MySQL（含 Percona Server/MariaDB 等） 性能优化

作者：黄金华

2017/03/09

1 MySQL 简介

1.1 简介

MySQL 是一款支持多用户、多线程的开源关系型数据库。它既可以嵌入到应用程序中，也可以支持数据仓库、内部索引、在线事务处理系统（OLTP）等各种应用类型。

同时，除了社区版 MySQL 外，还在其基础上衍生了多个开源版本，不仅对其性能也多其功能进行多方位的扩展，例如 Percona Server、MariaDB、AliSQL 等。但总体上，绝大部分衍生版本与社区版 MySQL 相兼容，因此其性能优化的基本流程也是一致的。

1.2 与其他关系型数据库对比

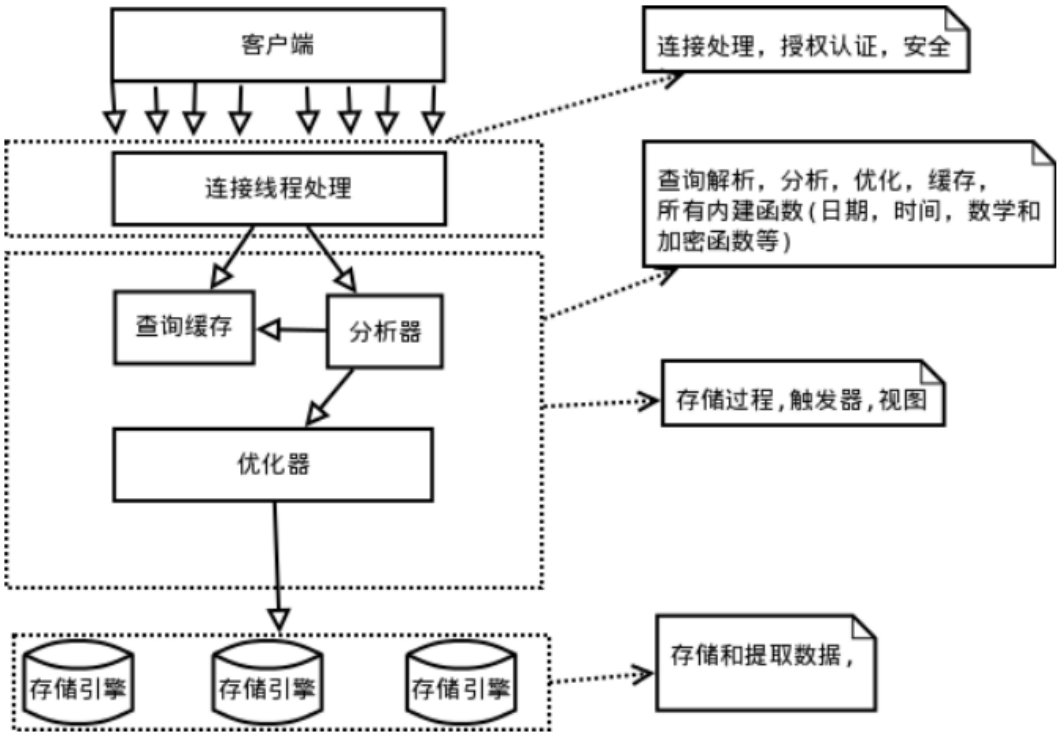
就关系数据库而言，目前开源且广泛使用的主要有 MySQL（及其衍生版本）和 PostgreSQL。因此，主要对这两种关系型数据库进行对比。

	MySQL(InnoDB)	PostgreSQL
连接处理	创建新或复用已有线程处理新的连接请求。通常新线程创建开销小，同时可创建数目多。	创建新的进程处理连接请求。通常创建新进程开销大，同时可创建进程数目受系统资源影响较大，因而进程数目相对较少。为此，常借助第三方中间件例如 PgBouncer 或 PgPool-Ⅱ等共享连接，避免因新连接导致频繁的进程创建和销毁
缓存处理	主进程分配大块缓存（Buffer Pool）供该进程所有线程共享。 为避免双重缓存，可支持 O_DIRECT 方式写入	主进程与所有工作进程通过共享内存方式共享缓存（Shared Buffer）。因不直接支持 O_DIRECT，因此会同时存在 PostgreSQL 内部缓存和文件系统本身的双重缓存
数据类型	支持基本数据类型，以及常见的索引等，但不支持 JSON 等复杂类型	除支持基本类型外， 还是支持复杂的数据类型，例如 JSON 等，同时支持扩展数量类型、函数以及索引等；
数据存储与索引	其内部索引通常是二级索引。通常是二级索引建立索引 key 到主键的索引，而一级索引是建立主键到物理存储位置的索引。优点是，因物理存储位置发生变化时只需要更新一级索引，而缺点是增加了 I/O 请求。 同时，当数据修改更新时，是在原数据上直接更新，而如果原数据还正在被引用则会被复制到新的地方。	其内部通常是一级索引，也就是说直接从索引 Key 到磁盘位置建立索引。同时，对数据的更新修改是重新分配磁盘空间，而非在原来位置直接更新。其缺点是，当小量数据更新时，需要重建相关索引。 同时，除支持基本 B-Tree 索引类型外，还支持

	常见索引是 B-Tree 索引以及 Hash 索引。	Hash/GiST/SP-GiST/GIT/BRIN 等索引类型
与应用的关系	MySQL 等是 Thin 数据库，也就是说数据库只是作为基本的数据存储，而复杂的业务逻辑由应用本身来处理	PostgreSQL 是数据 Thick 数据库，不仅可以作为基本的数据存储，而可以处理复杂的业务逻辑

2 MySQL 内部机制

2.1 架构总览



关系型数据库的架构基本上大同小异，其内部主要由下列几个模块组成：

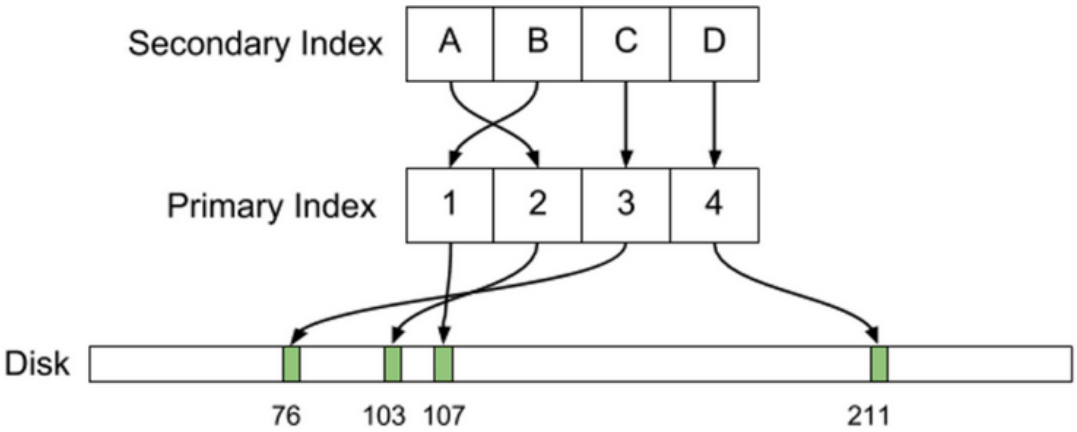
- SQL 解析器（或分析器）：主要负责查询语句的解析，优化以及改写等（某些实现可能把改写组成另外一个单独模块）；
- SQL 优化器：每个 SQL 语句可能有多种执行方式（或者执行计划），而 SQL 优化器负责根据相关算法选择最优计划（值得注意的是目前 SQL 优化器依然基于一个基本假设就是随机 I/O 请求远高于顺序 I/O 请求，而且 I/O 请求是衡量执行计划是否最优的主要参考因素，但随着 SSD 存储设备的逐渐普及，这种假设可能并不成立）
- SQL 执行：主要负责执行最优计划。
- 存储引擎：负责最后数据的存储和提取。就 MySQL 而言，广泛使用的事务型存储引擎是 InnoDB，而非事务存储引擎的典型代表是 MyISAM。鉴于目前广泛使用的是

InnoDB，因此本文默认所有优化都是基于 InnoDB 的。

2.2 索引机制

一般情况下，相对于全文检索而言，建立索引可减少 I/O 请求并提升性能。但如滥用索引，反而会降低性能。

默认索引是 B-Tree 索引，其基本结构如下图。与其他数据库（如 PostgreSQL 等）索引实现方式不同的是，其并不是直接建立从索引键（例如 A,B,C 等）到物理存储位置（例如 Disk 76,103 等）映射，而是建立二级索引既索引键到主键的索引和主键到物理位置的索引。其优点是当物理存储位置发生变更的时候只需要更新主键索引即可，而无需更新二级索引，而其缺点是新增的一层索引可能增加更多 I/O 请求。



同时还支持哈希索引和全文索引。相比 B-Tree 索引，没有冲突的 Hash 索引因 O(1)时间复杂度从而提供更好性能，但 Hash 索引无法提供类似 B-Tree 索引具有的排序、范围检索等功能。此外，MySQL 还提供配置选项，可支持自动建立 Hash 索引。也就说，MySQL 内部可根据索引键的使用情况自动建立（和删除）HASH 索引，而无需用户接入。

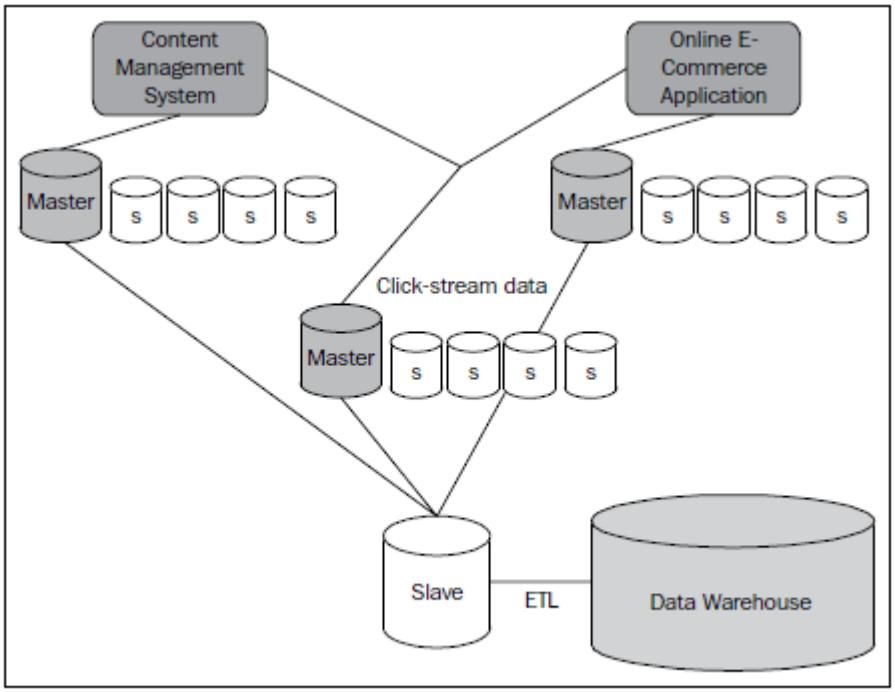
2.3 复制、可扩展性以及高可用性

2.3.1 主/从复制

仅有 Master 节点可以写，而其他多个从节点可以读。同时，主/从可以采用基于语句或行进行同步。目前这些同步是都异步的，也就是当主节点突然宕机时，从节点可能有部分数据并不是完全跟主节点一致的。顺便提一句，PostgreSQL 可支持同步的主从同步机制。

这个模式还可以有多个变体，例如：

- 主/从模式下主-主复制：典型情况下是两台机器互为主、从。也就是当一台宕机时，另一台服务器自动升级为主服务器。
  此外，还可以选择多台从服务器。至于选择哪台从服务器为主服务器，既可以指定，也可以借助 Zookeeper 等协议来自动选举一台作为主服务器。
- 多个主节点的主/从模式，例如：



2.3.2 主/主模式

所有节点都是 Master 节点，都支持读写操作，但是如何处理多个节点的写冲突依旧是一大

难题。目前 MySQL 并不能很好的支持这种模式。

2.3.3 分区/分片

上述复制模式只能扩展读能力，而无法扩展写能力。目前来说，扩展写能力的有效方法是采用“分区”或“分片”。

分区（Partitioning）可分为水平分区（Horizontal Partitioning）和垂直分区（Vertical Partitioning），其主要区别如下：

- 水平分区（如分片）：根据数据表的行来切分，也就是说每个节点存储数据表的部分行，但存储了该行的所有列的值；
- 垂直分区：根据列来切分，每个节点存储一列或多个列的值，因此一行的分布在多个节点；

同时，常见的分区方法如下：

- RANGE 分区：根据数据行的值范围将表分布到多个节点；
- LIST 分区：跟 RANGE 非常类似，其主要区别是不再是根据连续值来区分，而是根据值的列表来区分；
- HASH 区分：根据数据表行的自定义 HASH 函数返回的值来分区表；
- KEY 分区：跟 HASH 非常类似，其主要区别是使用内嵌的 HASH 函数，同时该 HASH 函数返回的值为整数；
- 组合分区：结合上述几种方法；

3 性能优化

3.1 基本方法论

对于性能优化的基本方法论，以及基本工具可参阅：https://zhuanlan.zhihu.com/p/25013051。

3.2 基准测试

对于新部署的机器，其基准测试包括两部分：

- 基础资源（CPU/Memory/Disk 等）基准测试
  主要是衡量基础资源是否配置恰当，是否得到有效利用，是否能承载后续业务能力；

- MySQL 基准测试
  对数据库而言，其性能的两个主要指标是：时延和吞吐量，并以时延或吞吐量作为优化目标。常见的基准测试是 sysbench 或 tpcc 测试。

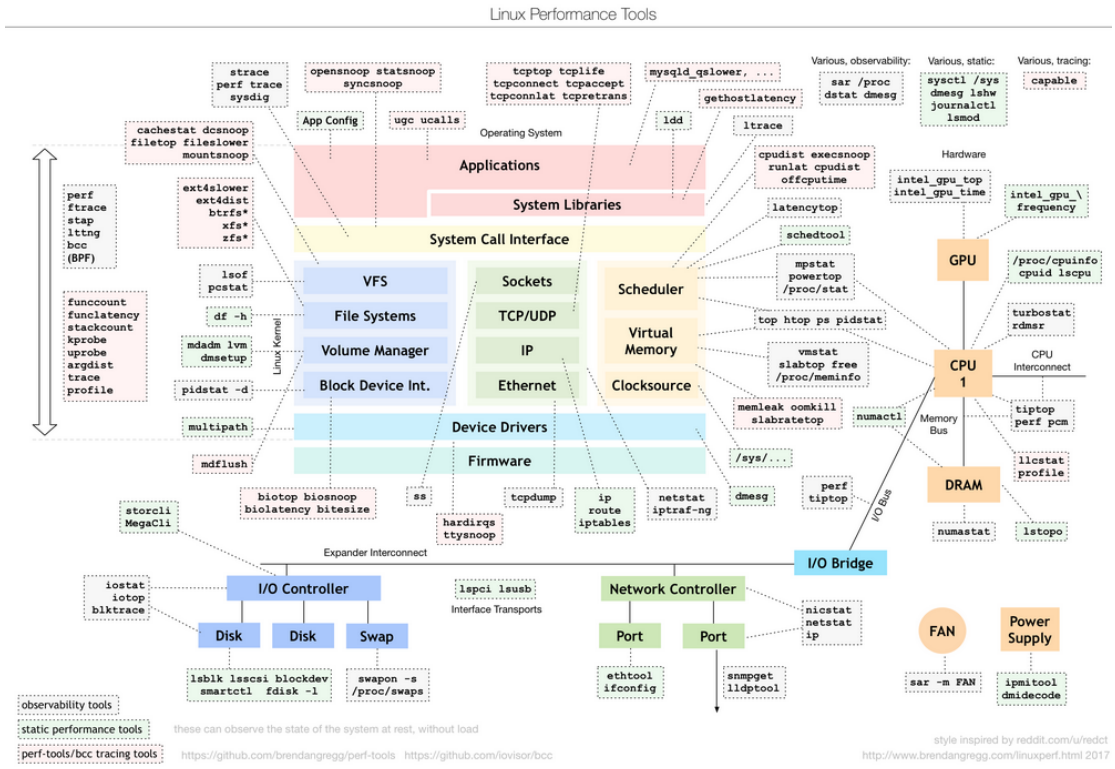
  关于基准测试用例配置、部署等，可参考自动化部署脚本（https://github.com/sjtuhjh/applications/tree/master/apps），在此不再赘述。

3.3 性能分析

3.3.1 基础性能分析

对 CPU/Memory/Disk/Networking/Kernel 等监控，可采用 perf/ftrace/bcc/top/dstat 等多种工具，既可以支持静动跟踪，也支持动态跟踪。

具体工具可参考 http://github.com/sjtuhjh/perftools，在此不再赘述。



3.3.2 MySQL 应用性能分析

对于 MySQL 性能，大致有下述几种方法:

- 总体性能分析：
  - 慢日志分析：设置慢日志相关配置使其统计所有查询，然后使用 Percona 提供的 pt-query-digest 分析慢日志，既可以对系统总体性能有一个大概了解，又可以找出真正慢的查询；
  - Tcpdump 分析：使用 Tcpdump 抓取 MySQL 服务器对应的端口（默认 3306）网络报文，然后再使用 pt-query-digest 来分析，则同样可对系统的整体性能有个大概了解；
  - Performance Schema(5.6 以后的版本)：既可以对所有用户进行统计分析，也可以针对用户或线程以及特定事件进行性能统计分析；
- 单个 Query 性能分析：
  - Explain ：先找出最慢的 Query，然后使用 Explain 分析其优化计划，从而有针对性的进行优化；
  - SHOW Profile: 使用 Show profile 可以统计分析单个 Query 执行情况以及相关性能数据，不过该命令将逐将被 Performance Schema 所取代；

还有其他各种工具以及分析方法，在此不再赘述。具体可参阅 http://github.com/sjtuhjh/perftools。

3.4 基础优化

3.4.1 BIOS/CPU/Memory

在对应用优化之前，要确保 BIOS/CPU/Memory 配置恰当，否则后续优化往往事倍功半。为此，需要借助一些测试工具来验证硬件功能是否工作正常（或者最优）。值得注意的是，如开始测试的时候无法根据数据来确定是否工作正常，可将数据与历史数据或者对应的 X86 测试数据进行对比。常见的测试工具如下：

- **CPU**：
  使用 SpecInt2006 或 Sysbench 测试 CPU 的基准性能，同时关注 L1/L2/L3 大小以及命中率；

- **内存**:
  - lmbench/stream 测试，重点关注有无跨 Die 时内存带宽；

- **存储设备**
  - RAID1：将两块硬盘构成 RAID 磁盘阵列，其容量仅相当于一块磁盘，但另一块磁盘总是保持完整的数据备份。一般支持“热交换”。
  - RAID10：首先建立两个独立的 RAID1，然后将两个独立的 RAID1 再组成一个 RAID0

  建议配备 RAND10，同时支持 BBU(Battery Backup Unit)和 WRITE-BACK 功能的存储设备。这样既可以有良好的读写性能，又能防止意外情况下（例如突然断电）存储内容丢失。

- **BIOS 电源管理**：将电源管理设置成“性能”模式以便最大发挥 CPU 性能；

常见的基准测试工具也可参考 https://github.com/sjtuhjh/applications/tree/master/apps。

3.4.2 OS/Kernel 优化

3.4.2.1文件系统

通常建议采用 EXT4 或 XFS 文件系统。

- Barriers I/O (Barrier 之前的所有 I/O 请求必须在 Barrier 之前结束，并持久化到非易失性介质中。而 Barrier 之后的 I/O 必须在 Barrier 之后执行。如果本身存储设备自带电池足以预防在突然掉电时其缓存内容依然可以被正确持久化，则可以关闭提高性能)。一般文件系统挂载时提供选项可以关闭 barriers,例如：mount –o nobarrier /dev/sda /u01/data
- /dev/sda2 / ext4 rw,noatime,nodiratime,data=writeback,discard 0 1
  采用 data=writeback 意味着仅有元数据保存，而没有任何形式的日志记录。这种方式对 InnoDB 使用，因为 InnoDB 本身有自己的日志。

- 打开 noatime，减少不必要的 I/O 操作，例如：mount –o nobarrier,noatime …
- 调整 vm.overcommit (将其设置为 2，从而当内存不够时返回内存请求失败，尽量避免 OOM(Out of Memory killer))
- 写缓存优化（需要根据需要调整参数，当脏页刷写太频繁，会导致过多 I/O 请求，而刷写频率过低，又可能导致所占内存过多，而在不得不刷写脏页时导致应用性能波动）：
  - vm.dirty_background_ratio: 当脏页数量达到一定比例时，触发内核线程异步将脏页写入磁盘；
  - vm.dirty_ratio: 当脏页数量达到一定比例时，系统不得不将脏页写入缓存。此过程可能导致应用的 I/O 被阻塞。
- I/O 调度
  - 对于普通磁盘，建议使用 deadline 调度方式，而固态硬盘各种调度方式影响并不大；
- Read-ahead 调整：
  - blockdev –getra /dev/sda 查看 sda
  - blockdev –setra <sectors> /dev/sda （设置预读缓冲大小）
- 对于部分临时文件系统，如果本身不需要存储（或者即使丢失也无关紧要），可通过 tmpfs 将其常驻内存，以便减少 I/O 操作，例如：
  tmpfs /tmp tmpfs defaults,noatime,mode=1777 0 0
  tmpfs/var/lock tmpfs defaults,noatime,mode=1777 0 0

3.4.2.2存储设备设置优化

3.4.2.2.1 分区对齐

分区对齐的目的是让逻辑块与物理块对齐从而减少 I/O 操作。通常可以借助 parted 命令来自动对齐，例如：

device=/dev/sdb

parted –s –a optimal $device mklable gpt

parted –s –a optimal $device mkpart primary ext4 0% 100%

其中 optimal 表示要尽量分区对齐从而达到最好性能。

3.4.2.2.2 SSD 优化

SSD 支持 TRIM 功能，打开该功能可以防止未来读写性能恶化。通过 Btrfs, EXT4, JFS 和 XFS 文件系统都支持该功能。可通过下列命令进行配置：

- 普通磁盘: 在 mount 的时候添加 discard 选项，例如：
  /dev/sda / ext4 rw,discard 0 1
- LVM (Logical Volumes)： 在/etc/lvm/lvm.conf 中添加 issue_discards=1；

3.4.2.2.3 I/O 调度算法

一般建议将/sys/block/<设备名>/queue/scheduler 设置为 deadline 或 noop。尤其是对普通机械硬盘，不要将其设置为 cfq 算法。

3.4.2.3SWAP

尽量禁止使用 SWAP，从而避免性能波动。可在/etc/sysctl.conf 中添加 vm.swappiness=0 并通过 sysctl –p 使其生效。

3.4.2.4HugePage

可以通过“cat /proc/meminfo | grep PageTables”可以查看页表大小。当页表过大时，建议使用 HugePage，从而可以提升 TLB 命中率，进而提升系统性能。可在/etc/sysctl.conf 中增加 vm.nr_hugepages。

3.4.2.5资源绑定

根据需要，可通过 cgroup、taskset 或 numactl 将任务绑定到特定 CPU 或 NUMA memory 节点上。同时如果网络中断频繁，需要将 irqbalance 关闭，并将中断绑定到特定 CPU 上。

3.5 MySQL 配置优化

对一些性能影响显著的配置进行说明优化。

- innodb_buffer_pool_size ：具体将其值设置多少，要取决该台机器总共有多少可供 MySQL 使用。一般将内存总量 – OS 本身使用的使用（含文件系统需要缓存的） – 其他应用使用的 – MySQL 自身需要的（例如为每个查询单独分配的，以及 query 缓存等）得到的值，并除以 105%，得到一个合理值作为一个初始配置值；
- innodb_buffer_pool_instances: 可将缓存池分成多段，从而访问冲突；
- innodb_old_blocks_time: 这个变量指定一个页面从 LRU 链表的“年轻”部分转移到“年老”部分必须等待的毫秒数，一般可将其设置为 1000ms；
- 线程缓存
  - thread_cache_size ：可以观察 thread_connected/thread_created 变化来判别 thread_cache_size 是否设置合理（一般可将其值设置足够大以便处理正常业务波动）
- 表缓存：
  - table_open_cache/table_cache_size: 应该被设置足够大，以避免总是需要重新打开和重新解析表的定义，可以通过 open_tables 的值来检查该变量是否设置合理；
- I/O 配置：

- InnoDB 事务日志
  - innodb_log_file_size/innodb_log_files_in_group: 要确定理想的日志文件大小，必须权衡正常数据变更的开销和奔溃恢复需要的时间。如果日志太小，则必须做更多的检查点，导致更多的日志写。另一方面，如果日志文件太大，则在崩溃恢复时需要更多的恢复时间；
  - innodb_flush_log_at_trx_commit:将其设置为 1 则每次事务提交时都要刷新日志到持久化存储。虽然这是最安全的，但会影响性能。因此一般建议将其设置为 2，则每次提交时把日志缓存写到日志文件，但并不刷新，而是每秒中刷新一次（因此如果突然断电就有可能有 1s 的数据被丢失的风险）。另外，还可以设置为 0。配置 0 与配置 2 的最大区别是，如果 MySQL 进程突然 crash，则配置 0 存在事务丢失的风险，而事务 2 则不存在，因为在每次提交的时候已经将数据从 MySQL 缓存拷贝到文件系统缓存。因此，一般建议使用配置 2
  - innodb_flush_method: 一般建议使用 O_DIRECT（依旧使用 fsync 来刷新磁盘，但是通知操作系统不需要缓存数据，也不需要缓存，避免双重缓存）。另外，O_DSYNC/O_SYNC: 与 O_DIRECT 不同的是并没有禁用操作系统的缓存。使用 O_SYNC 的话，则每个 write()/pwrite()都会在函数完成之前将数据同步到磁盘，并且在函数完成之前是阻塞的。而不使用 O_SYNC 的 fsync()允许写操作累计在缓存，然后一次性刷新所有数据。
- InnoDB 表空间
  - innodb_data_file_path: 可以通过类似 innodb_data_file_path = /disk1/ibda1:1G;/disk2/idbdata2:1G…方式将文件存放在不同的驱动器的不同目录。当然这种方式可能收益有限，而使用 RAID 控制器则收益更明显。
  - 打开 innodb_file_per_table
- 二进制日志
  - sync_binlog: 控制怎么刷新二进制日志到磁盘。默认为 0，表示 MySQL 本身并不控制刷新，而是交给文件系统来决定什么时候该刷新。一般将其设置为 1000，代表 1000 次二进制日志写操作后进行一次刷新。
  - expire_logs_days: 主要是避免因日志过多导致磁盘写满；
- innodb_io_capacity: 需要根据存储设备能力来设置（对于 PCI-E SSD 可能要设置为上万）；
- innodb_read_io_threads/innodb_write_io_threads: 可以简单将该值设置为提供 I/O 能力的磁盘数量
- 对于 SSD 来说，可禁用预读，禁用双写缓存（如果 SSD 支持一次写入 16K）和限制变更缓存（change buffer）的大小；

- InnoDB 并发配置：innodb_thread_concurrency 可设置为 CPU 数量*磁盘数量*2（或者配置线程池）
- tmp_table_size/max_heap_table_size: 控制 Memory 引擎的内存临时表能使用多大的内存；
- max_connections: 允许的最大连接数，主要是保证服务器不会因连接数激增而不堪重负；
- open_files_limit: 可将其设置为较大值，避免文件打开失败错误；
- 安全和稳定配置：
  - expire_logs_days: 开启该选项让服务器在指定天数后清理旧的二进制日志；

- max_allowed_packet: 可控制服务器发送太大的包，同时也会控制多大的包可以被接收；
- max_connect_errors: 如果因意外导致短时间内不断尝试连接并失败，则后续无法连接。一般无需更改。
- skip_name_resolve: 避免 DNS 查询导致连接延迟；

3.6 MySQL 索引优化

索引优化策略：

- 独立的列：索引列不能是表达式的一部分，也不能是函数的参数；
- 前缀索引和索引选择性：对于很长的字符列，可索引开始的部分字符，既可以节约索引空间，也可以提高索引效率；但通常会降低索引的选择性。
- 多列索引：要避免为每个列单独建立索引，因为大多数情况下并不会提升性能；
- 选择合适的索引列顺序：正确的顺序依赖于使用该索引的查询，并同时需要考虑如何更好的满足排序和分组的需要；
- 覆盖索引：在建立索引的时候要考虑整个查询，让索引包含（或覆盖）所有需要查询的字段的值，从而提升性能；

3.7 MySQL 查询优化

对于慢查询，通常可以通过下列两个步骤来分析：

- 确认应用程序是否在检索大量超过需要的数据；
- 确认 MySQL 服务器层是否在分析大量超过需要的数据行；

一般可以从两方面来进行优化：

- 重构查询：
  - 将一个复杂查询分解多个简单查询；
  - 分解关联查询
- 优化查询对应的执行计划：
  - 查询优化器提供的执行计划并不总是最优的，例如统计数据没有及时更新、针对部分语句优化器并不能很好的处理等；
  - 例如优化 COUNT()、关联查询、GROUPT BY/DISTINCT、LIMIT 等；

4 附录

[1] www.mysql.com
[2] http://momjian.us/main/writings/psql
[3] High Performance MySQL
[4] High Performance MariaDB