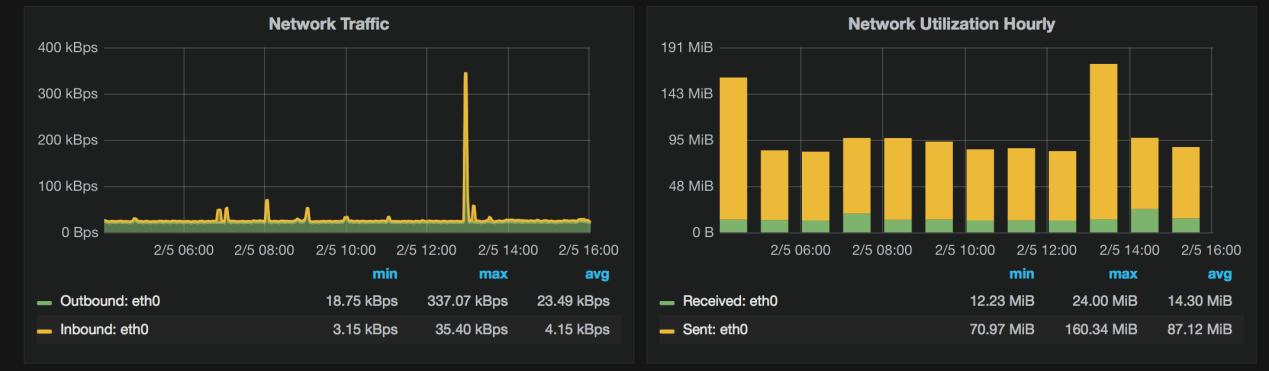
Graph samples

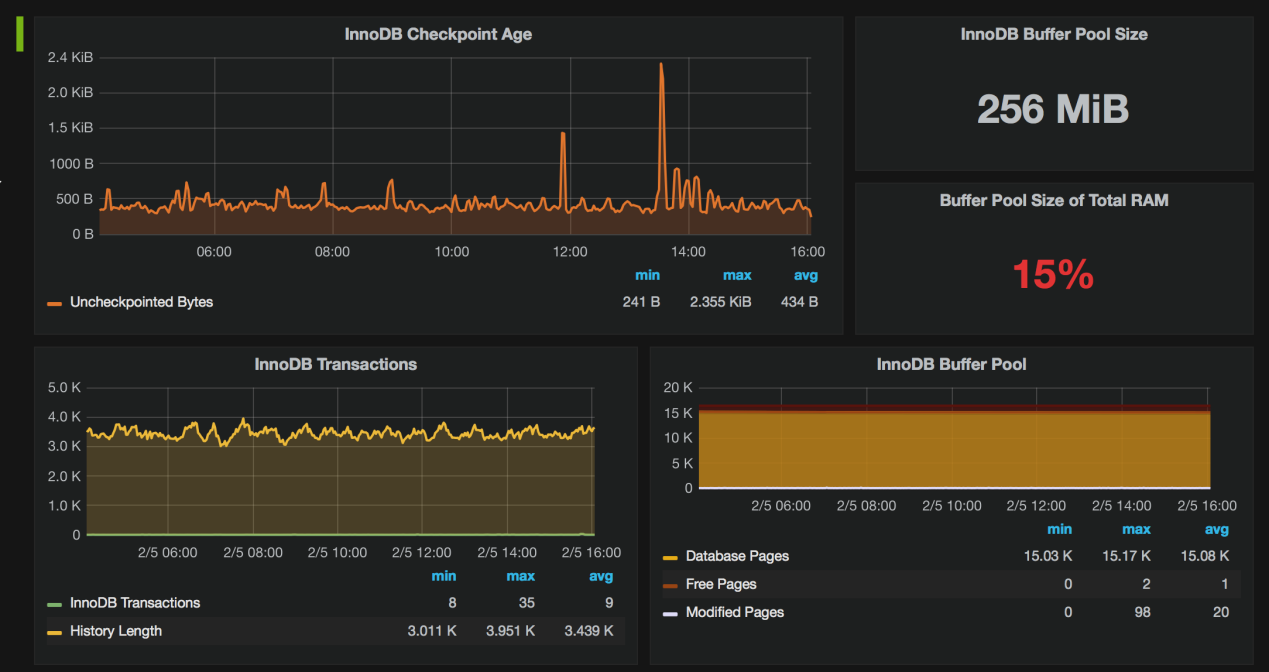
Here is some sample graphs.

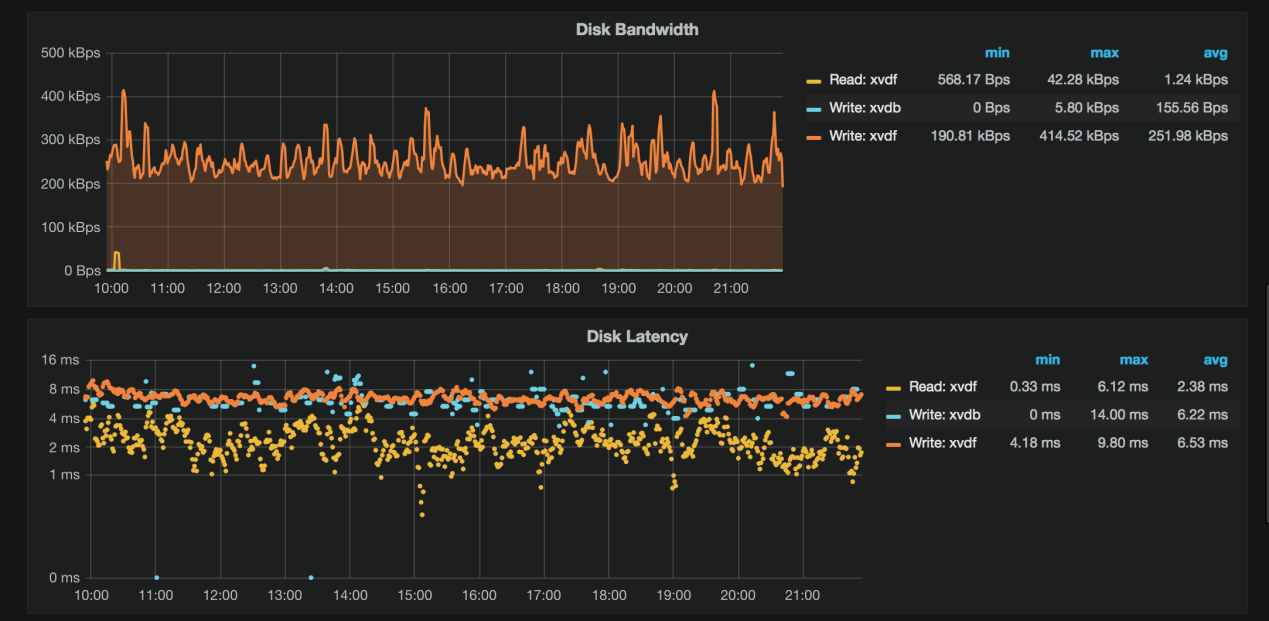
[](https://github.com/percona/grafana-dashboards/blob/master/assets/sample1.png)

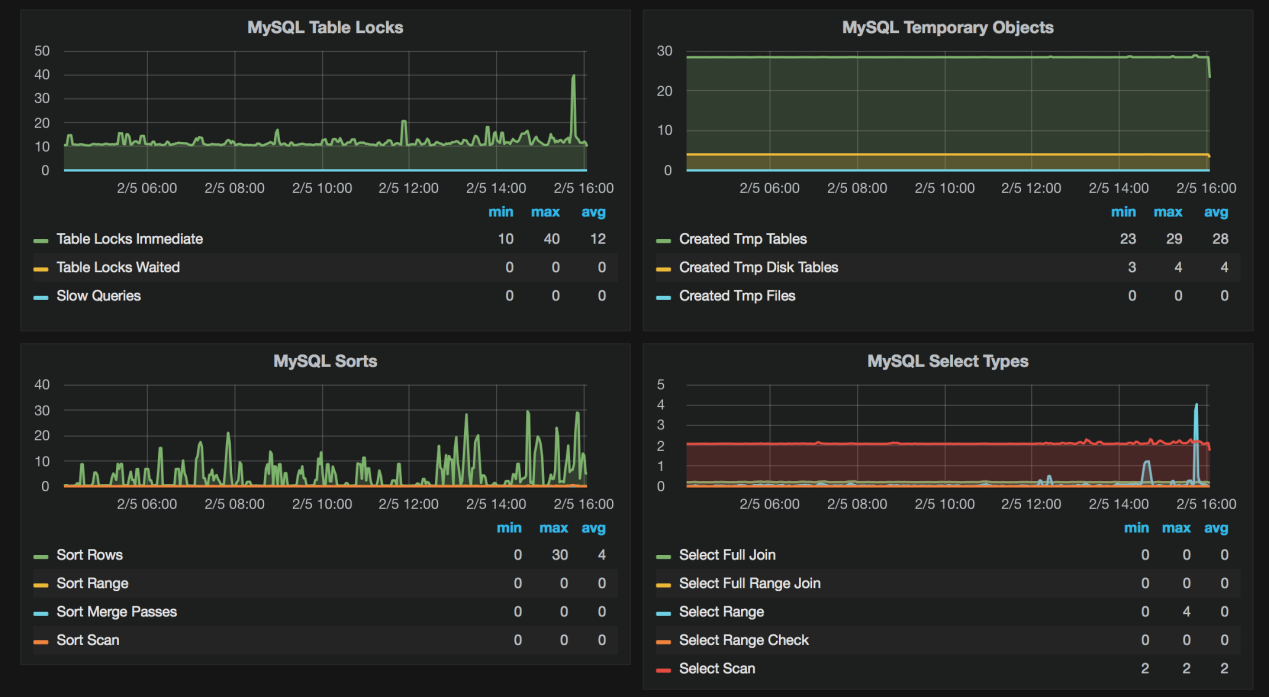
[](https://github.com/percona/grafana-dashboards/blob/master/assets/sample2.png)

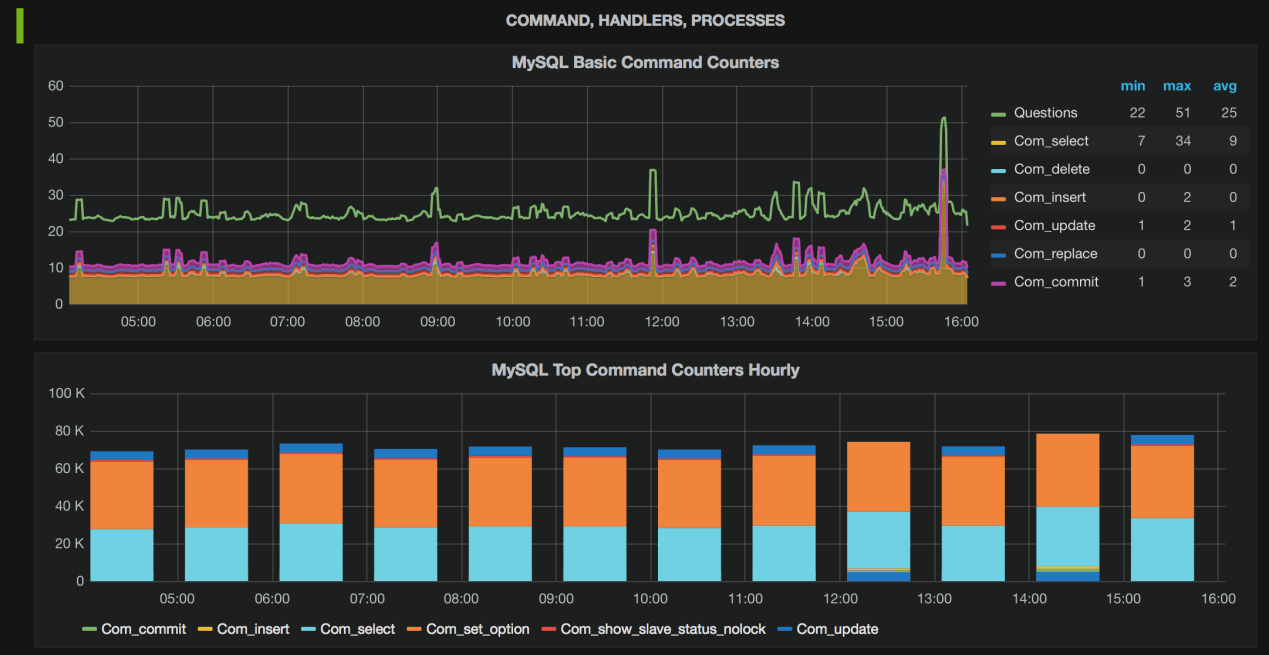
[](https://github.com/percona/grafana-dashboards/blob/master/assets/sample3.png)

[](https://github.com/percona/grafana-dashboards/blob/master/assets/sample4.png)

[](https://github.com/percona/grafana-dashboards/blob/master/assets/sample5.png)

[](https://github.com/percona/grafana-dashboards/blob/master/assets/sample6.png)

[](https://github.com/percona/grafana-dashboards/blob/master/assets/sample7.png)

[](https://github.com/percona/grafana-dashboards/blob/master/assets/sample8.png)

# How to count TPS? · Issue #198 · prometheus/mysqld\_exporter

https://github.com/prometheus/mysqld\_exporter/issues/198

TPS = (mysql\_global\_status\_commands\_total{command="commit"}+mysql\_global\_status\_commands\_total{command="rollback"})

This is done slightly differently with Prometheus, as you are trying to deal with two labels on the same metric. The + operator is designed for handling addition of different metric names. What you're hitting is a labeling miss-match.

Thankfully, this is a pretty easy operation in PromQL.

The query you're looking for is this:

sum(rate(mysql\_global\_status\_commands\_total{command=~"(commit|rollback)"}[5m])) without (command)

With PromQL, we can use label filtering to select the metrics we're interested in. Then we apply the rate function to give us the "per second" of those metrics. Then we sum up all of the metrics, but only without the command label. This will keep all of the other labels in the metric.

This might be a good one for our recording rules example. Do you have a link to the MySQL documentation that talks about this?

# Record "Transactions per second"

# See: https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos\_transaction

job:mysql\_transactions:rate5m = sum(rate(mysql\_global\_status\_commands\_total{command=~"(commit|rollback)"}[5m])) without (command)

# Maximal write througput in MySQL - Percona Database Performance Blog

https://www.percona.com/blog/2010/02/28/maximal-write-througput-in-mysql/

[Vadim Tkachenko](https://www.percona.com/blog/author/vadim/)  | February 28, 2010 |  Posted In: [Benchmarks](https://www.percona.com/blog/category/benchmarks/), [MySQL](https://www.percona.com/blog/category/mysql/)

I recently was asked what maximal amount transactions per second we can get using MySQL and XtraDB / InnoDB storage engine if we have high-end server. Good questions, though not easy to answer, as it depends on:

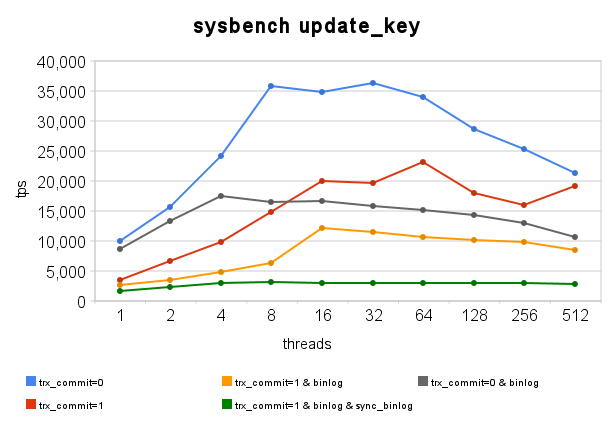
– durability setting ( innodb\_flush\_log\_at\_trx\_commit = 0 or 1 ) ?  
– do we use binary logs ( I used ROW based replication for 5.1)  
– do we have sync\_binlog options.

So why would not take these as variable parameters and run simple benchmark.  
I took sysbench update\_key scenario ( update indexed field on simple table)  
and used[Dell PowerEdge R900](https://www.percona.com/docs/wiki/benchmark:hardware:R900) with 16 cores, FusionIO as storage for table and RAID 10 with BBU as storage for innodb log files, innodb system table space and binary logs. And I used Percon-XtraDB-5.1.43-9.1 for benchmarks. All used partitions are formatted in XFS and mounted with nobarrier option.

I run update key for various threads and with next parameters

* trx\_commit=0 : innodb\_flush\_log\_at\_trx\_commit = 0 and no binary logs
* trx\_commit=1 : innodb\_flush\_log\_at\_trx\_commit = 1 and no binary logs
* trx\_commit=0 & binlog : innodb\_flush\_log\_at\_trx\_commit = 0 and binary logs
* trx\_commit=1 & binlog : innodb\_flush\_log\_at\_trx\_commit = 1 and binary logs
* trx\_commit=1 & binlog & sync\_bin : innodb\_flush\_log\_at\_trx\_commit = 1 and binary logs and sync\_binlog=1

There are results I get:

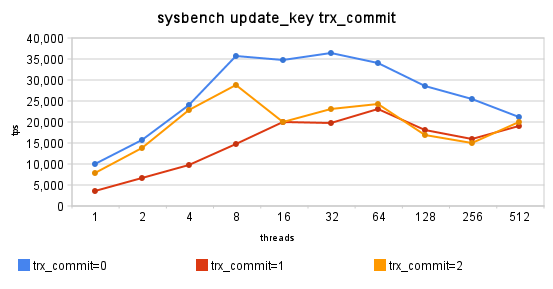
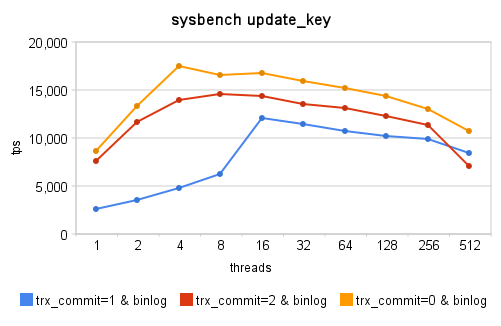
[](https://www.percona.com/blog/wp-content/uploads/2010/02/sysbench_update_key.png)

I found results being quite interesting.  
with innodb\_flush\_log\_at\_trx\_commit = 0 maximal tps is **36332.02** tps, which drops to **23115.04** tps as  
we switch to innodb\_flush\_log\_at\_trx\_commit = 1. As we use RAID10 with BBU, I did not expect the drops is going to be significant. In second case InnoDB spends

With enabling binary logs, the results drops to **17451.01** tps with innodb\_flush\_log\_at\_trx\_commit = 0 and to **12097.39**tps with innodb\_flush\_log\_at\_trx\_commit = 1. So with binary logs serialization is getting even worse.

Enabling sync\_binlog makes things really bad, and maximal results I have is  
**3086.7** tps. So this is good decision if binary log protection is worth such drop.

**UPDATE** ( 3/4/2010 )

Results with innodb\_flush\_log\_at\_trx\_commit = 2  
[](https://www.percona.com/blog/wp-content/uploads/2010/02/sysbench_update_key_trx_commit.png)  
Results with innodb\_flush\_log\_at\_trx\_commit = 2 and binlogs  
[](https://www.percona.com/blog/wp-content/uploads/2010/02/sysbench_update_key_binlog.png)

Related

# Capture database traffic using the Performance Schema

https://www.percona.com/blog/2015/10/01/capture-database-traffic-using-performance-schema/

Capturing data is a critical part of performing a query analysis, or even just to have an idea of what’s going on inside the database.

There are several known ways to achieve this. For example:

* Enable the General Log
* Use the Slow Log with long\_query\_time = 0
* Capture packets that go to MySQL from the network stream using [TCPDUMP](https://www.percona.com/blog/2008/11/07/poor-mans-query-logging/)
* Use the pt-query-digest with the [–processlist](https://www.percona.com/doc/percona-toolkit/2.2/pt-query-digest.html#cmdoption-pt-query-digest--processlist) parameter

However, these methods can add significant overhead and might even have negative performance consequences, such as:

* Using SHOW PROCESSLIST requires a [Mutex](http://dev.mysql.com/doc/refman/5.6/en/show-processlist.html)
* Logging ALL queries [can hurt MySQL](https://www.percona.com/blog/2009/02/10/impact-of-logging-on-mysql%E2%80%99s-performance/), especially in CPU-bound cases
* Log rotation might lead to [bad things](https://www.percona.com/blog/2012/09/20/a-prototype-lower-impact-slow-query-log/)
* …and several other reasons that you can find [surfing the web](https://www.vividcortex.com/blog/2014/02/25/performance-schema-slow-query-log-tcp-sniffing/)

Now, sometimes you just need to sneak a peek at the traffic. Nothing fancy. In that case, probably the faster and easiest way to gather some traffic data is to use pt-query-digest with the –processlist. It doesn’t require any change in the server’s configuration nor critical handling of files. It doesn’t even require access to the server, just a user with the proper permissions to run “show full processlist”. But, and this is a significantly big “but,” you have to take into account that polling the SHOW PROCESSLIST command misses quite a number of queries and gives very poor timing resolution, among other things (like the processlist Mutex).

What’s the alternative? Use the **Performance Schema**. Specifically: The **events\_statements\_\*** and **threads** tables.

First, we have to make sure that we have the correspondent consumers enabled:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | mysql> select \* from setup\_consumers where name like 'events%statement%' and enabled = 'yes';  +--------------------------------+---------+  | NAME                           | ENABLED |  +--------------------------------+---------+  | events\_statements\_current      | YES     |  | events\_statements\_history      | YES     |  | events\_statements\_history\_long | YES     |  +--------------------------------+---------+  3 rows in set (0.00 sec) |

Additionally, for statistics to be collected for statements, it is not sufficient to enable only the final **statement/sql/\*** instruments used for individual statement types. The abstract statement**/abstract/\*** instruments must be enabled as well. This should not normally be an issue because all statement instruments are enabled by default.

If you can’t see the event\_statements\_\* consumers on your setup\_consumers tables, you’re probably running a MySQL version prior to ***5.6.3***. Before that version, the events\_statements\_\* tables didn’t exists. MySQL 5.6 might not be more widely used, as was already pointed out [in this same blog](https://www.percona.com/blog/2014/02/11/performance_schema-vs-slow-query-log/).

Before continuing, it’s important to note that the most important condition at the moment of capture data is that:

**The statements must have ended.**

If the statement is still being executed, it can’t be part of the collected traffic. For the ones out there that want to know what’s running inside MySQL, there’s already a detailed non-blocking processlist view to replace [INFORMATION\_SCHEMA. | SHOW FULL] PROCESSLIST available with[Sys Schema](https://github.com/MarkLeith/mysql-sys#processlist--xprocesslist) (that will come as default in MySQL 5.7).

Our options to capture data are: get it from one of the three available tables: events\_statements\_current, events\_statements\_history or events\_statements\_history\_long.

**First option:** use the [**events\_statements\_current**](http://dev.mysql.com/doc/refman/5.6/en/events-statements-current-table.html) table, which contains current statement events. Since we only want to get statements that have ended, the query will need to add the condition **END\_EVENT\_ID IS NOT NULL** to the query. This column is set to NULL when the event starts and updated to the thread current event number when the event ends, but when testing, there were too many missing queries. This is probably because between iterations, the associated threads were removed from the threads table or simply because the time between END\_EVENT\_ID being updated and the row being removed from the table is too short. This option is discarded.

***Second option:*** The [**events\_statements\_history**](http://dev.mysql.com/doc/refman/5.6/en/events-statements-history-table.html) table contains the most recent statement events per thread and since statement events are not added to the events\_statements\_history table until they have ended, using this table will do the trick without additional conditions in order to know if the event is still running or not. Also, as new events are added, older events are discarded if the table is full.

That means that this table size is fixed. You can change the table size by modifying the variable[performance\_schema\_events\_statements\_history\_size](http://dev.mysql.com/doc/refman/5.6/en/performance-schema-system-variables.html#sysvar_performance_schema_events_statements_history_size). In the server version I used (5.6.25-73.1-log Percona Server (GPL), Release 73.1, Revision 07b797f) the table size is, by default, defined as autosized (-1) and can have 10 rows per thread. For example: if you are running 5 threads, the table will have 50 rows.

Since it is a fixed size, chances are that some events might be lost between iterations.

***Third option:*** The [**events\_statements\_history\_long**](http://dev.mysql.com/doc/refman/5.6/en/events-stages-history-long-table.html) table, which is kind of an extended version of events\_statements\_history table. Depending on the MySQL version, by default it can hold up to 10000 rows or be autosized (also modifiable with the variable [performance\_schema\_events\_statements\_history\_long\_size](http://dev.mysql.com/doc/refman/5.6/en/performance-schema-system-variables.html#sysvar_performance_schema_events_stages_history_long_size))

One major -and not cool at all- drawback for this table is that **“When a thread ends, its rows are removed from the table”.**So it is not history-history data. It will go as far as the oldest thread, with the older event still alive.

The logical option to choose would be the third one: use the events\_statements\_history\_long table. I’ve created a small script (available [here](https://github.com/nethalo/mysql-tools/blob/master/get_history_long.sh)) to collect infinite iterations on all the events per thread between a range of event\_id’s. The idea of the range is to avoid capturing the same event more than once. Turns out that the execute a query against this table is pretty slow, something between 0.53 seconds and 1.96 seconds. It can behave in a quite invasive way.

Which leave us with the second option: The events\_statements\_history table.

Since the goal is to capture data in a slow log format manner,  additional information needs to be obtained from the [threads](http://dev.mysql.com/doc/refman/5.6/en/threads-table.html)table, which has a row for each server thread. The most important thing to remember: access to threads does not require a mutex and has minimal impact on server performance.

Combined, these two tables give us enough information to simulate a very comprehensive slow log format. We just need the proper query:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | SELECT  CONCAT\_WS(  '','# Time: ', date\_format(CURDATE(),'%y%m%d'),' ',TIME\_FORMAT(NOW(6),'%H:%i:%s.%f'),'\n'  ,'# User@Host: ',t.PROCESSLIST\_USER,'[',t.PROCESSLIST\_USER,'] @ ',PROCESSLIST\_HOST,' []  Id: ',t.PROCESSLIST\_ID,'\n'  ,'# Schema: ',CURRENT\_SCHEMA,'  Last\_errno: ',MYSQL\_ERRNO,'  ','\n'  ,'# Query\_time: ',ROUND(s.TIMER\_WAIT / 1000000000000, 6),' Lock\_time: ',ROUND(s.LOCK\_TIME / 1000000000000, 6),'  Rows\_sent: ',ROWS\_SENT,'  Rows\_examined: ',ROWS\_EXAMINED,'  Rows\_affected: ',ROWS\_AFFECTED,'\n'  ,'# Tmp\_tables: ',CREATED\_TMP\_TABLES,'  Tmp\_disk\_tables: ',CREATED\_TMP\_DISK\_TABLES,'  ','\n'  ,'# Full\_scan: ',IF(SELECT\_SCAN=0,'No','Yes'),'  Full\_join: ',IF(SELECT\_FULL\_JOIN=0,'No','Yes'),'  Tmp\_table: ',IF(CREATED\_TMP\_TABLES=0,'No','Yes'),'  Tmp\_table\_on\_disk: ',IF(CREATED\_TMP\_DISK\_TABLES=0,'No','Yes'),'\n'  , t.PROCESSLIST\_INFO,';')  FROM performance\_schema.events\_statements\_history s  JOIN performance\_schema.threads t using(thread\_id)  WHERE  t.TYPE = 'FOREGROUND'  AND t.PROCESSLIST\_INFO IS NOT NULL  AND t.PROCESSLIST\_ID != connection\_id()  ORDER BY t.PROCESSLIST\_TIME desc; |

The idea of this query is to get a Slow Log format as close as possible to the one that can be obtained by using all the options from the [log\_slow\_filter](https://www.percona.com/doc/percona-server/5.6/diagnostics/slow_extended.html) variable.

The other conditions are:

* **t.TYPE = ‘FOREGROUND’**: The threads table provides information about background threads, which we don’t intend to analyze. User connection threads are foreground threads.
* **t.PROCESSLIST\_INFO IS NOT NULL**: This field is NULL if the thread is not executing any statement.
* **t.PROCESSLIST\_ID != connection\_id()**: Ignore me (this query).

The output of the query will look like a proper Slow Log output:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # Time: 150928 18:13:59.364770  # User@Host: root[root] @ localhost []  Id: 58918  # Schema: test  Last\_errno: 0  # Query\_time: 0.000112 Lock\_time: 0.000031  Rows\_sent: 1  Rows\_examined: 1  Rows\_affected: 0  # Tmp\_tables: 0  Tmp\_disk\_tables: 0  # Full\_scan: No  Full\_join: No  Tmp\_table: No  Tmp\_table\_on\_disk: No  INSERT INTO sbtest1 (id, k, c, pad) VALUES (498331, 500002, '94319277193-32425777628-16873832222-63349719430-81491567472-95609279824-62816435936-35587466264-28928538387-05758919296'  , '21087155048-49626128242-69710162312-37985583633-69136889432'); |

And this file can be used with pt-query-digest to aggregate similar queries, just as it was a regular slow log output. I ran a small test which consists of:

* Generate traffic using sysbench. This is the sysbench command used:

Shell

|  |  |
| --- | --- |
| 1 | sysbench --test=/usr/share/doc/sysbench/tests/db/oltp.lua --mysql-host=localhost --mysql-port=3306 --mysql-user=root --mysql-password= --mysql-db=test --mysql-table-engine=innodb --oltp-test-mode=complex --oltp-read-only=off --oltp-reconnect=on --oltp-table-size=1000000 --max-requests=100000000 --num-threads=4 --report-interval=1 --report-checkpoints=10 --tx-rate=0 run |

* Capture the data using slow log + long\_query\_time = 0
* Capture data using pt-query-digest –processlist
* Capture data from Performance Schema
* Run pt-query-digest on the 3 files

The results were:

– Slow Log:

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # Profile  # Rank Query ID           Response time Calls  R/Call V/M   Item  # ==== ================== ============= ====== ====== ===== ==============  #    1 0x813031B8BBC3B329 47.7743 18.4%  15319 0.0031  0.01 COMMIT  #    2 0x737F39F04B198EF6 39.4276 15.2%  15320 0.0026  0.00 SELECT sbtest?  #    3 0x558CAEF5F387E929 37.8536 14.6% 153220 0.0002  0.00 SELECT sbtest?  #    4 0x84D1DEE77FA8D4C3 30.1610 11.6%  15321 0.0020  0.00 SELECT sbtest?  #    5 0x6EEB1BFDCCF4EBCD 24.4468  9.4%  15322 0.0016  0.00 SELECT sbtest?  #    6 0x3821AE1F716D5205 22.4813  8.7%  15322 0.0015  0.00 SELECT sbtest?  #    7 0x9270EE4497475EB8 18.9363  7.3%   3021 0.0063  0.00 SELECT performance\_schema.events\_statements\_history performance\_schema.threads  #    8 0xD30AD7E3079ABCE7 12.8770  5.0%  15320 0.0008  0.01 UPDATE sbtest?  #    9 0xE96B374065B13356  8.4475  3.3%  15319 0.0006  0.00 UPDATE sbtest?  #   10 0xEAB8A8A8BEEFF705  8.0984  3.1%  15319 0.0005  0.00 DELETE sbtest?  # MISC 0xMISC              8.5077  3.3%  42229 0.0002   0.0 <10 ITEMS> |

– pt-query-digest –processlist

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # Profile  # Rank Query ID           Response time Calls R/Call V/M   Item  # ==== ================== ============= ===== ====== ===== ===============  #    1 0x737F39F04B198EF6 53.4780 16.7%  3676 0.0145  0.20 SELECT sbtest?  #    2 0x813031B8BBC3B329 50.7843 15.9%  3577 0.0142  0.10 COMMIT  #    3 0x558CAEF5F387E929 50.7241 15.8%  4024 0.0126  0.08 SELECT sbtest?  #    4 0x84D1DEE77FA8D4C3 35.8314 11.2%  2753 0.0130  0.11 SELECT sbtest?  #    5 0x6EEB1BFDCCF4EBCD 32.3391 10.1%  2196 0.0147  0.21 SELECT sbtest?  #    6 0x3821AE1F716D5205 28.1566  8.8%  2013 0.0140  0.17 SELECT sbtest?  #    7 0x9270EE4497475EB8 22.1537  6.9%  1381 0.0160  0.22 SELECT performance\_schema.events\_statements\_history performance\_schema.threads  #    8 0xD30AD7E3079ABCE7 15.4540  4.8%  1303 0.0119  0.00 UPDATE sbtest?  #    9 0xE96B374065B13356 11.3250  3.5%   885 0.0128  0.09 UPDATE sbtest?  #   10 0xEAB8A8A8BEEFF705 10.2592  3.2%   792 0.0130  0.09 DELETE sbtest?  # MISC 0xMISC              9.7642  3.0%   821 0.0119   0.0 <3 ITEMS> |

– Performance Schema

Shell

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | # Profile  # Rank Query ID           Response time Calls R/Call V/M   Item  # ==== ================== ============= ===== ====== ===== ==============  #    1 0x813031B8BBC3B329 14.6698 24.8% 12380 0.0012  0.00 COMMIT  #    2 0x558CAEF5F387E929 12.0447 20.4% 10280 0.0012  0.00 SELECT sbtest?  #    3 0x737F39F04B198EF6  7.9803 13.5% 10280 0.0008  0.00 SELECT sbtest?  #    4 0x3821AE1F716D5205  4.6945  7.9%  5520 0.0009  0.00 SELECT sbtest?  #    5 0x84D1DEE77FA8D4C3  4.6906  7.9%  7350 0.0006  0.00 SELECT sbtest?  #    6 0x6EEB1BFDCCF4EBCD  4.1018  6.9%  6310 0.0007  0.00 SELECT sbtest?  #    7 0xD30AD7E3079ABCE7  3.7983  6.4%  3710 0.0010  0.00 UPDATE sbtest?  #    8 0xE96B374065B13356  2.3878  4.0%  2460 0.0010  0.00 UPDATE sbtest?  #    9 0xEAB8A8A8BEEFF705  2.2231  3.8%  2220 0.0010  0.00 DELETE sbtest?  # MISC 0xMISC              2.4961  4.2%  2460 0.0010   0.0 <7 ITEMS> |

The P\_S data is closer to the Slow Log one than the captured with regular SHOW FULL PROCESSLIST, but it is still far from being accurate. Remember that this is an alternative for a fast and easy way to capture traffic without too much trouble, so that’s a trade-off that you might have to accept.

**Summary:** Capture traffic always comes with a tradeoff, but if you’re willing to sacrifice accuracy it can be done with minimal impact on server performance, using the Performance Schema. Because P\_S is enabled by default since MySQL 5.6.6 you might already be living with the overhead (if using 5.6). If you are one of the lucky ones that have P\_S on production, don’t be afraid to use it. There’s a lot of data already in there.

# Aggregate INNODB\_METRICS Metrics · Issue #129 · prometheus/mysqld\_exporter

https://github.com/prometheus/mysqld\_exporter/issues/129

Some INNODB\_METRICS would better be aggregated.  
(which are disabled by default and can be enabled with innodb\_monitor\_enable variable)

Example: mysql\_info\_schema\_innodb\_metrics\_buffer\_buffer\_.\*

To make it easy, you can just aggregate according to subsystem as listed in<https://dev.mysql.com/doc/refman/5.6/en/innodb-information-schema-metrics-table.html>

I originally made the innodb\_metrics labeled by subsystem, but there were many cases where it didn't make sense.

So here's the aggregation that makes sense for subsystem=buffer buffer\_pool\_pages.\*.  
Total is the sum of data+mis+free. But dirty seems like a separate metric.

We have already mysql\_global\_status\_buffer\_pool\_pages with state data, dirty, misc, free an dmysql\_global\_status\_innodb\_buffer\_pool\_bytes\_data, mysql\_global\_status\_innodb\_buffer\_pool\_bytes\_dirty.  
So there is no benefit in this from innodb\_metrics at all.

# Monitoring MySQL performance metrics

https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/

his post is part 1 of a 3-part series about MySQL monitoring.[*Part 2*](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/)is about collecting metrics from MySQL, and[*Part 3*](https://www.datadoghq.com/blog/mysql-monitoring-with-datadog/)explains how to monitor MySQL using Datadog.

What is MySQL?

[MySQL](http://www.oracle.com/us/products/mysql/overview/index.html) is the most popular open source relational database server in the world. Owned by Oracle, MySQL is available in the freely downloadable Community Edition as well as in commercial editions with added features and support. Initially released in 1995, MySQL has since spawned high-profile forks for competing technologies such as MariaDB and Percona.

Key MySQL statistics

If your database is running slowly, or failing to serve queries for any reason, every part of your stack that depends on that database will suffer performance problems as well. In order to keep your database running smoothly, you can actively monitor metrics covering four areas of performance and resource utilization:

* [Query throughput](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#query-throughput)
* [Query execution performance](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#query-performance)
* [Connections](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#connections)
* [Buffer pool usage](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#buffer-pool-usage)

MySQL users can access hundreds of metrics from the database, so in this article we’ll focus on a handful of key metrics that will enable you to gain real-time insight into your database’s health and performance. In the [second part](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) of this series we’ll show you how to access and collect all of these metrics.

This article references metric terminology introduced in [our Monitoring 101 series](https://www.datadoghq.com/blog/monitoring-101-collecting-data/), which provides a framework for metric collection and alerting.

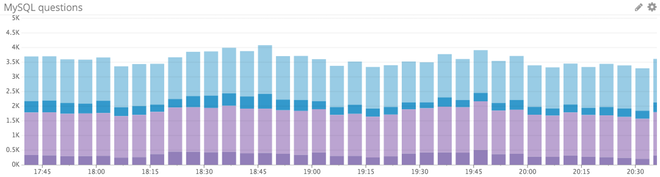
**Compatibility between versions and technologies**

Some of the monitoring strategies discussed in this series are specific to MySQL versions 5.6 and 5.7. Differences between those versions will be pointed out along the way.

Most of the metrics and monitoring strategies outlined here also apply to MySQL-compatible technologies such MariaDB and Percona Server, with some notable differences. For instance, some of the features in the MySQL Workbench, which is detailed in [Part 2](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) of this series, are not compatible with currently available versions of MariaDB.

Amazon RDS users should check out our specialized monitoring guides for[MySQL on RDS](https://www.datadoghq.com/blog/monitoring-rds-mysql-performance-metrics/) and for the MySQL-compatible [Amazon Aurora](https://www.datadoghq.com/blog/monitoring-amazon-aurora-performance-metrics/).

**Query throughput**

[](https://datadog-live.imgix.net/img/blog/monitoring-mysql-performance-metrics/mysql-questions.png?fit=max)

| **Name** | **Description** | [**Metric type**](https://www.datadoghq.com/blog/monitoring-101-collecting-data/) | [**Availability**](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) |
| --- | --- | --- | --- |
| Questions | Count of executed statements (sent by client) | Work: Throughput | Server status variable |
| Com\_select | SELECT statements | Work: Throughput | Server status variable |
| Writes | Inserts, updates, or deletes | Work: Throughput | Computed from server status variables |

Your primary concern in monitoring any system is making sure that its [work is being done](https://www.datadoghq.com/blog/monitoring-101-collecting-data/#work-metrics) effectively. A database’s work is running queries, so your first monitoring priority should be making sure that MySQL is executing queries as expected.

MySQL has an internal counter (a “server status variable”, in MySQL parlance) called Questions, which is incremented for all statements sent by client applications. The client-centric view provided by the Questions metric often makes it easier to interpret than the related Queries counter, which also counts statements executed as part of [stored programs](https://dev.mysql.com/doc/refman/5.6/en/stored-programs-views.html), as well as commands such as PREPARE and DEALLOCATE PREPARE run as part of server-side [prepared statements](https://dev.mysql.com/doc/refman/5.6/en/sql-syntax-prepared-statements.html).

To query a server status variable such as Questions or Com\_select:

SHOW GLOBAL STATUS LIKE "Questions";

+---------------+--------+

| Variable\_name | Value |

+---------------+--------+

| Questions | 254408 |

+---------------+--------+

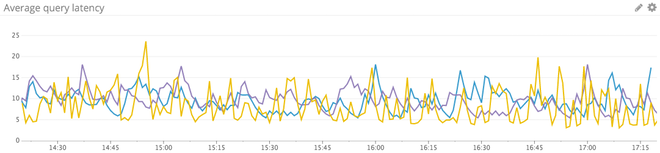
You can also monitor the breakdown of read and write commands to better understand your database’s workload and identify potential bottlenecks. Read queries are generally captured by the Com\_select metric. Writes increment one of three status variables, depending on the command:

Writes = Com\_insert + Com\_update + Com\_delete

Metric to alert on: Questions

The current rate of queries will naturally rise and fall, and as such it’s not always an actionable metric based on fixed thresholds. But it is worthwhile to alert on sudden changes in query volume—drastic drops in throughput, especially, can indicate a serious problem.

**Query performance**

[](https://datadog-live.imgix.net/img/blog/monitoring-mysql-performance-metrics/mysql-latency.png?fit=max)

| **Name** | **Description** | [**Metric type**](https://www.datadoghq.com/blog/monitoring-101-collecting-data/) | [**Availability**](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) |
| --- | --- | --- | --- |
| Query run time | Average run time, per schema | Work: Performance | Performance schema query |
| Query errors | Number of SQL statements that generated errors | Work: Error | Performance schema query |
| Slow\_queries | Number of queries exceeding configurablelong\_query\_time limit | Work: Performance | Server status variable |

MySQL users have a number of options for monitoring query latency, both by making use of MySQL’s built-in metrics and by querying the performance schema. Enabled by default [since MySQL 5.6.6](https://dev.mysql.com/doc/refman/5.6/en/performance-schema-quick-start.html), the tables of theperformance\_schema database within MySQL store low-level statistics about server events and query execution.

Performance schema statement digest

Many key metrics are contained in the performance schema’sevents\_statements\_summary\_by\_digest table, which captures information about latency, errors, and query volume for each normalized statement. A sample row from the table shows a statement that has been run twice and that took 325 milliseconds on average to execute (all timer measurements are in picoseconds):

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SCHEMA\_NAME: employees

DIGEST: 0c6318da9de53353a3a1bacea70b4fce

DIGEST\_TEXT: SELECT \* FROM `employees` WHERE `emp\_no` > ?

COUNT\_STAR: 2

SUM\_TIMER\_WAIT: 650358383000

MIN\_TIMER\_WAIT: 292045159000

AVG\_TIMER\_WAIT: 325179191000

MAX\_TIMER\_WAIT: 358313224000

SUM\_LOCK\_TIME: 520000000

SUM\_ERRORS: 0

SUM\_WARNINGS: 0

SUM\_ROWS\_AFFECTED: 0

SUM\_ROWS\_SENT: 520048

SUM\_ROWS\_EXAMINED: 520048

...

SUM\_NO\_INDEX\_USED: 0

SUM\_NO\_GOOD\_INDEX\_USED: 0

FIRST\_SEEN: 2016-03-24 14:25:32

LAST\_SEEN: 2016-03-24 14:25:55

[The digest table](https://dev.mysql.com/doc/refman/5.7/en/performance-schema-statement-digests.html) normalizes all the statements (as seen in the DIGEST\_TEXTfield above), ignoring data values and standardizing whitespace and capitalization, so that the following two queries would be considered the same:

select \* from employees where emp\_no >200;

SELECT \* FROM employees WHERE emp\_no > 80000;

To extract a per-schema average run time in microseconds, you can query the performance schema:

SELECT schema\_name

, SUM(count\_star) count

, ROUND( (SUM(sum\_timer\_wait) / SUM(count\_star))

/ 1000000) AS avg\_microsec

FROM performance\_schema.events\_statements\_summary\_by\_digest

WHERE schema\_name IS NOT NULL

GROUP BY schema\_name;

+--------------------+-------+--------------+

| schema\_name | count | avg\_microsec |

+--------------------+-------+--------------+

| employees | 223 | 171940 |

| performance\_schema | 37 | 20761 |

| sys | 4 | 748 |

+--------------------+-------+--------------+

Similarly, to count the total number of statements per schema that generated errors:

SELECT schema\_name

, SUM(sum\_errors) err\_count

FROM performance\_schema.events\_statements\_summary\_by\_digest

WHERE schema\_name IS NOT NULL

GROUP BY schema\_name;

+--------------------+-----------+

| schema\_name | err\_count |

+--------------------+-----------+

| employees | 8 |

| performance\_schema | 1 |

| sys | 3 |

+--------------------+-----------+

The sys schema

Querying the performance schema as shown above works great for programmatically retrieving metrics from the database. For ad hoc queries and investigation, however, it is usually easier to use MySQL’s [sys schema](https://github.com/mysql/mysql-sys). The sys schema provides an organized set of metrics in a more human-readable format, making the corresponding queries much simpler. For instance, to find the slowest statements (those in the 95th percentile by runtime):

SELECT \* FROM sys.statements\_with\_runtimes\_in\_95th\_percentile;

Or to see which normalized statements have generated errors:

SELECT \* FROM sys.statements\_with\_errors\_or\_warnings;

Many other useful examples are detailed in the sys schema [documentation](https://github.com/mysql/mysql-sys). The sys schema is included in MySQL starting with version 5.7.7, but MySQL 5.6 users can install it with just a few commands. See [Part 2](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) of this series for instructions.

Slow queries

In addition to the wealth of performance data available in the performance schema and sys schema, MySQL features a Slow\_queries counter, which increments every time a query’s execution time exceeds the number of seconds specified by the long\_query\_time parameter. The threshold is set to 10 seconds by default:

SHOW VARIABLES LIKE 'long\_query\_time';

+-----------------+-----------+

| Variable\_name | Value |

+-----------------+-----------+

| long\_query\_time | 10.000000 |

+-----------------+-----------+

The long\_query\_time parameter can be adjusted with one command. For example, to set the slow query threshold to five seconds:

SET GLOBAL long\_query\_time = 5;

(Note that you may have to close your session and reconnect to the database for the change to be applied at the session level.)

Investigating query performance issues

If your queries are executing more slowly than expected, it is often the case that a recently changed query is the culprit. If no query is determined to be unduly slow, the next things to evaluate are system-level metrics to look for constraints in core resources (CPU, disk I/O, memory, and network). CPU saturation and I/O bottlenecks are common culprits. You may also wish to check the Innodb\_row\_lock\_waits metric, which counts how often the InnoDB storage engine had to wait to acquire a lock on a particular row. InnoDB has been the default storage engine since MySQL version 5.5, and MySQL uses row-level locking for InnoDB tables.

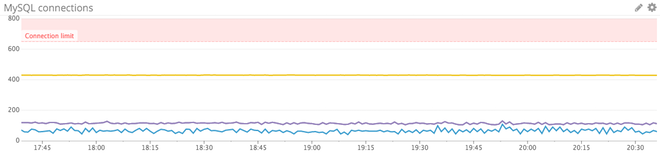
To increase the speed of read and write operations, many users will want to tune the size of the [buffer pool](https://dev.mysql.com/doc/refman/5.6/en/innodb-buffer-pool.html) used by InnoDB to cache table and index data. More on monitoring and resizing the buffer pool [below](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#buffer-pool-usage).

Metrics to alert on

* Query run time: Managing latency for key databases is critical. If the average run time for queries in a production database starts to climb, look for resource constraints on your database instances, possible contention for row or table locks, and changes in query patterns on the client side.
* Query errors: A sudden increase in query errors can indicate a problem with your client application or your database itself. You can use the sys schema to quickly explore which queries may be causing problems. For instance, to list the 10 normalized statements that have returned the most errors:
* SELECT \* FROM sys.statements\_with\_errors\_or\_warnings
* ORDER BY errors DESC
* LIMIT 10;

* Slow\_queries: How you define a slow query (and therefore how you configure the long\_query\_time parameter) depends on your use case. Whatever your definition of “slow,” you will likely want to investigate if the count of slow queries rises above baseline levels. To identify the actual queries executing slowly, you can [query the sys schema](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#the-sys-schema) or dive into MySQL’s optional slow query log, which is disabled by default. More information on enabling and accessing the slow query log is available [in the MySQL documentation](http://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html).

**Connections**

[](https://datadog-live.imgix.net/img/blog/monitoring-mysql-performance-metrics/mysql-conn.png?fit=max)

| **Name** | **Description** | [**Metric type**](https://www.datadoghq.com/blog/monitoring-101-collecting-data/) | [**Availability**](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) |
| --- | --- | --- | --- |
| Threads\_connected | Currently open connections | Resource: Utilization | Server status variable |
| Threads\_running | Currently running connections | Resource: Utilization | Server status variable |
| Connection\_errors\_ internal | Count of connections refused due to server error | Resource: Error | Server status variable |
| Aborted\_connects | Count of failed connection attempts to the server | Resource: Error | Server status variable |
| Connection\_errors\_ max\_connections | Count of connections refused due tomax\_connections limit | Resource: Error | Server status variable |

Checking and setting the connection limit

Monitoring your client connections is critical, because once you have exhausted your available connections, new client connections will be refused. The MySQL connection limit defaults to 151, but can be verified with a query:

SHOW VARIABLES LIKE 'max\_connections';

+-----------------+-------+

| Variable\_name | Value |

+-----------------+-------+

| max\_connections | 151 |

+-----------------+-------+

[MySQL’s documentation](https://dev.mysql.com/doc/refman/5.7/en/too-many-connections.html) suggests that robust servers should be able to handle connections in the high hundreds or thousands:

“Linux or Solaris should be able to support 500 to 1000 simultaneous connections routinely and as many as 10,000 connections if you have many gigabytes of RAM available and the workload from each is low or the response time target undemanding. Windows is limited to (open tables × 2 + open connections) < 2048 due to the Posix compatibility layer used on that platform.”

The connection limit can be adjusted on the fly:

SET GLOBAL max\_connections = 200;

That setting will return to the default when the server restarts, however. To permanently set the connection limit, add a line like this to your my.cnfconfiguration file (see [this post](http://stackoverflow.com/questions/2482234/how-to-know-mysql-my-cnf-location) for help in locating the config file):

max\_connections = 200

Monitoring connection utilization

MySQL exposes a Threads\_connected metric counting connection threads—[one thread per connection](https://dev.mysql.com/doc/refman/5.7/en/connection-threads.html). By monitoring this metric alongside your configured connection limit, you can ensure that you have enough capacity to handle new connections. MySQL also exposes the Threads\_runningmetric to isolate which of those threads are actively processing queries at any given time, as opposed to connections that are open but are currently idle.

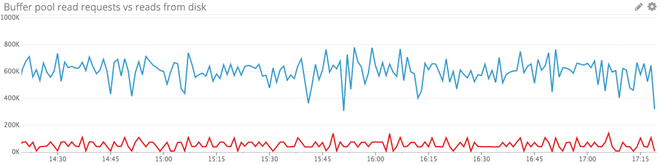
If your server does reach the max\_connections limit, it will start to refuse connections. In that event, the metric Connection\_errors\_max\_connectionswill be incremented, as will the Aborted\_connects metric tracking all failed connection attempts.

MySQL exposes a variety of other metrics on connection errors, which can help you investigate connection problems. The metricConnection\_errors\_internal is a good one to watch, because it is incremented only when the error comes from the server itself. Internal errors can reflect an out-of-memory condition or the server’s inability to start a new thread.

Metrics to alert on

* Threads\_connected: If a client attempts to connect to MySQL when all available connections are in use, MySQL will return a “Too many connections” error and increment Connection\_errors\_max\_connections. To prevent this scenario, you should monitor the number of open connections and make sure that it remains safely below the configuredmax\_connections limit.
* Aborted\_connects: If this counter is increasing, your clients are trying and failing to connect to the database. Investigate the source of the problem with fine-grained connection metrics such asConnection\_errors\_max\_connections and Connection\_errors\_internal.

**Buffer pool usage**

[](https://datadog-live.imgix.net/img/blog/monitoring-mysql-performance-metrics/mysql-buffer.png?fit=max)

| **Name** | **Description** | [**Metric type**](https://www.datadoghq.com/blog/monitoring-101-collecting-data/) | [**Availability**](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) |
| --- | --- | --- | --- |
| Innodb\_buffer\_pool\_pages\_total | Total number of pages in the buffer pool | Resource: Utilization | Server status variable |
| Buffer pool utilization | Ratio of used to total pages in the buffer pool | Resource: Utilization | Computed from server status variables |
| Innodb\_buffer\_pool\_read\_requests | Requests made to the buffer pool | Resource: Utilization | Server status variable |
| Innodb\_buffer\_pool\_reads | Requests the buffer pool could not fulfill | Resource: Saturation | Server status variable |

MySQL’s default storage engine, InnoDB, uses an area of memory called the buffer pool to cache data for tables and indexes. Buffer pool metrics are[resource metrics as opposed to work metrics](https://www.datadoghq.com/blog/monitoring-101-collecting-data/), and as such are primarily useful for investigating (rather than detecting) performance issues. If database performance starts to slide while disk I/O is rising, expanding the buffer pool can often provide benefits.

Sizing the buffer pool

The buffer pool defaults to a relatively small 128 mebibytes, but MySQL advises that you can [increase it to as much as 80 percent](http://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_buffer_pool_size) of physical memory on a dedicated database server. MySQL also adds a few notes of caution, however, as InnoDB’s memory overhead can increase the memory footprint by about 10 percent beyond the allotted buffer pool size. And if you run out of physical memory, your system will resort to paging and performance will suffer significantly.

The buffer pool also can be divided into separate regions, known as instances. Using multiple instances [can improve concurrency](https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_buffer_pool_instances) for buffer pools in the multi-GiB range.

Buffer-pool resizing operations are performed in chunks, and the size of the buffer pool must be set to a multiple of the chunk size times the number of instances:

innodb\_buffer\_pool\_size = N \* innodb\_buffer\_pool\_chunk\_size

\* innodb\_buffer\_pool\_instances

The chunk size defaults to 128 MiB but is [configurable](https://dev.mysql.com/doc/refman/5.7/en/innodb-parameters.html#sysvar_innodb_buffer_pool_chunk_size) as of MySQL 5.7.5. The value of both parameters can be checked as follows:

SHOW GLOBAL VARIABLES LIKE "innodb\_buffer\_pool\_chunk\_size";

SHOW GLOBAL VARIABLES LIKE "innodb\_buffer\_pool\_instances";

If the innodb\_buffer\_pool\_chunk\_size query returns no results, the parameter is not tunable in your version of MySQL and can be assumed to be 128 MiB.

To set the buffer pool size and number of instances at server startup:

$ mysqld --innodb\_buffer\_pool\_size=8G --innodb\_buffer\_pool\_instances=16

As of MySQL 5.7.5, you can also [resize the buffer pool on-the-fly](https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool-resize.html#innodb-buffer-pool-online-resize) via a SETcommand specifying the desired size in bytes. For instance, with two buffer pool instances, you could set each to 4 GiB size by setting the total size to 8 GiB:

SET GLOBAL innodb\_buffer\_pool\_size=8589934592;

Key InnoDB buffer pool metrics

MySQL exposes a handful of metrics on the buffer pool and its utilization. Some of the most useful are the metrics tracking the total size of the buffer pool, how much is in use, and how effectively the buffer pool is serving reads.

The metrics Innodb\_buffer\_pool\_read\_requests andInnodb\_buffer\_pool\_reads are key to understanding buffer pool utilization.Innodb\_buffer\_pool\_read\_requests tracks the the number of logical read requests, whereas Innodb\_buffer\_pool\_reads tracks the number of requests that the buffer pool could not satisfy and therefore had to be read from disk. Given that reading from memory is generally orders of magnitude faster than reading from disk, performance will suffer if Innodb\_buffer\_pool\_readsstarts to climb.

Buffer pool utilization is a useful metric to check before you consider resizing the buffer pool. The utilization metric is not available out of the box but can be easily calculated as follows:

(Innodb\_buffer\_pool\_pages\_total - Innodb\_buffer\_pool\_pages\_free) /

Innodb\_buffer\_pool\_pages\_total

If your database is serving a large number of reads from disk, but the buffer pool is far from full, it may be that your cache has recently been cleared and is still warming up. If your buffer pool does not fill up but is effectively serving reads, your working set of data likely fits comfortably in memory.

High buffer pool utilization, on the other hand, is not necessarily a bad thing in isolation, as old or unused data is automatically aged out of the cache using an [LRU](https://en.wikipedia.org/wiki/Cache_algorithms#LRU) policy. But if the buffer pool is not effectively serving your read workload, it may be time to scale up your cache.

Converting buffer pool metrics to bytes

Most buffer pool metrics are reported as a count of memory pages, but these metrics can be converted to bytes, which makes it easier to connect these metrics with the actual size of your buffer pool. For instance, to find the total size of buffer pool in bytes using the [server status variable](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/#collecting-server-status-variables) tracking total pages in the buffer pool:

Innodb\_buffer\_pool\_pages\_total \* innodb\_page\_size

The InnoDB page size is adjustable but defaults to 16 KiB, or 16,384 bytes. Its current value can be checked with a SHOW VARIABLES query:

SHOW VARIABLES LIKE "innodb\_page\_size";

Conclusion

In this post we have explored a handful of the most important metrics you should monitor to keep tabs on MySQL activity and performance. If you are building out your MySQL monitoring, capturing the metrics outlined below will put you on the path toward understanding your database’s usage patterns and potential constraints. They will also help you to identify when it is necessary to scale out or move your database instances to more powerful hosts in order to maintain good application performance.

* [Query throughput](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#query-throughput)
* [Query latency and errors](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#query-performance)
* [Client connections and errors](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#connections)
* [Buffer pool utilization](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#buffer-pool-usage)

[Part 2](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) of this series provides instructions for collecting and monitoring all the metrics you need from MySQL.

Acknowledgments

Many thanks to [Dave Stokes](https://www.linkedin.com/in/davidmstokes) of Oracle and [Ewen Fortune](https://www.linkedin.com/in/efortune) of [VividCortex](https://www.vividcortex.com/) for providing valuable feedback on this article prior to publication.

[*Want to write articles like this one? Our team is hiring!*](http://dtdg.co/2lkgplP)

# Collecting MySQL statistics and metrics

https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/

This post is part 2 of a 3-part MySQL monitoring series.[*Part 1*](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/)explores key performance statistics in MySQL, and[*Part 3*](https://www.datadoghq.com/blog/mysql-monitoring-with-datadog/)explains how to set up MySQL monitoring in Datadog.

As covered in [Part 1](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/) of this series, MySQL users can access a wealth of performance metrics and statistics via two types of database queries:

* Querying internal server status variables for high-level summary metrics
* Querying the performance schema and sys schema for a more granular view

In this article we’ll walk through both approaches to metric collection. We’ll also discuss how to view those metrics in the free MySQL Workbench GUI or in a full-featured monitoring system.

Collecting server status variables

Out of the box, recent versions of MySQL come with about 350 metrics, known as [server status variables](http://dev.mysql.com/doc/refman/5.7/en/server-status-variables.html). Each of them can be queried at the session or global level.

Each of the server status variables highlighted in [Part 1](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/) of this series can be retrieved using a [SHOW STATUS](http://dev.mysql.com/doc/refman/5.7/en/show-status.html) statement:

SHOW GLOBAL STATUS LIKE 'Questions';

+---------------+-------+

| Variable\_name | Value |

+---------------+-------+

| Questions | 89537 |

+---------------+-------+

These statements also support pattern matching to query a family of related metrics simultaneously. To check metrics on connection errors, for instance:

SHOW GLOBAL STATUS LIKE '%Connection\_errors%';

+-----------------------------------+-------+

| Variable\_name | Value |

+-----------------------------------+-------+

| Connection\_errors\_accept | 0 |

| Connection\_errors\_internal | 0 |

| Connection\_errors\_max\_connections | 15 |

| Connection\_errors\_peer\_address | 0 |

| Connection\_errors\_select | 0 |

| Connection\_errors\_tcpwrap | 0 |

+-----------------------------------+-------+

Server status variables are easy to collect on an ad hoc basis, as shown above, but they can also be [queried programmatically](https://github.com/DataDog/integrations-core/blob/master/mysql/check.py) and passed into an external monitoring system.

Querying the performance schema and sys schema

**Enabling the performance schema**

The performance schema stores performance metrics about individual SQL statements, rather than the summary statistics of the server status variables. The performance schema comprises database tables that can be queried like any other.

The performance schema is enabled by default [since MySQL 5.6.6](https://dev.mysql.com/doc/refman/5.6/en/performance-schema-quick-start.html). You can verify that it is enabled by running the following command from a shell prompt:

mysqld --verbose --help | grep "^performance-schema\s"

In the output, you should see a line like this:

performance-schema TRUE

To enable the performance schema, add the following line under the[mysqld] heading in your my.conf [file](http://stackoverflow.com/questions/2482234/how-to-know-mysql-my-cnf-location):

performance\_schema

The configuration change will be picked up after server restart.

**Performance schema queries**

Once the performance schema is enabled, it will collect metrics on all the statements executed by the server. Many of those metrics are summarized in the events\_statements\_summary\_by\_digest table, available in MySQL 5.6 and later.

Metrics on query volume, latency, errors, time spent waiting for locks, index usage, and more are available for each normalized SQL statement executed. (Normalization here means stripping data values from the SQL statement and standardizing whitespace.)

You can query the performance schema using ordinary SELECT statements. For instance, to find the statement with the longest average run time:

SELECT digest\_text

, count\_star

, avg\_timer\_wait

FROM events\_statements\_summary\_by\_digest

ORDER BY avg\_timer\_wait DESC

LIMIT 1;

+---------------------------------------+------------+----------------+

| digest\_text | count\_star | avg\_timer\_wait |

+---------------------------------------+------------+----------------+

| SELECT \* FROM `employees` . `titles` | 2 | 468854512000 |

+---------------------------------------+------------+----------------+

Here we see that one query, which has been executed twice, takes nearly half a second to complete on average. (All timer measurements are reported in picoseconds.)

A full sample row from this table can be found in [Part 1](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#performance-schema-statement-digest) of this series, along with example queries to extract metrics on query run time and errors for each MySQL schema.

**Installing the sys schema**

Though you can query the performance schema directly, it is generally easier to use the [sys schema](https://github.com/mysql/mysql-sys). The sys schema contains easily interpretable tables for inspecting your performance data.

The sys schema comes installed with MySQL starting with version 5.7.7, but users of earlier versions can install it in seconds. For instance, to install the sys schema on MySQL 5.6:

git clone https://github.com/mysql/mysql-sys.git

cd mysql-sys/

mysql -u root -p < ./sys\_56.sql

For MySQL 5.7, simply replace the file name in the final command with./sys\_57.sql.

If you prefer to work in a GUI, you can also install the sys schema [using the MySQL Workbench tool](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/#using-the-mysql-workbench-gui).

**Sys schema queries**

The tables of the sys schema distill the information of the performance schema into a more user-friendly, readable form. Its ease of use makes the sys schema ideal for ad hoc investigations or performance tuning, as opposed to programmatic access.

The [sys schema documentation](https://github.com/mysql/mysql-sys) provides detailed information on the various tables and functions, along with a number of useful examples. For instance, to summarize all the statements executed on each host, along with their associated latencies:

SELECT \* FROM host\_summary\_by\_statement\_type;

+------+----------------------+--------+---------------+-------------+--------------+-----------+---------------+---------------+------------+

| host | statement | total | total\_latency | max\_latency | lock\_latency | rows\_sent | rows\_examined | rows\_affected | full\_scans |

+------+----------------------+--------+---------------+-------------+--------------+-----------+---------------+---------------+------------+

| hal | create\_view | 2063 | 00:05:04.20 | 463.58 ms | 1.42 s | 0 | 0 | 0 | 0 |

| hal | select | 174 | 40.87 s | 28.83 s | 858.13 ms | 5212 | 157022 | 0 | 82 |

| hal | stmt | 6645 | 15.31 s | 491.78 ms | 0 ps | 0 | 0 | 7951 | 0 |

| hal | call\_procedure | 17 | 4.78 s | 1.02 s | 37.94 ms | 0 | 0 | 19 | 0 |

| hal | create\_table | 19 | 3.04 s | 431.71 ms | 0 ps | 0 | 0 | 0 | 0 |

...

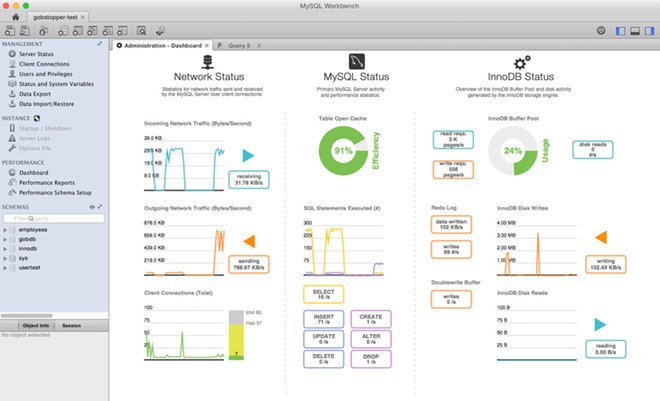
+------+----------------------+--------+---------------+-------------+--------------+-----------+---------------+---------------+------------+

Note that in the table above, all timer measurements have been converted from raw picosecond counts to human-readable units.

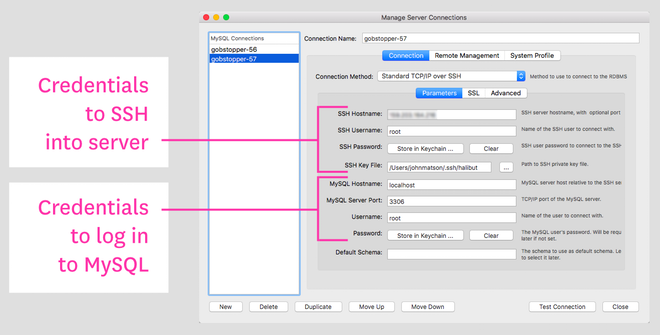
See [Part 1](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#the-sys-schema) of this series for examples of how to use the sys schema to surface slow-running queries or to find the source of errors.

Using the MySQL Workbench GUI

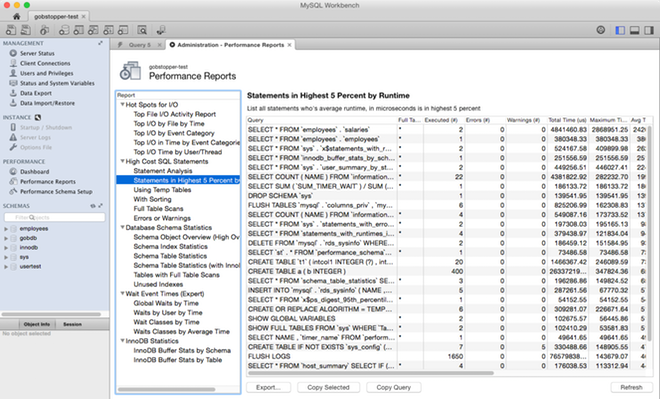
[MySQL Workbench](http://dev.mysql.com/downloads/workbench/) is a free application with a GUI for managing and monitoring a MySQL instance. MySQL Workbench provides a high-level performance dashboard, as well as an easy-to-use interface for browsing the performance metrics available from the [sys schema](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/#sys-schema-queries).

[](https://datadog-live.imgix.net/img/blog/collecting-mysql-statistics-and-metrics/workbench-ui.png?fit=max)

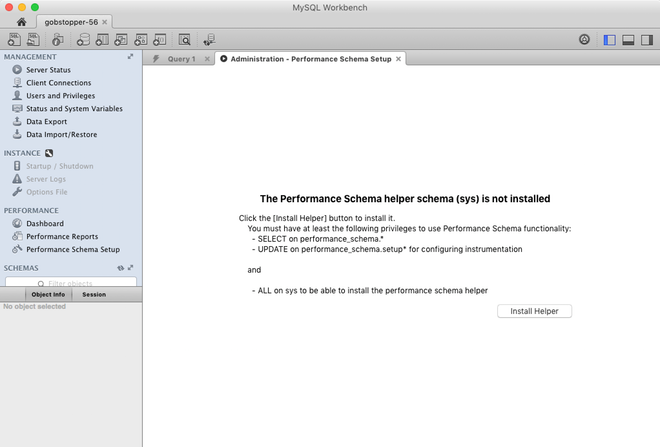
If you are running MySQL on a remote server, you can connect MySQL Workbench to your database instance via SSH tunneling:

[](https://datadog-live.imgix.net/img/blog/collecting-mysql-statistics-and-metrics/workbench-mysql-ssh.png?fit=max)

You can then view recent metrics on the performance dashboard or click through the statistics available from the sys schema:

[](https://datadog-live.imgix.net/img/blog/collecting-mysql-statistics-and-metrics/95th-percentile.png?fit=max)

If you are using a version before MySQL 5.7.7 and you have not [installed the sys schema](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/#installing-the-sys-schema), MySQL Workbench will prompt you to install it from the GUI.

[](https://datadog-live.imgix.net/img/blog/collecting-mysql-statistics-and-metrics/sys-schema-56.png?fit=max)

Using a full-featured monitoring tool

All of the metric collection methods listed above are useful for ad hoc performance checks, investigation, and tuning. Some of these metrics can also be accessed programmatically—server status variables, in particular, can easily be queried and parsed at regular intervals. But to implement ongoing monitoring of a production MySQL database, you will likely want to use a full-featured monitoring tool that integrates with MySQL.

Mature monitoring platforms allow you to visualize and alert on real-time metrics, as well as view your metrics’ evolution over time. They also allow you to correlate your metrics across systems, so you can quickly determine if errors from your application originated in MySQL, or if increased query latency is due to system-level resource constraints. [Part 3](https://www.datadoghq.com/blog/mysql-monitoring-with-datadog/) of this series shows you how to set up comprehensive MySQL monitoring with Datadog.

Wrap-up

In this post we have shown you how to collect summary or low-level metrics from MySQL. Whether you prefer writing SQL queries or using a GUI, the approaches described above should help you gain immediate insight into the usage patterns and performance of your MySQL databases.

In [the next and final part](https://www.datadoghq.com/blog/mysql-monitoring-with-datadog/) of this series, we’ll show you how you can quickly integrate MySQL with Datadog for continuous, comprehensive monitoring.

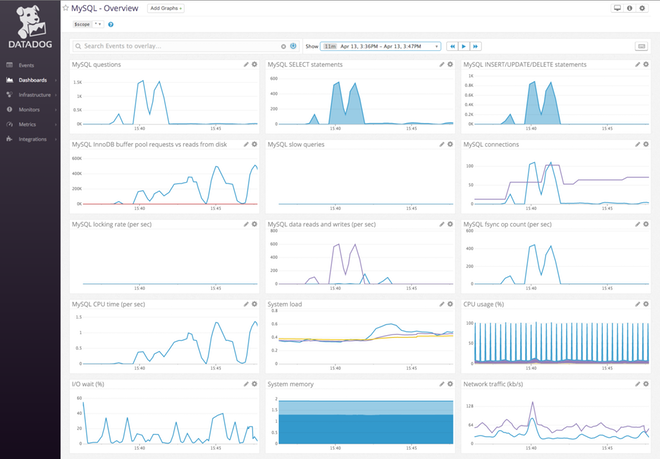
[*Want to write articles like this one? Our team is hiring!*](http://dtdg.co/2lkgplP)

# MySQL monitoring with Datadog

https://www.datadoghq.com/blog/mysql-monitoring-with-datadog/

This is the final post in a 3-part series about MySQL monitoring.[*Part 1*](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/)explores the key metrics available from MySQL, and[*Part 2*](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/)explains how to collect those metrics.

If you’ve already read [our post](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/) on collecting MySQL metrics, you’ve seen that you have several options for ad hoc performance checks. For a more comprehensive view of your database’s health and performance, however, you need a monitoring system that continually collects MySQL statistics and metrics, that lets you identify both recent and long-term performance trends, and that can help you identify and investigate issues when they arise. This post will show you how to set up comprehensive MySQL monitoring by installing the Datadog Agent on your database servers.

[](https://datadog-live.imgix.net/img/blog/mysql-monitoring-with-datadog/mysql-dash-dd.png?fit=max)

Integrate Datadog with MySQL

As explained in [Part 1](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/), MySQL exposes hundreds of valuable metrics and statistics about query execution and database performance. To collect those metrics on an ongoing basis, the Datadog Agent connects to MySQL at regular intervals, queries for the latest values, and reports them to Datadog for graphing and alerting.

**Installing the Datadog Agent**

Installing the Agent on your MySQL server is easy: it usually requires just a single command, and the Agent can collect basic metrics even if [the MySQL performance schema](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/#querying-the-performance-schema-and-sys-schema) is not enabled and the sys schema is not installed. Installation instructions for a variety of operating systems and platforms are available [here](https://app.datadoghq.com/account/settings#agent).

**Configure the Agent to collect MySQL metrics**

Once the Agent is installed, you need to grant it access to read metrics from your database. In short, this process has four steps:

The [MySQL configuration tile](https://app.datadoghq.com/account/settings#integrations/mysql) in the Datadog app has the full instructions, including the exact SQL commands you need to run to create the datadoguser and apply the appropriate permissions.

**Configure collection of additional MySQL metrics**

Out of the box, Datadog collects more than 60 standard metrics from modern versions of MySQL. Definitions and measurement units for most of those standard metrics can be found [here](http://docs.datadoghq.com/integrations/mysql/#metrics).

Starting with Datadog Agent version 5.7, many additional metrics are available by enabling specialized checks in the conf.d/mysql.yaml file (see[the configuration template](https://github.com/DataDog/integrations-core/blob/master/mysql/conf.yaml.example) for context):

# options:

# replication: false

# galera\_cluster: false

# extra\_status\_metrics: true

# extra\_innodb\_metrics: true

# extra\_performance\_metrics: true

# schema\_size\_metrics: false

# disable\_innodb\_metrics: false

To collect average statistics on query latency, as described in [Part 1](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/#query-performance) of this series, you will need to enable the extraperformancemetrics option and ensure that [the performance schema is enabled](https://www.datadoghq.com/blog/collecting-mysql-statistics-and-metrics/#querying-the-performance-schema-and-sys-schema). The Agent’s datadog user in MySQL will also need the [additional permissions](https://app.datadoghq.com/account/settings#integrations/mysql) detailed in the MySQL configuration instructions in the Datadog app.

Note that the extraperformancemetrics and schemasizemetrics options trigger heavier queries against your database, so you may be subject to performance impacts if you enable those options on servers with a large number of schemas or tables. Therefore you may wish to test out these options on a limited basis before deploying them to production.

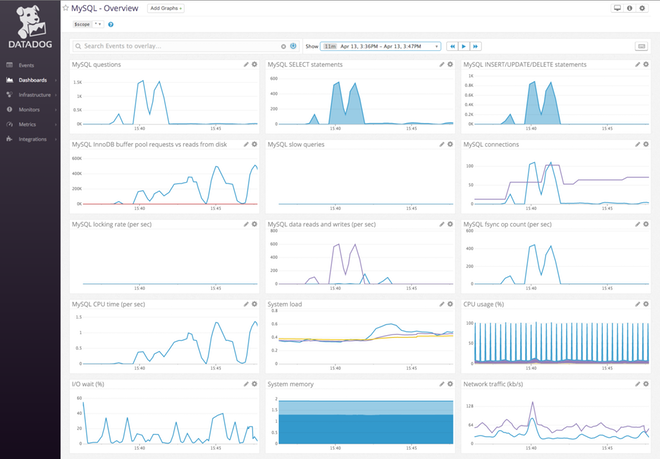
Other options include:

* extra\_status\_metrics to expand the set of server status variables reported to Datadog
* extra\_innodb\_metrics to collect more than 80 additional metrics specific to the InnoDB storage engine
* replication to collect basic metrics (such as replica lag) on MySQL replicas

To override default behavior for any of these optional checks, simply uncomment the relevant lines of the configuration file (along with theoptions: line) and restart the agent.

The specific metrics associated with each option are detailed in [the source code](https://github.com/DataDog/integrations-core/blob/master/mysql/check.py) for the MySQL Agent check.

View your comprehensive MySQL dashboard

[](https://datadog-live.imgix.net/img/blog/mysql-monitoring-with-datadog/mysql-dash-dd.png?fit=max)

Once you have integrated Datadog with MySQL, a comprehensive dashboard called “[MySQL - Overview](https://app.datadoghq.com/dash/integration/mysql)” will appear in your list of [integration dashboards](https://app.datadoghq.com/dash/list). The dashboard gathers key MySQL metrics highlighted in [Part 1](https://www.datadoghq.com/blog/monitoring-mysql-performance-metrics/) of this series, along with server resource metrics, such as CPU and I/O wait, which are invaluable for investigating performance issues.

**Customize your dashboard**

The Datadog Agent can also collect metrics from the rest of your infrastructure so that you can correlate your entire system’s performance with metrics from MySQL. The Agent collects metrics from [Docker](https://www.datadoghq.com/blog/the-docker-monitoring-problem/), [NGINX](https://www.datadoghq.com/blog/how-to-monitor-nginx/),[Redis](https://www.datadoghq.com/blog/how-to-monitor-redis-performance-metrics/), and 150+ other applications and services. You can also easily instrument your own application code to [report custom metrics to Datadog using StatsD](https://www.datadoghq.com/blog/statsd/).

To add more metrics from MySQL or related systems to your MySQL dashboard, simply clone [the template dash](https://app.datadoghq.com/dash/integration/mysql) by clicking on the gear in the upper right.

Conclusion

In this post we’ve walked you through integrating MySQL with Datadog so you can access all your database metrics in one place, whether standard metrics from MySQL, more detailed metrics from the InnoDB storage engine, or automatically computed metrics on query latency.

Monitoring MySQL with Datadog gives you critical visibility into what’s happening with your database and the applications that depend on it. You can easily create automated alerts on any metric, with triggers tailored precisely to your infrastructure and your usage patterns.

If you don’t yet have a Datadog account, you can sign up for a [free trial](https://www.datadoghq.com/blog/mysql-monitoring-with-datadog/) to start monitoring all your servers, applications, and services today.

[*Want to write articles like this one? Our team is hiring!*](http://dtdg.co/2lkgplP)

# 其他

## 企业级服务50强榜单出炉：阿里云、腾讯云、钉钉位居前三

\_TechWeb http://www.techweb.com.cn/internet/2016-07-04/2356205.shtml

其中阿里云、腾讯云以及钉钉位居帮办的前三位。

据了解，该榜单囊括了云服务、大数据、移动办公、开发者服务、企业级智能、人力资源、营销、网络安全、办公租赁等众多细分行业，使用独创的估值模型揭示行业格局及创新趋势。

50家上榜企业总估值约1,979亿元，平均估值39.6亿元，中位数为18亿元，入围门槛6.5亿元。

云服务是企业服务50强的最大组成部分，合计估值1,181.5亿元，占上榜企业总估值的59.7%，阿里云以680亿元估值成为该领域巨无霸，占上榜企业总估值约1/3，其2016财年营收达30亿元，在全球公有云市场仅次于亚马逊AWS、微软Azure。

大数据公司9家入选企业中TalkingData体量最大，在39亿元水平，其余公司估值都在20亿元以下。

企业级智能即人工智能的企业级应用，包括开源的PaaS开发平台及具体的智能解决方案，受AlphaGo等事件影响，市场的估值水平普遍较高。其中，云知声以24.7亿元的估值成为估值最高的人工智能独角兽企业，在总体排名中位列第13名。紧随其后的人工智能公司，还有Face++ 22.7亿元、商汤科技22亿元、思必驰13.5亿元、格林深瞳13亿元、地平线机器人9.8亿元。

网络安全则与云服务共同作为基础设施服务移动办公、大数据等应用。（明宇）



企业级服务50强榜单出炉：阿里云、腾讯云、钉钉位居前三

更多阿里云相关： [阿里云官网](http://click.aliyun.com/m/1857/) [服务器全网底价30元](http://click.aliyun.com/m/17143/) [30余款产品免费半年](http://click.aliyun.com/m/17145/) [9.9元学生专享](http://click.aliyun.com/m/17146/)