

《MySQL 5.5 新特性详解及参数优化》

蓝皮书

目录

《MySQL 5.5 新特性详解及参数优化》蓝皮书.....	1
第 1 部分 MySQL 5.5 新特性篇.....	3
第 1 章 MySQL 5.5 介绍.....	3
1.1 性能上显著的改变.....	4
1.1.1 MySQL5.5 默认存储引擎改为 InnoDB plugin (1.1.X)，而不是传统的 MyISAM 引擎。.....	4
1.1.2 充分利用 CPU 多核的处理能力。.....	8
1.1.3 提高刷新脏页数量和合并插入数量，改善磁盘 IO 处理能力。.....	9
1.1.4 增加了自适应刷新脏页功能。.....	10
1.1.5 让 Innodb_Buffer_Pool 缓冲池热数据存活更久。.....	11
1.1.6 加快了 InnoDB 的数据恢复时间。.....	13
1.1.7 INNODB 同时支持多个 BufferPool 实例.....	18
1.1.8 可关闭自适应哈希索引.....	20
1.1.9 可选用内存分配程序使用控制.....	21
1.1.10 提高了默认 innodb 线程并发数.....	26
1.1.11 预读算法的变化.....	27
1.1.12 首次在 Linux 上实现了异步 I/O.....	28
1.1.13 恢复组提交.....	29
1.1.14 innodb 使用多个回滚段提升性能.....	31

1.1.15 改善清除程序进度.....	31
1.1.16 添加了删除缓冲和清除缓冲.....	32
1.1.17 控制自旋锁 Spin Lock 轮训间隔.....	33
1.1.18 快速创建、删除、更改索引.....	34
1.1.19 InnoDB 支持创建压缩数据页.....	36
1.1.20 可动态关闭 InnoDB 更新元数据统计功能.....	44
1.2 安全性、稳定性显著的改变.....	45
1.2.1 复制功能(Replication)加强.....	45
1.2.2 中继日志 relay-log 自我修复。.....	46
1.2.3 开启 InnoDB 严格检查模式.....	46
1.3 动态更改系统配置参数.....	47
1.3.1 支持动态更改独立表空间.....	47
1.3.2 支持动态更改 InnoDB 锁超时时间.....	47
1.4 Innodb 新参数汇总.....	48
1.5 同步复制新参数汇总.....	58
1.6 SQL 语句写法的改变.....	64
1.6.1 delete 表连接语法改变.....	64

贺春暘

2012.7

第 1 部分 MySQL 5.5 新特性篇

第 1 章 MySQL 5.5 介绍

MySQL 是一个中、小型关系型数据库管理系统，由瑞典 MySQL AB 公司开发，目前属于 [Oracle](#) 公司。MySQL 在过去由于性能高、成本低、可靠性好，已经成为最流行的开源数据库，因此被广泛地应用在 Internet 上的中小型网站中。随着 MySQL 的不断成熟，它也逐渐用于更多大规模网站和应用，比如维基百科、Google 和 Facebook 等网站。非常流行的开源软件组合 LAMP 中的“M”指的就是 MySQL。

MySQL 5.5 提供了一组专用功能集，在当今现代化、多功能处理硬件和软件以及中间件构架涌现的环境中，极大地提高了 MySQL 的性能、可扩展性、可用性。

MySQL 5.5 融合了 MySQL 数据库和 InnoDB 存储引擎的优点，能够提供高性能的数据管理解决方案，包括：

- InnoDB 作为默认的数据库存储引擎
- 提升了 Windows 系统下的系统性能和可扩展性
- 改善性能和可扩展性，全面利用各平台现代多核构架的计算能力
- 提高实用性
- 提高易管理性和效率
- 提高可用性
- 改善检测与诊断性能

本书介绍性的讲解了 MySQL 5.5 和 InnoDB 的一些增强性能，这些增强性能极大地提高了系统和 MySQL 的性能。下面，我们将详细介绍每一个关键的增强性能及其实现过程。

为了不误导读者，保证全文的准确性，通过翻译 MySQL 5.5 官方手册《14.4. New Features of InnoDB 1.1》来向大家一一介绍，下面是我列举的一些较为重要的改变，也许有疏漏的地方，那么请大家访问 <http://dev.mysql.com/doc/refman/5.5/en/innodb-5-5.html>，参考原文英文文档。

1.1 性能上显著的改变

1.1.1 MySQL5.5 默认存储引擎改为 InnoDB plugin (1.1.X)，而不是传统的 MyISAM 引擎。

在 MySQL5.1.X 前版本中，默认的存储引擎是 MyISAM 引擎，每个 MyISAM 在磁盘上存储成三个文件。第一个文件的名称以表的名称开始，扩展名指出文件类型。.frm 文件存储表定义。数据文件的扩展名为.MYD (MYData)。索引文件的扩展名是.MYI (MYIndex)。

它的特点是表级锁、不支持事务和全文索引，适合一些 CMS 内容管理系统作为后台数据库用，在大并发、重负荷生产系统上，表锁的特性就显得力不从心。

并且在系统出现宕机、mysqld 进程崩溃时，MyISAM 引擎表很容易受到损坏，你不得不用外部命令 myisamchk 去修复它。

从 MySQL5.5.X 开始，默认的存储引擎变为 InnoDB Plugin 引擎。

InnoDB 给 MySQL 提供了具有提交、回滚和崩溃恢复能力的事务安全(ACID 兼容) 存储引擎。InnoDB 锁定在行级并且也在 SELECT 语句提供一个 Oracle 风格一致的非锁定读。这些特色增加了多用户部署和性能。没有在 InnoDB 中扩大锁定的需要，因为在 InnoDB 中行级锁定适合非常小的空间。InnoDB 也支持 FOREIGN KEY 强制。在 SQL 查询中，你可以自由地将 InnoDB 类型的表与其它 MySQL 的表的类型混合起来，甚至在同一个查询中也可以混合。

InnoDB 是为处理巨大数据量时的最大性能设计。它的 CPU 效率可能是任何其它基于磁盘的关系数据库引擎所不能匹敌的。

InnoDB 存储引擎被完全与 MySQL 服务器整合，InnoDB 存储引擎为在主内存中缓存数据和索引而维持它自己的缓冲池。InnoDB 存储它的表&索引在一个表空间中，表空间可以包含数个文件（或原始磁盘分区）。这与 MyISAM 表不同，比如在 MyISAM 表中每个表被存在分离的文件中。InnoDB 表可以是任何尺寸，即使在文件尺寸被限制为 2GB 的操作系统上。

InnoDB 被用来在众多需要高性能的大型数据库站点上产生。著名的 Internet 新闻站点 Slashdot.org 运行在 InnoDB 上。Mytrix, Inc.在 InnoDB 上存储超过 1TB 的数据，还有一些其它站点在 InnoDB 上处理平均每秒 800 次插入/更新的负荷。

随着 InnoDB 存储引擎的崛起，更多公司为其提供了 patch 补丁，使其性能发挥至最大化，其中包括 Google 公司、Percona 公司、Sun Microsystems 等公司为开源事业所做的贡献，在 MySQL5.1.X 版本里，你可以自己选择是否加载打过补丁后的 InnoDB Plugin（版本 1.0.X），InnoDB Plugin 较之 Built-in 版本新增了

很多特性，在后面会一一介绍，这里不再叙述。我之前用的 MySQL5.1.43 二进制版，InnoDB Plugin 已经包含在其/usr/local/mysql/lib/plugin/目录下了。

```
total 17808
drwxr-xr-x 2 mysql mysql    4096 Jun  1  2010 .
drwxr-xr-x 3 mysql mysql    4096 Jun  1  2010 ..
-rw-r--r-- 1 mysql mysql 11686678 Jun  1  2010 ha_innodb_plugin.a
-rwxr-xr-x 1 mysql mysql   1054 Jun  1  2010 ha_innodb_plugin.la
lrwxrwxrwx 1 mysql mysql    25 Jun 10  2010 ha_innodb_plugin.so -> ha_innodb_plugin.so.0.0.0
lrwxrwxrwx 1 mysql mysql    25 Jun 10  2010 ha_innodb_plugin.so.0 -> ha_innodb_plugin.so.0.0.0
-rwxr-xr-x 1 mysql mysql 6390758 Jun  1  2010 ha_innodb_plugin.so.0.0.0
```

修改 my.cnf 配置文件，添加如下：

```
ignore_builtin_innodb
plugin-load=innodb=ha_innodb_plugin.so;
innodb_trx=ha_innodb_plugin.so;
innodb_locks=ha_innodb_plugin.so;
innodb_lock_waits=ha_innodb_plugin.so;
innodb_cmp=ha_innodb_plugin.so;
innodb_cmp_reset=ha_innodb_plugin.so;
innodb_cmpmem=ha_innodb_plugin.so;
innodb_cmpmem_reset=ha_innodb_plugin.so
```

重启 MySQL 服务后，登陆 mysql>select @@innodb_version; 验证是否成功。

而在 MySQL5.5.X 版本里，你可以省去上面那么多操作步骤，直接修改 my.cnf 配置文件，添加如下即可：

```
innodb_file_per_table=1
innodb_file_format=barracuda
innodb_strict_mode=1
```

```
mysql> select @@version;
```

```
+-----+
| @@version |
+-----+
| 5.5.19    |
+-----+
1 row in set (0.00 sec)
```

```
mysql> select @@innodb_version;
```

```
+-----+
| @@innodb_version |
+-----+
| 1.1.8            |
+-----+
1 row in set (0.00 sec)
```

参见 MySQL5.5 手册：

Because the InnoDB storage engine has now replaced the built-in InnoDB, you no longer need to specify options like `--ignore-builtin-innodb` and `--plugin-load` during startup.

To take best advantage of current InnoDB features, we recommend specifying the following options in your configuration file:

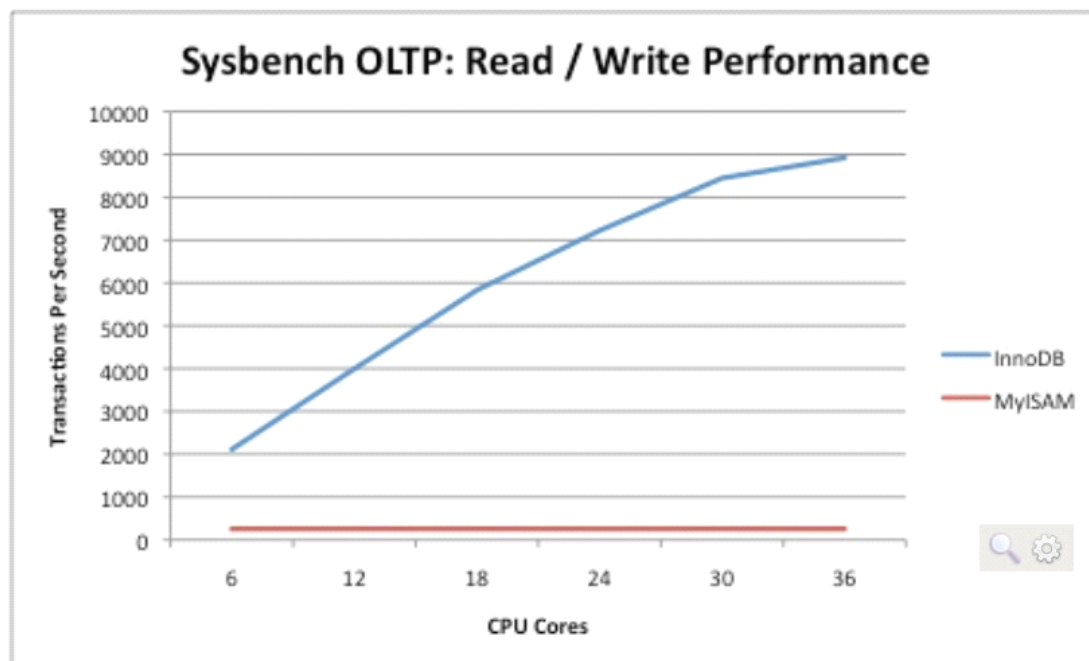
```
innodb_file_per_table=1
innodb_file_format=barracuda
innodb_strict_mode=1
```

MySQL5.1 里 built-in InnoDB 文件格式是 Antelope，而在 MySQL5.5 里 InnoDB plugin 文件格式要调整为 Barracuda。

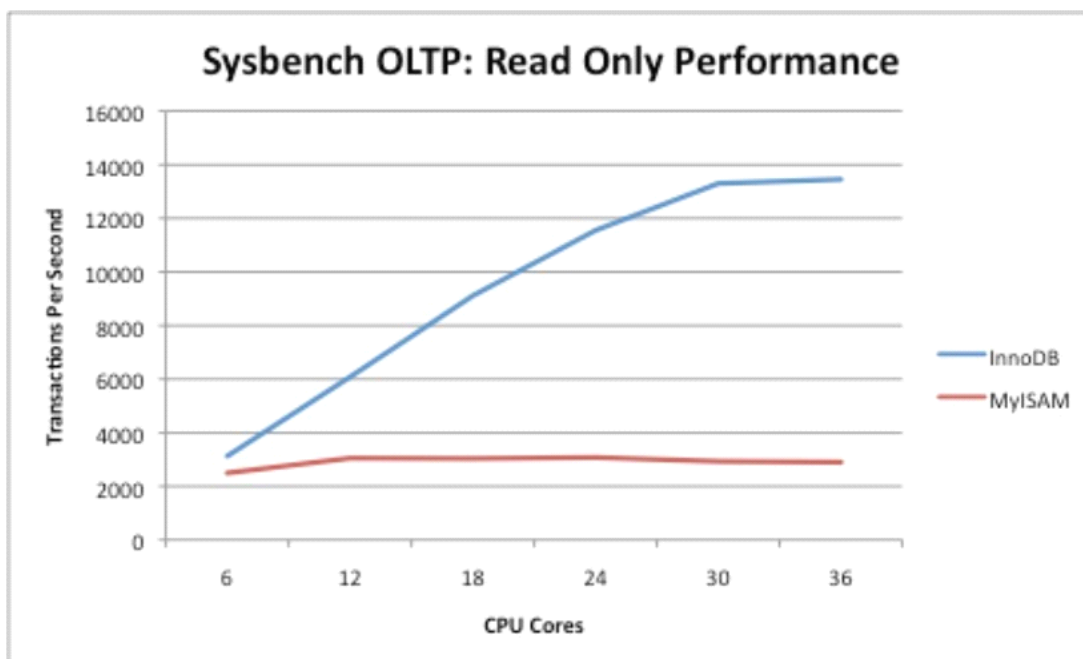
Barracuda 格式支持表压缩功能，TRUNCATE TABLE 的速度比以前要快。

下面是官方提供的 InnoDB and MyISAM 的压力测试结果：

Read-Write Results



Read-Only Results



在 Sysbench 读和写及只读压力测试中，服务器分别采用了 6、12、18、24、30、36 核 CPU 来对比，并发连接数均设置为 64。从图中来看，随着 CPU 核数的增加，InnoDB 吞吐量就越好，反观 MyISAM 吞吐量几乎没有什么变化，显然，MyISAM 的表锁定机制降低了读和写的吞吐量。

由此，证明了为啥在 MySQL5.5.X 版本中 InnoDB 被设置为默认的存储引擎。

附官方 InnoDB 参数：

```
--innodb_purge_threads=1
--innodb_file_format=barracuda
--innodb-buffer-pool-size=8192M
--innodb_support_xa=FALSE
--innodb_flush_method=O_DIRECT
--innodb-flush-log-at-trx-commit=2
--innodb-log-file-size=2000M
--innodb-log-buffer-size=64M
--innodb-io-capacity=200
--skip-innodb-adaptive-hash-index
--innodb-read-io-threads=8
--innodb-write-io-threads=8
--innodb_change_buffering=all
--innodb_stats_on_metadata=off
--innodb-buffer-pool-instances=12
--skip-grant-tables
--max_tmp_tables=100
--query_cache_size=0
--query_cache_type=0
```

```
--max_connections=1000
--max_prepared_stmt_count=1048576
--sort_buffer_size=32768
```

参考 MySQL5.5 手册：

InnoDB is a high-reliability and high-performance storage engine for MySQL. Starting with MySQL 5.5, it is the default MySQL storage engine. Key advantages of InnoDB include:

- Its design follows the [ACID](#) model, with [transactions](#) featuring [commit](#), [rollback](#), and [crash-recovery](#) capabilities to protect user data.
- Row-level [locking](#) and Oracle-style [consistent reads](#) increase multi-user concurrency and performance.
- **InnoDB** tables arrange your data on disk to optimize common queries based on [primary keys](#). Each **InnoDB** table has a primary key index called the [clustered index](#) that organizes the data to minimize I/O for primary key lookups.
- To maintain data integrity, **InnoDB** also supports [FOREIGN KEY](#) referential-integrity constraints.
- You can freely mix **InnoDB** tables with tables from other MySQL storage engines, even within the same statement. For example, you can use a join operation to combine data from **InnoDB** and **MEMORY** tables in a single query.

14.4.7.1. Overview of InnoDB Performance

InnoDB has always been highly efficient, and includes several unique architectural elements to assure high performance and scalability. The latest InnoDB storage engine includes new features that take advantage of advances in operating systems and hardware platforms, such as multi-core processors and improved memory allocation systems. In addition, new configuration options let you better control some InnoDB internal subsystems to achieve the best performance with your workload.

Starting with MySQL 5.5 and InnoDB 1.1, the built-in InnoDB storage engine within MySQL is upgraded to the full feature set and performance of the former InnoDB Plugin. This change makes these performance and scalability enhancements available to a much wider audience than before, and eliminates the separate installation step of the InnoDB Plugin. After learning about the InnoDB performance features in this section, continue with [Chapter 8, Optimization](#) to learn the best practices for overall MySQL performance, and [Section 8.5, "Optimizing for InnoDB Tables"](#) in particular for InnoDB tips and guidelines.

1.1.2 充分利用 CPU 多核的处理能力。

在 MySQL5.1.X 版本，innodb_file_io_threads 参数默认是 4，在 linux 系统上是不可更改的，windows 上可以调整。这个参数的作用是，innodb 使用后台线程处理数据页上的读写 I/O（输入输出）请求。

在 MySQL5.5.X 版本，或者在 InnoDB Plugin1.0.4 以后，用两个新的参数取代了 innodb_file_io_threads，那就是 innodb_read_io_threads 和 innodb_write_io_threads，这样就可以支持 linux 平台调整，你可以根据你的 CPU 核数来更改，默认是 4。

假如你的 CPU 是 2 颗 8 核的，那么你可以设置：

```
innodb_read_io_threads = 8
innodb_write_io_threads = 8
```

如果你的数据库读操作比写操作多，那么可以设置：


```
innodb_read_io_threads = 10
innodb_write_io_threads = 6
```

根据你的情况加以设置。

注：这两个参数不支持动态改变，需要把该参数加入到 `my.cnf` 里，修改完后重启 MySQL 服务，允许值的范围从 1-64。

调整后，你可以用命令 `show engine innodb status\G`来查看：

```
show engine innodb status
-----
FILE I/O
-----
I/O thread 0 state: waiting for completed aio requests (insert buffer thread)
I/O thread 1 state: waiting for completed aio requests (log thread)
I/O thread 2 state: waiting for completed aio requests (read thread)
I/O thread 3 state: waiting for completed aio requests (read thread)
I/O thread 4 state: waiting for completed aio requests (read thread)
I/O thread 5 state: waiting for completed aio requests (read thread)
I/O thread 6 state: waiting for completed aio requests (write thread)
I/O thread 7 state: waiting for completed aio requests (write thread)
I/O thread 8 state: waiting for completed aio requests (write thread)
I/O thread 9 state: waiting for completed aio requests (write thread)
```

参见 MySQL5.5 手册：

In place of `innodb_file_io_threads`, two new configuration parameters are introduced in the InnoDB storage engine 1.0.4, which are effective on all supported platforms. The two parameters `innodb_read_io_threads` and `innodb_write_io_threads` signify the number of background threads used for read and write requests respectively. You can set the value of these parameters in the MySQL option file (`my.cnf` or `my.ini`). These parameters cannot be changed dynamically. The default value for these parameters is 4 and the permissible values range from 1-64.

The purpose of this change is to make InnoDB more scalable on high end systems. Each background thread can handle up to 256 pending I/O requests. A major source of background I/O is the read-ahead requests. InnoDB tries to balance the load of incoming requests in such way that most of the background threads share work equally. InnoDB also attempts to allocate read requests from the same extent to the same thread to increase the chances of coalescing the requests together. If you have a high end I/O subsystem and you see more than $64 \times \text{innodb_read_io_threads}$ pending read requests in `SHOW ENGINE INNODB STATUS`, you might gain by increasing the value of `innodb_read_io_threads`.

1.1.3 提高刷新脏页数量和合并插入数量，改善磁盘 IO 处理能力。

在 MySQL5.1.X 版本，由于代码写死，最多只会刷新 100 个脏页到磁盘，合并 20 个插入缓冲，即使磁盘有能力处理更多的请求，也只会处理这么多，这样在更新量较大（比如大批量 INSERT）的时候，脏页刷新可能会跟不上，导致性能下降。

而在 MySQL5.5.X 版本里，innodb_io_capacity 参数可以动态调整刷新脏页的数量，在一定程度上解决了这一问题。

innodb_io_capacity 参数默认是 200，单位页。设置的大小取决于你的硬盘的 IOPS，即每秒的输入输出量(或读写次数)。

下面有一些值供大家参考：

innodb_io_capacity	磁盘配置
200	单盘 SAS/SATA
2000	SAS*12 RAID10
5000	SSD
50000	FUSION-IO

注：此参数支持动态改变，但需要 SUPER 权限。

```
SET GLOBAL innodb_io_capacity = 2000;
```

参见 MySQL5.5 手册：

14.4.7.11. Controlling the Master Thread I/O Rate

The **master thread** in InnoDB is a thread that performs various tasks in the background. Most of these tasks are I/O related, such as flushing dirty pages from the buffer pool or writing changes from the insert buffer to the appropriate secondary indexes. The master thread attempts to perform these tasks in a way that does not adversely affect the normal working of the server. It tries to estimate the free I/O bandwidth available and tune its activities to take advantage of this free capacity. Historically, InnoDB has used a hard coded value of 100 IOPs (input/output operations per second) as the total I/O capacity of the server.

You can set the value of this parameter in the MySQL option file (**my.cnf** or **my.ini**) or change it dynamically with the **SET GLOBAL** command, which requires the **SUPER** privilege.

1.1.4 增加了自适应刷新脏页功能。

这个功能是在 InnoDB Plugin 引入的。InnoDB 刷新脏页的规则是，当超过 innodb_max_dirty_pages_pct 设定的值后，或者当重做日志 ib_logfile 文件写满了以后，或者是机器空闲的时候，这三种情况下才会把 InnoDB_Buffer_Pool 的脏页刷入磁盘。

当写操作很频繁的时候，重做日志 ib_logfile 切换的次数就会很频繁，每当一个写满后，就会进行大批量将脏页刷入磁盘，会对系统的整体性能造成不小的影响。为了避免过大的磁盘 IO，innodb_adaptive_flushing 自适应刷新，使用一个全新的算法，会根据重做日志 ib_logfile 生成的速度和刷新频率来将脏页刷入磁盘，这样重做日志 ib_logfile 还没有写满时，也可以刷新一定的量。

innodb_adaptive_flushing 参数默认开启，可动态更新。

参见 MySQL5.5 手册：

InnoDB uses a new algorithm to estimate the required rate of flushing, based on the speed of redo log generation and the current rate of flushing. The intent is to smooth overall performance by ensuring that buffer flush activity keeps up with the need to keep the buffer pool “clean”. Automatically adjusting the rate of flushing can help to avoid steep dips in throughput, when excessive buffer pool flushing limits the I/O capacity available for ordinary read and write activity.

InnoDB uses its log files in a circular fashion. Before reusing a portion of a log file, InnoDB flushes to disk all dirty buffer pool pages

whose redo entries are contained in that portion of the log file, a process known as a [sharp checkpoint](#). If a workload is write-intensive, it generates a lot of redo information, all written to the log file. If all available space in the log files is used up, a sharp checkpoint occurs, causing a temporary reduction in throughput. This situation can happen even though `innodb_max_dirty_pages_pct` is not reached.

InnoDB uses a heuristic-based algorithm to avoid such a scenario, by measuring the number of dirty pages in the buffer pool and the rate at which redo is being generated. Based on these numbers, InnoDB decides how many dirty pages to flush from the buffer pool each second. This self-adapting algorithm is able to deal with sudden changes in the workload.

1.1.5 让 `InnoDB_Buffer_Pool` 缓冲池热数据存活更久。

`InnoDB_Buffer_Pool` 缓冲区有两个区域，一个是 `sublist of new blocks` 区域（经常被访问的数据——热数据），一个是 `sublist of old blocks` 区域（不最近访问的数据）。当用户访问数据时，如果缓冲区里有直接返回，否则会从磁盘读入到缓冲区的 `sublist of old blocks` 区域，然后再移动到 `sublist of new blocks` 区域，通过 LRU 最近最少使用算法踢出旧数据页。

但这样，也许会遇到一个问题，假如有些 SQL 语句做统计用全表扫描，例如 `select * from t1`，或者做一次 `mysqldump`，这时就会进入到 `sublist of new blocks` 区域，把一些真正热数据给“踢走”，这样就会造成缓冲区的数据进进出出，导致频繁磁盘 I/O。

所以就这个问题，从 MySQL5.5.X 版本开始，`innodb_old_blocks_pct` 参数可以控制进入缓冲区 `sublist of old blocks` 区域的数量，默认是 37，占整个缓冲池的比例为 3/8。当全表扫描一个大表时，或者做 `mysqldump`，就可以 `innodb_old_blocks_pct` 设置的小些，例如 `innodb_old_blocks_pct=5`，使数据块进入少量 `sublist of old blocks` 区域，移动到 `sublist of new blocks` 区域，从而让更多的热数据不被踢出。当你访问一个小表时，或者 `select` 查询结果很少，那么就可以保持默认的 `innodb_old_blocks_pct=37`，或者可以设置的更大，`innodb_old_blocks_pct=50`。

另外，另一个参数 `innodb_old_blocks_time`，当访问 `sublist of old blocks` 区域里的数据块时，并不是马上就移动到 `sublist of new blocks` 区域，而是先让其停留在 `sublist of old blocks` 区域 `innodb_old_blocks_time`（微秒），然后再将其移动到 `sublist of new blocks` 区域，这样延缓了 `sublist of new blocks` 区域里的数据不会马

上就被踢出。

用 `show engine innodb status\G`; 可以查看当前的信息:

```
Total memory allocated 1107296256; in additional pool allocated 0
Dictionary memory allocated 80360
Buffer pool size 65535
Free buffers 0
Database pages 63920
Old database pages 23600
Modified db pages 34969
Pending reads 32
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 414946, not young 2930673
1274.75 youngs/s, 16521.90 non-youngs/s
Pages read 486005, created 3178, written 160585
2132.37 reads/s, 3.40 creates/s, 323.74 writes/s
Buffer pool hit rate 950 / 1000, young-making rate 30 / 1000 not 392 / 1000
Pages read ahead 1510.10/s, evicted without access 0.00/s
LRU len: 63920, unzip_LRU len: 0
I/O sum[43690]:cur[221], unzip sum[0]:cur[0]
```

Old database pages 23600: 在旧区域存放着有多少个页

Pages made young: 移动到新区域有多少个页

Pages made not young: 没有移动到新区域有多少个页

youngs/s: 每秒移动到新区域有多少个

non-youngs/s: 每秒没有移动到新区域有多少个

young-making rate: 移动到新区域的比例

young-making not rate: 没有移动到新区域的比例

如果你没有全表扫描, 发现 youngs/s 的值很小, 那么你就应该增大 `innodb_old_blocks_pct` 或者减少 `innodb_old_blocks_time`。

如果你全表扫描, 发现 non-youngs/s 很小, 那么你就应该增大 `innodb_old_blocks_time`。

参见 MySQL5.5 手册:

When scanning large tables that cannot fit entirely in the buffer pool, setting `innodb_old_blocks_pct` to a small value keeps the data that is only read once from consuming a significant portion of the buffer pool. For example, setting `innodb_old_blocks_pct=5` restricts this data that is only read once to 5% of the buffer pool.

When scanning small tables that do fit into memory, there is less overhead for moving pages around within the buffer pool, so you can

leave `innodb_old_blocks_pct` at its default value, or even higher, such as `innodb_old_blocks_pct=50`.

Specifies how long in milliseconds (ms) a block inserted into the old sublist must stay there after its first access before it can be moved to the new sublist. The default value is 0: A block inserted into the old sublist moves immediately to the new sublist the first time it is accessed, no matter how soon after insertion the access occurs. If the value is greater than 0, blocks remain in the old sublist until an access occurs at least that many ms after the first access. For example, a value of 1000 causes blocks to stay in the old sublist for 1 second after the first access before they become eligible to move to the new sublist.

- If you see very low `youngs/s` values when you do not have large scans going on, that indicates that you might need to either reduce the delay time, or increase the percentage of the buffer pool used for the old sublist. Increasing the percentage makes the old sublist larger, so blocks in that sublist take longer to move to the tail and be evicted. This increases the likelihood that they will be accessed again and be made young.
- If you do not see a lot of `non-youngs/s` when you are doing large table scans (and lots of `youngs/s`), to tune your delay value to be larger.

1.1.6 加快了 InnoDB 的数据恢复时间。

一般地，MySQL/InnoDB 都是运行在普通的 PC Server + Linux(Unix)上，虽然不期待小型机+AIX 的高可用，但想尽一切办法缩短 MySQL 的不可用时间，仍然是 DBA 的目标。

根据经验，主机 OS 崩溃、硬件故障，仍然是影响 MySQL 可用性的最主要因素，当这些故障(OS、硬件)恢复后，另一个非常耗时的恢复就是 InnoDB 自己的恢复时间。

一般主机发生一次重启，正常大约小于 5 分钟，但此时 InnoDB 恢复可能需要 40 分钟或者更久(这依赖于 Buffer Pool、脏页面比例、TPS 等因素)。试想，如果每次能够把故障时间控制在 10 分钟之内，那么通过应用容错、Cache 支持等办法，用户体验和可用时间 都将有进一步的提升。

而 Mysql5.5 版本里，通过算法和内存管理上的改进，将 crash recovery 大大缩短了，这也就意味着以后 redo log 可以顶着 4G 用了(xtraDB 可以超过 4G)，这样可以在很大程度上降低 IO 需求、从而极大地提高 InnoDB 的写性能。

在这里，你无须任何操作即可实现快速恢复。下面，针对 MySQL5.1.59 Innodb 和 MySQL5.5 Innodb Plugin 1.1.X，我们来一次破坏性实验，来验证恢复时间。

两台机器均为虚拟机，内存 1G，参数设置的均一致，如下：

```
innodb_log_file_size = 300M
innodb_log_files_in_group = 3
innodb_log_buffer_size = 16M
innodb_max_dirty_pages_pct = 75
innodb_force_recovery = 0
innodb_buffer_pool_size = 600M
innodb_flush_log_at_trx_commit = 0
```

Sysbench 是一个模块化的、跨平台、多线程基准测试工具，主要用于评估测试各种不同系统参数下的数据库负载情况。

它主要包括以下几种方式的测试：

- 1、cpu 性能
- 2、磁盘 io 性能
- 3、调度程序性能
- 4、内存分配及传输速度
- 5、POSIX 线程性能
- 6、数据库性能(OLTP 基准测试)

目前 sysbench 主要支持 MySQL,pgsql,oracle 这 3 种数据库。

一、安装

首先，在 <http://sourceforge.net/projects/sysbench> 下载源码包。

接下来，按照以下步骤安装：

```
# tar zxvf sysbench-0.4.8.tar.gz
# cd sysbench-0.4.8
# ./configure --with-mysql-includes=/usr/local/mysql/include
--with-mysql-libs=/usr/local/mysql/lib
# make && make install
```

二、开始测试

命令及参数：

```
sysbench --test=oltp --mysql-table-engine=innodb \
--oltp-table-size=9000000 \
--max-requests=10000 \
--num-threads=16 \
--mysql-host=127.0.0.1 \
--mysql-port=3306 \
--mysql-user=root \
--mysql-password=123456 \
--mysql-db=test \
--mysql-socket=/tmp/mysql.sock prepare
```

然后另开一个终端，执行 sleep 120;kill -9 mysqld，我们跑 2 分钟后，然后强杀 MySQL 进程，然后再启动 MySQL，恢复的过程和时间如下：

MySQL5.1 恢复过程	MySQL5.5 恢复时间
120630 21:19:19 mysqld_safe Starting mysqld daemon with databases from /usr/local/mysql/data	120630 21:45:39 mysqld_safe Starting mysqld daemon with databases from /usr/local/mysql/data
120630 21:19:19 [Note] Plugin 'FEDERATED' is disabled.	120630 21:45:39 [Warning] You need to use --log-bin to make --binlog-format

<p>120630 21:19:19 InnoDB: Initializing buffer pool, size = 600.0M</p> <p>120630 21:19:19 InnoDB: Completed initialization of buffer pool</p> <p>InnoDB: Log scan progressed past the checkpoint lsn 0 337236631</p> <p>120630 21:19:20 InnoDB: Database was not shut down normally!</p> <p>InnoDB: Starting crash recovery.</p> <p>InnoDB: Reading tablespace information from the .ibd files...</p> <p>InnoDB: Restoring possible half-written data pages from the doublewrite InnoDB: buffer...</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 0 342479360</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 0 347722240</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 0 352965120</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 0 358208000</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 0 363450880</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 0 368693760</p> <p>.....</p> <p>.....</p> <p>.....</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 0 756666880</p> <p>120630 21:20:03 InnoDB: Starting an apply batch of log records to the database...</p> <p>InnoDB: Progress in percents: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99</p> <p>InnoDB: Apply batch completed</p>	<p>work.</p> <p>120630 21:45:39 [Note] Plugin 'FEDERATED' is disabled.</p> <p>120630 21:45:39 InnoDB: The InnoDB memory heap is disabled</p> <p>120630 21:45:39 InnoDB: Mutexes and rw_locks use InnoDB's own implementation</p> <p>120630 21:45:39 InnoDB: Compressed tables use zlib 1.2.3</p> <p>120630 21:45:39 InnoDB: Using Linux native AIO</p> <p>120630 21:45:39 InnoDB: Initializing buffer pool, size = 600.0M</p> <p>120630 21:45:40 InnoDB: Completed initialization of buffer pool</p> <p>120630 21:45:40 InnoDB: highest supported file format is Barracuda.</p> <p>InnoDB: Log scan progressed past the checkpoint lsn 125985128</p> <p>120630 21:45:41 InnoDB: Database was not shut down normally!</p> <p>InnoDB: Starting crash recovery.</p> <p>InnoDB: Reading tablespace information from the .ibd files...</p> <p>InnoDB: Restoring possible half-written data pages from the doublewrite InnoDB: buffer...</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 131227648</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 136470528</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 141713408</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 146956288</p> <p>.....</p> <p>.....</p> <p>.....</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 540172288</p> <p>InnoDB: Doing recovery: scanned up to log sequence number 545415168</p> <p>120630 21:46:03 InnoDB: Starting an</p>
---	--

InnoDB: Doing recovery: scanned up to log sequence number 0 761909760	apply batch of log records to the database...
InnoDB: Doing recovery: scanned up to log sequence number 0 767152640	InnoDB: Progress in percents: 0 1 2 3 4 5
.....	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
.....	21 22 23 24 25 26 27 28 29 30 31 32 33
.....	34 35 36 37 38 39 40 41 42 43 44 45 46
InnoDB: Doing recovery: scanned up to log sequence number 0 997839360	47 48 49 50 51 52 53 54 55 56 57 58 59
InnoDB: Doing recovery: scanned up to log sequence number 0 999899389	60 61 62 63 64 65 66 67 68 69 70 71 72
120630 21:21:56 InnoDB: Starting an apply batch of log records to the database...	73 74 75 76 77 78 79 80 81 82 83 84 85
InnoDB: Progress in percents: 0 1 2 3 4 5	86 87 88 89 90 91 92 93 94 95 96 97 98
6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	99
21 22 23 24 25 26 27 28 29 30 31 32 33	InnoDB: Apply batch completed
34 35 36 37 38 39 40 41 42 43 44 45 46	InnoDB: Doing recovery: scanned up to log sequence number 550658048
47 48 49 50 51 52 53 54 55 56 57 58 59	InnoDB: Doing recovery: scanned up to log sequence number 555900928
60 61 62 63 64 65 66 67 68 69 70 71 72
73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98
99	InnoDB: Doing recovery: scanned up to log sequence number 755130368
InnoDB: Apply batch completed	InnoDB: Doing recovery: scanned up to log sequence number 755420180
120630 21:22:48 InnoDB: Started; log sequence number 0 999899389	InnoDB: 1 transaction(s) which must be rolled back or cleaned up
120630 21:22:48 [Note] Event Scheduler: Loaded 0 events	InnoDB: in total 7621 row operations to undo
120630 21:22:48 [Note]	InnoDB: Trx id counter is 600
/usr/local/mysql/bin/mysqld: ready for connections.	120630 21:47:10 InnoDB: Starting an apply batch of log records to the database...
Version: '5.1.59' socket: '/tmp/mysql.sock' port: 3306 MySQL Community Server (GPL)	InnoDB: Progress in percents: 0 1 2 3 4 5
	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
	21 22 23 24 25 26 27 28 29 30 31 32 33
	34 35 36 37 38 39 40 41 42 43 44 45 46
	47 48 49 50 51 52 53 54 55 56 57 58 59
	60 61 62 63 64 65 66 67 68 69 70 71 72
	73 74 75 76 77 78 79 80 81 82 83 84 85
	86 87 88 89 90 91 92 93 94 95 96 97 98
	99
	InnoDB: Apply batch completed
	InnoDB: Starting in background the rollback of uncommitted transactions
	120630 21:47:21 InnoDB: Rolling back

	<p>trx with id 424, 7621 rows to undo</p> <p>InnoDB: Progress in percents: 1120630 21:47:21 InnoDB: Waiting for the background threads to start</p> <p>2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100</p> <p>InnoDB: Rolling back of trx id 424 completed 120630 21:47:22 InnoDB: Rollback of non-prepared transactions completed 120630 21:47:22 InnoDB: 1.1.8 started; log sequence number 755420180 120630 21:47:23 [Note] Event Scheduler: Loaded 0 events 120630 21:47:23 [Note] /usr/local/mysql/bin/mysqld: ready for connections. Version: '5.5.19' socket: '/tmp/mysql.sock' port: 3306 MySQL Community Server (GPL)</p>
--	--

MySQL5.1 恢复时间	MySQL5.5 恢复时间
3 分 28 秒	1 分 44 秒

从结果来看，MySQL5.5 的恢复时间要比 MySQL5.1 要快 2 倍多。当然这个结果取决的因素有很多，这依赖于 Buffer Pool、脏页面比例、TPS 等因素，在实际环境上，你需要根据自己的情况来加以测试。

参见 MySQL5.5 手册：

14.4.7.16. Improvements to Crash Recovery Performance

A number of optimizations speed up certain steps of the [recovery](#) that happens on the next startup after a crash. In particular, scanning the [redo log](#) and applying the redo log are faster than in MySQL 5.1 and earlier, due to improved algorithms for memory management. You do not need to take any actions to take advantage of this performance enhancement. If you kept the size of your redo log files artificially low because recovery took a long time, you can consider increasing the file size.

1.1.7 INNODB 同时支持多个 BufferPool 实例

InnoDB 用来缓存它的数据和索引的内存缓冲区的大小。你把这个值设得越高，访问表中数据需要得磁盘 I/O 越少。在一个专用的数据库服务器上，你可以设置这个参数达机器物理内存大小的80%。尽管如此，还是不要把它设置得太大，因为对物理内存的竞争可能在操作系统上导致内存调度。

`innodb_buffer_pool_size` 是 InnoDB 性能的决定性因素，当你的数据库大小，小于 `innodb_buffer_pool_size` 设置的缓冲池大小，那么此时数据库的性能是最好的，因为客户端访问的数据都在内存里。

InnoDB_Buffer_Pool 缓冲池复制管理着 free list（初始化空闲页，为每一个 page 指定一个 block 头结构，并初始化各种 mutex 与 rw-lock，将 page 加入 Buffer_Pool 的 free list 链表，等待分配），flush list（缓冲池产生的脏页（数据库被修改，但未写入磁盘），当 `innodb_max_dirty_pages_pct` 超过设置的值，会把修改时间越早的 page 刷进磁盘），LRU（在内存中但最近又不用数据块，按照最近最少使用算法，MySQL 会根据哪些数据属于 LRU 而将其移出内存而腾出空间来加载另外的数据。）等，当 InnoDB_Buffer_Pool 缓冲池达到好几十 G 时，某个线程正在更新缓冲池而造成其它线程必须等待的瓶颈。

在 MySQL5.5 里，可以通过 `innodb_buffer_pool_instances` 参数来增加 InnoDB_Buffer_Pool 实例的个数，使用哈希函数将读取缓存的数据页随机分配到一个缓冲池里面，这样每个缓冲区实例就可以分别管理着自己的 free list, flush list, LRU，来解决此问题。

简单的说，就是以前是由一个人负责管理着上百台服务器，当任务较多时，势必会出现差错，领导安排的任务不能在指定的时间里完成，现在由多个人一起管理，每个人分配的机器数量少了，管理上也轻松了，这样可以更有效的提高工作效率。

注：`innodb_buffer_pool_size` 必须大于 1G，生成 InnoDB_Buffer_Pool 多实例才有效，最多支持 64 个 InnoDB_Buffer_Pool 实例。

修改 `my.cnf` 配置文件，添加如下：

```
innodb_buffer_pool_instances = 3
```

调整后，你可以用命令 `show engine innodb status\G`来查看：

```
-----  
BUFFER POOL AND MEMORY
```

```
-----  
Total memory allocated 1282572288; in additional pool allocated 0
```

Dictionary memory allocated 22706
 Buffer pool size 76799
 Free buffers 76657
 Database pages 142
 Old database pages 0
 Modified db pages 0
 Pending reads 0
 Pending writes: LRU 0, flush list 0, single page 0
 Pages made young 0, not young 0
 0.00 youngs/s, 0.00 non-youngs/s
 Pages read 142, created 0, written 0
 0.16 reads/s, 0.00 creates/s, 0.00 writes/s
 Buffer pool hit rate 875 / 1000, young-making rate 0 / 1000 not 0 / 1000
 Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
 LRU len: 142, unzip_LRU len: 0
 I/O sum[0]:cur[0], unzip sum[0]:cur[0]

----- INDIVIDUAL BUFFER POOL INFO -----

--BUFFER POOL 0

Buffer pool size 25600
 Free buffers 25506
 Database pages 94
 Old database pages 0
 Modified db pages 0
 Pending reads 0
 Pending writes: LRU 0, flush list 0, single page 0
 Pages made young 0, not young 0
 0.00 youngs/s, 0.00 non-youngs/s
 Pages read 94, created 0, written 0
 0.08 reads/s, 0.00 creates/s, 0.00 writes/s
 Buffer pool hit rate 917 / 1000, young-making rate 0 / 1000 not 0 / 1000
 Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
 LRU len: 94, unzip_LRU len: 0
 I/O sum[0]:cur[0], unzip sum[0]:cur[0]

--BUFFER POOL 1

Buffer pool size 25599
 Free buffers 25551
 Database pages 48
 Old database pages 0
 Modified db pages 0
 Pending reads 0
 Pending writes: LRU 0, flush list 0, single page 0
 Pages made young 0, not young 0

```

0.00 youngs/s, 0.00 non-youngs/s
Pages read 48, created 0, written 0
0.08 reads/s, 0.00 creates/s, 0.00 writes/s
Buffer pool hit rate 750 / 1000, young-making rate 0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 48, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
---BUFFER POOL 2
Buffer pool size      25600
Free buffers          25600
Database pages        0
Old database pages 0
Modified db pages    0
Pending reads 0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 0, created 0, written 0
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 0, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]

```

参见 MySQL5.5 手册：

When the InnoDB buffer pool is large, many data requests can be satisfied by retrieving from memory. You might encounter bottlenecks from multiple threads trying to access the buffer pool at once. Starting in InnoDB storage engine 1.1 and MySQL 5.5, you can enable multiple buffer pools to minimize this contention. Each page that is stored in or read from the buffer pool is assigned to one of the buffer pools randomly, using a hashing function. Each buffer pool manages its own free lists, flush lists, LRUs, and all other data structures connected to a buffer pool, and is protected by its own buffer pool mutex.

To enable this feature, set the `innodb_buffer_pool_instances` configuration option to a value from 1 (the default) to 64 (the maximum). This option only takes effect when you set the `innodb_buffer_pool_size` to a size of 1 gigabyte or more. The total size you specify is divided up among all the buffer pools. We recommend specifying a combination of `innodb_buffer_pool_instances` and `innodb_buffer_pool_size` so that each buffer pool instance is at least 1 gigabyte.

1.1.8 可关闭自适应哈希索引

如果一个表几乎完全驻留在内存，在其上执行查询最快的方法就是使用哈希索引。InnoDB 有一个自动机制，它监视对为一个表定义的索引的索引搜索。如果 InnoDB 注意到查询会从建立一个哈希索引中获益，它会自动地这么做，无须 DBA 人工加以干涉。

注意，哈希索引总是基于表上已存在的 B 树索引来建立。根据 InnoDB 对 B 树索引观察的搜索方式，InnoDB 会在为该 B 树定义的任何长度的键的一个前缀上建立哈希索引。哈希索引可以是部分的：它不要求整个 B 树索引被缓存在缓冲池。InnoDB 根据需要对被经常访问的索引的那些页面建立哈希索引。

通常情况下，哈希索引可以提高查询性能，但是，在高并发情况下，会造成 RW-latch 争用，进而堵塞很多进程。

你可以 `show engine innodb status\G`来监控 SEMAPHORES 这项，如果你发现有很多 waits，那么应该关闭该功能，提升系统性能。

下面是我一个测试机的情况：

```
-----
SEMAPHORES
-----
OS WAIT ARRAY INFO: reservation count 38019, signal count 29541
Mutex spin waits 26137, rounds 282623, OS waits 7904
RW-shared spins 25122, rounds 637286, OS waits 17326
RW-excl spins 4071, rounds 400926, OS waits 12336
Spin rounds per wait: 10.81 mutex, 25.37 RW-shared, 98.48 RW-excl
```

```
-----
INSERT BUFFER AND ADAPTIVE HASH INDEX
-----
Ibuf: size 1, free list len 29, seg size 31, 2 merges
merged operations:
  insert 1, delete mark 2, delete 0
discarded operations:
  insert 0, delete mark 0, delete 0
Hash table size 1245217, node heap has 12 buffer(s)
2765.81 hash searches/s, 2513.37 non-hash searches/s
```

Waits 次数并不是很多，且使用哈希索引的比例 50%，那么就不用关闭自适应哈希索引。

参见 MySQL5.5 手册：

The adaptive hash index mechanism allows InnoDB to take advantage of large amounts of memory, something typically done only by database systems specifically designed for databases that reside entirely in memory. Normally, the automatic building and use of adaptive hash indexes improves performance. However, sometimes, the read/write lock that guards access to the adaptive hash index may become a source of contention under heavy workloads, such as multiple concurrent joins.

You can monitor the use of the adaptive hash index and the contention for its use in the "SEMAPHORES" section of the output of the `SHOW ENGINE INNODB STATUS` command. If you see many threads waiting on an RW-latch created in `btr0sea.c`, then it might be useful to disable adaptive hash indexing.

1.1.9 可选用内存分配程序使用控制

在 InnoDB 刚刚被开发的时候，当时多核 CPU 还并不是这么流行，当时也

并没有针对多核 CPU 优化的内存分配器（memory allocator libraries），所以 InnoDB 实现了一个自己的内存分配子系统（mem subsystem），这个子系统是通过单个互斥量（mutex）实现管理的，而这可能是一个瓶颈。除此，InnoDB 还对 OS 自带的内存分配管理（malloc 和 free）作了一次简单封装，这些管理函数实现也类似地通过互斥量的机制实现的，所以这并没有解决问题。

从 MySQL5.5.X 版本开始，用户可以控制 InnoDB 是使用自带的内存分配程序，还是使用当前部署的操作系统中现有的更高效的内存分配程序。通过在 MySQL 5.5 选项文件 my.cnf 中设置新的系统配置参数 innodb_use_sys_malloc，可方便地进行控制。默认设置值为 1，表示 InnoDB 使用操作系统的内存分配程序。

TCMalloc（Thread-Caching Malloc）是 google 开发的开源工具——“[google-perftools](#)”中的成员。与标准的 glibc 库的 malloc 相比，TCMalloc 在内存的分配上效率和速度要高得多，可以在很大程度上提高 MySQL 服务器在高并发情况下的性能，降低系统负载。

下面介绍下 TCMalloc 的安装和使用：

安装 libunwind（32位操作系统忽略此步骤）

```
# wget http://download.savannah.gnu.org/releases/libunwind/libunwind-0.99.tar.gz
# tar zxvf libunwind-0.99.tar.gz
# cd libunwind-0.99
# CFLAGS=-fPIC ./configure --enable-shared
# make CFLAGS=-fPIC
# make CFLAGS=-fPIC install
```

安装 google-perftools

```
# wget http://lndmp-web-server.googlecode.com/files/google-perftools-1.7.tar.gz
# tar zxvf google-perftools-1.7.tar.gz
# cd google-perftools-1.7
# ./configure
# make && make install
# vi /usr/local/mysql/bin/mysqld_safe
在# executing mysqld_safe 的下一行，加入以下内容
```

```
export LD_PRELOAD=/usr/local/lib/libtcmalloc.so
```

重启 Mysql 服务

```
# service mysqld restart
```

使用 lsof 命令查看 tcmalloc 是否生效

```
# lsof -n |grep tcmalloc
```

如果可以显示类似下列信息，则表明 MySQL 已经成功加载 tcmalloc

mysqld	13093	mysql	mem	REG	253,0
--------	-------	-------	-----	-----	-------

下面，我用 SysBench 压力测试了两种内存管理模式的性能对比。

由于是虚拟机，SysBench 参数设置的很小：

```
#!/bin/bash

i=1
while true
do
    while [ $i -le 1000 ]
    do

/usr/local/bin/sysbench --test=oltp --mysql-table-engine=innodb
--oltp-table-size=10000 --max-requests=100 --num-threads=6 --
mysql-host=192.168.110.140 --mysql-port=3306 --mysql-user=admin
--mysql-password=123456 --mysql-db=test --mysql-socket=/tmp/
mysql.sock run >> semi_replication.txt

        let i++
    done

break
done
```

系统：Red Hat Enterprise Linux Server release 6.0 (Santiago)

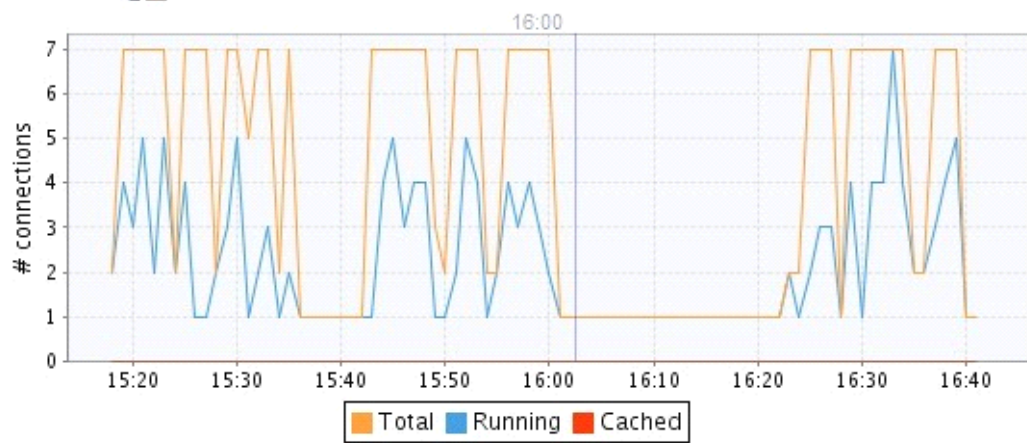
内核：2.6.32-71.el6.x86_64

左边是使用 InnoDB 自带的内存分配程序

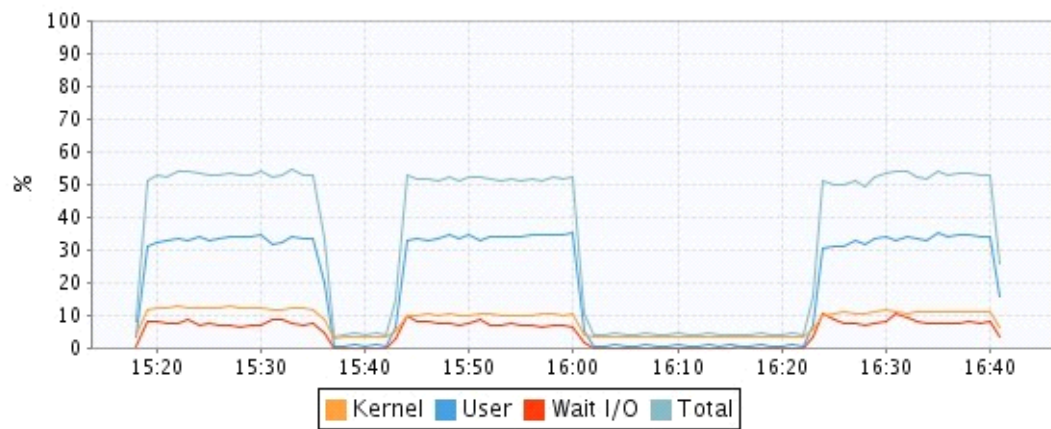
中间是使用谷歌的 TCMalloc 内存分配程序

右边是使用系统自带的 Malloc 内存分配程序

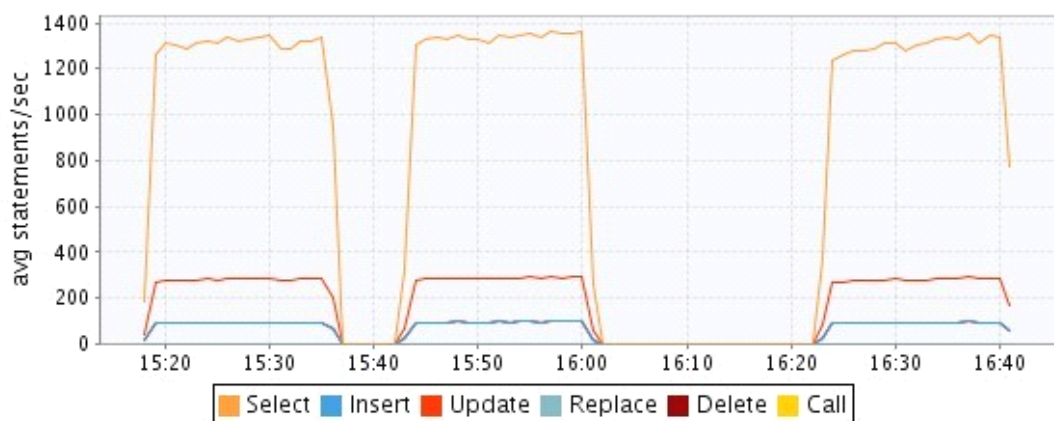
Connections



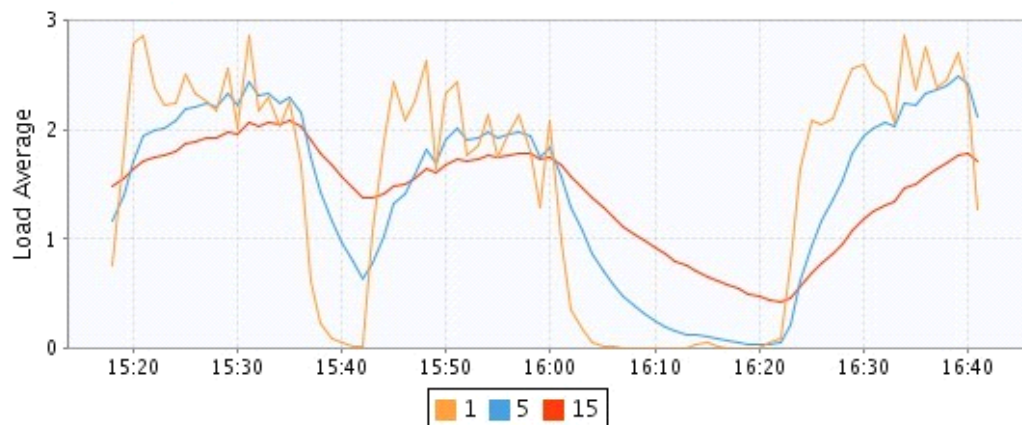
CPU Utilization



Database Activity



Load Average



结论：从结果上看，使用谷歌的 TCMalloc 内存分配程序对系统的压力相对要小一些。当然如果是真正的物理机，性能的差异会更明显，有兴趣的读者可以一试，期待着你的测试结果。

参见 MySQL5.5 手册：

14.4.7.3. Using Operating System Memory Allocators

When InnoDB was developed, the memory allocators supplied with operating systems and run-time libraries were often lacking in performance and scalability. At that time, there were no memory allocator libraries tuned for multi-core CPUs. Therefore, InnoDB implemented its own memory allocator in the `mem` subsystem. This allocator is guarded by a single mutex, which may become a [bottleneck](#). InnoDB also implements a wrapper interface around the system allocator (`malloc` and `free`) that is likewise guarded by a single mutex.

Today, as multi-core systems have become more widely available, and as operating systems have matured, significant improvements have been made in the memory allocators provided with operating systems. New memory allocators perform better and are more scalable than they were in the past. The leading high-performance memory allocators include [Hoard](#), [libumem](#), [mtmalloc](#), [ptmalloc](#), [tbbmalloc](#), and [TCMalloc](#). Most workloads, especially those where memory is frequently allocated and released (such as multi-table joins), benefit from using a more highly tuned memory allocator as opposed to the internal, InnoDB-specific memory allocator.

1.1.10 提高了默认 innodb 线程并发数

InnoDB 使用操作系统线程来处理用户事务请求，它是这样工作的：当 InnoDB 收到一个用户的请求，如果此时超过 `innodb_thread_concurrency` 预先设置的并发线程数量，那么就会按照 `innodb_thread_sleep_delay` 预先设定的值休眠 N 秒后再次尝试连接，重试两次的机制是为了减少 CPU 的上下文切换的次数，以降低 CPU 消耗。如果请求被接受了，则会获得一个 `innodb_concurrency_tickets` 默认 500 次的通行证，在次数用完之前，该线程重新请求时无须再进行前面所说 `innodb_thread_concurrency` 的检查。如果还没有被接受，那么就会进入队列中，直到最终被处理掉。

从 MySQL5.5.X 版本开始，`innodb_thread_concurrency` 被默认设置为 0，表示不限制并发数，以下是该参数的历史默认值。

InnoDB Version	MySQL Version	Default value	Default limit of concurrent threads	Value to allow unlimited threads
Built-in	Earlier than 5.1.11	20	No limit	20 or higher
Built-in	5.1.11 and newer	8	8	0
InnoDB storage engine before 1.0.3	(corresponding to Plugin)	8	8	0
InnoDB storage engine 1.0.3 and newer	(corresponding to Plugin)	0	No limit	0

注：`innodb_thread_concurrency = 0` 时，`innodb_thread_sleep_delay` 参数就无效了。同样 `innodb_concurrency_tickets` 也没了意思，这里推荐设置为 0，更好去发挥 CPU 多核处理能力，提高并发量。

参见 MySQL5.5 手册：

14.4.7.6. Changes Regarding Thread Concurrency

InnoDB uses operating system [threads](#) to process requests from user transactions. (Transactions may issue many requests to InnoDB before they commit or roll back.) On modern operating systems and servers with multi-core processors, where context switching is efficient, most workloads run well without any limit on the number of concurrent threads. Scalability improvements in MySQL 5.5 and up reduce the need to limit the number of concurrently executing threads inside InnoDB.

1.1.11 预读算法的变化

InnoDB 有两种预读算法提高 I/O 性能，一种是线性预读，另一种是随机预读。

线性预读：当顺序读取 extent 块（包含 64 个 page）innodb_read_ahead_threshold 设置的 page 页数量时，触发一个异步读取请求，将下一个页提前读取到 buffer pool 中。在 MySQL5.1.X 版本时，顺序读取 extent 块最后一个页时，InnoDB 决定是否将下一个页提前读取到 Innodb_Buffer_Pool 缓冲池中。

随机预读：在 Innodb_Buffer_Pool 缓冲池中发现同一个 extent 块内有若干个页，那么会触发一个异步读取请求，把剩余的页页读取进来，随机预读增加了不必要的复杂性，常常导致性能下降，在 MySQL5.5.X 版本时，已经将其删除了。

innodb_read_ahead_threshold 参数默认是 56，可动态更改。你可以 show engine innodb status\G，来查看当前的情况：

----- BUFFER POOL AND MEMORY

.....
.....
.....

Pages read ahead 1510.10/s, evicted without access 0.00/s

.....

Pages read ahead：表示每秒预读了多少页

evicted without access：表示预读的页没有被访问，每秒被踢出了多少页。

如果你发现有很多 evicted without access，说明你设置的值过小了，应该增大。

参见 MySQL5.5 手册：

Linear read-ahead is based on the access pattern of the pages in the buffer pool, not just their number. You can control when InnoDB performs a read-ahead operation by adjusting the number of sequential page accesses required to trigger an asynchronous read request, using the configuration parameter `innodb_read_ahead_threshold`. Before this parameter was added, InnoDB would only calculate whether to issue an asynchronous prefetch request for the entire next extent when it read in the last page of the current extent.

Random read-ahead is a former technique that has now been removed as of MySQL 5.5. If a certain number of pages from the same extent (64 consecutive pages) were found in the buffer pool, InnoDB asynchronously issued a request to prefetch the remaining pages of the extent. Random read-ahead added unnecessary complexity to the InnoDB code and often resulted in performance degradation rather than improvement. This feature is no longer part of InnoDB, and users should generally see equivalent or improved performance.

1.1.12 首次在 Linux 上实现了异步 I/O

这里先解释下，什么是同步 I/O，什么是异步 I/O？在同步文件 IO 中，线程启动一个 IO 操作然后就立即进入等待状态，直到 IO 操作完成后才醒来继续执行。而异步文件 IO 方式中，线程发送一个 IO 请求到内核，然后继续处理其他的事情，内核完成 IO 请求后，将会通知线程 IO 操作完成了。简单的说，如果是同步 I/O，当一个 I/O 操作执行时，应用程序必须等待，直到此 I/O 执行完。相反，异步 I/O 操作在后台运行，I/O 操作和应用程序可以同时运行，提高了系统性能。因此像数据库等应用往往会利用异步 I/O，使得多个 I/O 操作同时执行。

从 MySQL5.5.X 版本开始，在 Linux 系统上实现异步 I/O 功能，也就是 linux native AIO，为了使用 linux native AIO，可以利用 libaio 库，libaio 就是简单的对 linux native AIO 的系统调用进行了简单的封装，因此也可以直接通过系统调用来使用 linux native AIO。

在此之前，你要先安装 libaio rpm 包，`yum install libaio -y`

```
[root@TestPrfServer ~]# rpm -qa | grep libaio
libaio-0.3.106-5
libaio-0.3.106-5
[root@TestPrfServer ~]#
```

可以通过 `innodb_use_native_aio` 参数来选择是否启用异步 I/O，默认是 ON，可以开启的，此参数不支持动态修改。

下面分别是 MySQL5.5.X 和 MySQL5.1.X 启动时的信息。

MySQL5.5

```
120715 21:02:48 mysqld_safe Starting mysqld daemon with databases from /usr/local/mysql/data
120715 21:02:48 [Note] Plugin 'FEDERATED' is disabled.
120715 21:02:48 InnoDB: The InnoDB memory heap is disabled
120715 21:02:48 InnoDB: Mutexes and rw_locks use InnoDB's own implementation
120715 21:02:48 InnoDB: Compressed tables use zlib 1.2.3
120715 21:02:48 InnoDB: Using Linux native AIO
120715 21:02:48 InnoDB: Initializing buffer pool, size = 100.0M
120715 21:02:48 InnoDB: Completed initialization of buffer pool
120715 21:02:48 InnoDB: highest supported file format is Barracuda.
120715 21:02:50 InnoDB: Waiting for the background threads to start
120715 21:02:51 InnoDB: 1.1.8 started; log sequence number 890420134
120715 21:02:52 [Note] Event Scheduler: Loaded 0 events
120715 21:02:52 [Note] /usr/local/mysql/bin/mysqld: ready for connections.
Version: '5.5.19' socket: '/tmp/mysql.sock' port: 3306 MySQL Community Server (GPL)
```

MySQL5.1

```

120715 19:13:03 mysqld_safe Starting mysqld daemon with databases from /usr/local/mysql/data
120715 19:13:04 [Note] Plugin 'FEDERATED' is disabled.
120715 19:13:05 InnoDB: Initializing buffer pool, size = 100.0M
120715 19:13:05 InnoDB: Completed initialization of buffer pool
120715 19:13:07 InnoDB: Started; log sequence number 0 1983421835
120715 19:13:07 [Note] Recovering after a crash using mysql-bin
120715 19:13:07 [Note] Starting crash recovery...
120715 19:13:07 [Note] Crash recovery finished.
120715 19:13:08 [Note] Event Scheduler: Loaded 0 events
120715 19:13:08 [Note] /usr/local/mysql/bin/mysqld: ready for connections.
Version: '5.1.59-log' socket: '/tmp/mysql.sock' port: 3306 MySQL Community Server (GPL)

```

通过 `cat /proc/slabinfo | grep kio` 命令，查看是否工作正常。

```

[root@vm01 ~]# cat /proc/slabinfo | grep kio
kiotx      11      20      192      20      1 : tunables 120      60      8 : slabdata      1      1      0
kiocb      16      30      128      30      1 : tunables 120      60      8 : slabdata      1      1      0
[root@vm01 ~]#

```

如果 `kiocb` 那项不为 0，代表异步 I/O 已工作。

1.1.13 恢复组提交

一个事务提交时，是采取先写日志后刷入磁盘的方式。假如，此时有多个用户同时提交，那么按照顺序把写入的事务日志页刷入到磁盘上，那么会使磁盘做多次 I/O 操作，从而降低 IOPS 吞吐率。

从 MySQL5.5.X 版本开始，采用组提交的方式来刷入到磁盘，也就是当有多个用户同时提交事务，那么就合并在一起一次性来刷入磁盘，大大提高了吞吐量。

举个搬饮料放入库房的例子：在之前，他每搬一箱饮料就放入库房，这样进入库房的频率就很高，他来回回也很累，索性，他每次搬 5 箱饮料，这样一次坨的东西也多了，频率也就变低了。

注：组提交工作模式只支持在 `sync_binlog = 0` 的情况下，同样，`innodb_support_xa` 也必须等于 0。其目的是为了保证 InnoDB 存储引擎的 redo log 事务日志与 binlog 日志的顺序一致性。

例如在 1 分钟里，有 5 条更新记录，其顺序是：

```

1) begin;update t1 set money= money +1000 where uid=11; commit;
2) begin;update t1 set money= money -2000 where uid=11; commit;
3) begin;insert into t2(uid,money) values(12,2000); commit;
4) begin;insert into t3(uid,money) values(13,3000); commit;
5) begin;insert into t4(uid,money) values(14,4000); commit;

```

未采用组提交模式时，t1 表的 uid=11 账户里有 1000 元，按照正常顺序是先存入

1000，然后再取出 2000 元，接着其他 uid 做后面的开户插入操作。而如果采用组提交模式，其顺序是：

```
1) begin;
update t1 set money= money +1000 where uid=11;
insert into t2(uid,money) values(12,2000);
insert into t3(uid,money) values(13,3000);
insert into t4(uid,money) values(14,4000);
commit;
2) begin;update t1 set money= money -2000 where uid=11; commit;
```

在 sync_binlog 等于 1 时，先写入 binlog_cache，事务提交后会马上刷入 binlog 日志，这样按照 MySQL 源码编写是不能合并一起提交。而如果 sync_binlog 等于 1 时，仍旧采取组提交模式，其顺序有可能是这样：

```
1) begin;update t1 set money= money -2000 where uid=11;commit;
2) begin;
update t1 set money= money +1000 where uid=11;
insert into t2(uid,money) values(12,2000);
insert into t3(uid,money) values(13,3000);
insert into t4(uid,money) values(14,4000);
commit;
```

这样就出现了问题，t1 表的 uid=11 账户里只有 1000 元，取出 2000 元会提示报错，并回滚掉，导致该用户取钱失败，如果开启了同步复制，也会把 binlog 日志同步到 Slave 上，所以采取组提交模式，sync_binlog 必须等于 0。

此特性无须 **DBA** 更改任何参数即可实现。

参见 MySQL5.5 手册：

14.4.7.10. Group Commit

InnoDB, like any other **ACID**-compliant database engine, flushes the **redo log** of a transaction before it is committed. Historically, InnoDB used **group commit** functionality to group multiple such flush requests together to avoid one flush for each commit. With group commit, InnoDB issues a single write to the log file to perform the commit action for multiple user transactions that commit at about the same time, significantly improving throughput.

Group commit in InnoDB worked until MySQL 4.x, and works once again with MySQL 5.1 with the InnoDB Plugin, and MySQL 5.5 and higher. The introduction of support for the distributed transactions and Two Phase Commit (2PC) in MySQL 5.0 interfered with the InnoDB group commit functionality. This issue is now resolved.

The group commit functionality inside InnoDB works with the Two Phase Commit protocol in MySQL. Re-enabling of the group commit functionality fully ensures that the ordering of commit in the MySQL binlog and the InnoDB logfile is the same as it was before. It means it is **totally safe to use MySQL Enterprise Backup with InnoDB 1.0.4** (that is, the InnoDB Plugin with MySQL 5.1) and above. When the binlog is enabled, you typically also set the configuration option **sync_binlog=0**, because group commit for the binary log is only supported if it is set to 0.

Group commit is transparent; you do not need to do anything to take advantage of this significant performance improvement.

1.1.14 innodb 使用多个回滚段提升性能

InnoDB 现在可以使用多个回滚段来提升性能和可扩展性，并且能够极大地增加并发事务处理的数量，之前的几个 InnoDB 版本最多只能处理 1023 个并发事务处理操作。

现在 MySQL5.5 可以支持高达 128K 的并发事务处理操作，创建回滚数据（undo data）（来自插入、更新、和删除操作）。这种改进措施减少了在单个回滚段上的互斥争用，增加了吞吐量。

此特性无须 **DBA** 更改任何参数即可实现。

参见 MySQL5.5 手册：

14.4.7.19. Better Scalability with Multiple Rollback Segments

Starting in InnoDB 1.1 with MySQL 5.5, the limit on concurrent [transactions](#) is greatly expanded, removing a bottleneck with the InnoDB [rollback segment](#) that affected high-capacity systems. The limit applies to concurrent transactions that change any data; read-only transactions do not count against that maximum.

The single rollback segment is now divided into 128 segments, each of which can support up to 1023 transactions that perform writes, for a total of approximately 128K concurrent transactions. The original transaction limit was 1023.

Each transaction is assigned to one of the rollback segments, and remains tied to that rollback segment for the duration. This enhancement improves both scalability (higher number of concurrent transactions) and performance (less contention when different transactions access the rollback segments).

To take advantage of this feature, you do not need to create any new database or tables, or reconfigure anything. You must do a [slow shutdown](#) before upgrading from MySQL 5.1 or earlier, or some time afterward. InnoDB makes the required changes inside the [system tablespace](#) automatically, the first time you restart after performing a slow shutdown.

For more information about performance of InnoDB under high transactional load, see [Section 8.5.2, “Optimizing InnoDB Transaction Management”](#).

1.1.15 改善清除程序进度

InnoDB 中的清除操作是一类定期回收无用数据的操作。在之前的几个版本中，清除操作是主线程的一部分，这意味着运行时它可能会堵塞其它的数据库操作。

从 MySQL5.5.X 版本开始，该操作运行于独立的线程中，并支持更多的并发数。用户可通过设置 `innodb_purge_threads` 配置参数来选择清除操作是否使用单独线程，默认情况下参数设置为 0（不使用单独线程），设置为 1 时表示使用单独的清除线程。

注：`innodb_purge_threads` 参数不支持动态修改，需要添加到 `my.cnf` 里修改，并重启生效。当设置为 1 时，需要结合 `innodb_purge_batch_size` 参数来使用，默认值是 20，最大可设置 5000，这个参数一般不用调整，默认的即可。

参见 MySQL5.5 手册：

14.4.7.20. Better Scalability with Improved Purge Scheduling

Starting in InnoDB 1.1 with MySQL 5.5, the [purge](#) operations (a type of garbage collection) that InnoDB performs automatically can be done in a separate thread, rather than as part of the [master thread](#). This change improves scalability, because the main database operations run independently from maintenance work happening in the background.

To enable this feature, set the configuration option `innodb_purge_threads=1`, as opposed to the default of 0, which combines the purge operation into the master thread.

You might not notice a significant speedup, because the purge thread might encounter new types of contention; the single purge thread really lays the groundwork for further tuning and possibly multiple purge threads in the future. There is another new configuration option, `innodb_purge_batch_size` with a default of 20 and maximum of 5000. This option is mainly intended for experimentation and tuning of purge operations, and should not be interesting to typical users.

1.1.16 添加了删除缓冲和清除缓冲

当向一个表进行 `insert`、`delete` 或 `update` 时，里面的索引（聚集索引和非聚集索引）也会随即更新，主键（聚集索引）是按照顺序进行插入的，而非聚集索引则是分散性的插入。顺序读写的速度要比随机读写的速度快，表越大就越明显，插入的性能就会变低。

因此在 MySQL 5.1.X 版本里，InnoDB 引入了一种优化措施，当一个表做 `insert` 操作非聚集索引更新时，如果该非聚集索引页被读入 `InnoDB Buffer Pool` 缓冲池里，那么就直接更新非聚集索引，并使用正常的写脏数据块方法闪存到磁盘中；如果没有读入缓冲池里，则使用插入缓冲区来缓存非聚集索引页的变化，直到该页被读入 `InnoDB Buffer Pool` 缓冲池里，执行插入缓存合并操作，并使用正常的写脏数据块方法闪存到磁盘中，从而提高了插入性能。

然而，插入缓冲区占用部分 `InnoDB Buffer Pool` 缓冲池，减少了可用内存来缓存数据页。如果数据和索引全部读入 `InnoDB Buffer Pool` 缓冲池，并且你的表有相对较少的非聚集索引，那么就可以关闭 InnoDB 的插入缓冲功能，上一段也介绍了，如果该非聚集索引页被读入 `InnoDB Buffer Pool` 缓冲池里，那么就直接更新非聚集索引，并使用正常的写脏数据块方法闪存到磁盘中，插入缓冲区在这里没有什么作用了，并且还占用一定的内存，这种情况关闭较好。

从 MySQL 5.5.X 版本开始，还为删除操作扩展了同样的功能（首先是删除标记操作，然后使用收集/清除所有已删除记录的清除操作）。现在可以使用 `innodb_change_buffering` 配置参数来控制删除缓冲和既有插入缓冲功能，默认是 `all`，此参数支持动态设置：

```
SET GLOBAL innodb_change_buffering = all;
```

参见 MySQL 5.5 手册：

14.4.7.4. Controlling InnoDB Change Buffering

When `INSERT`, `UPDATE`, and `DELETE` operations are done to a table, often the values of indexed columns (particularly the values of secondary keys) are not in sorted order, requiring substantial I/O to bring secondary indexes up to date. InnoDB has an `insert buffer` that caches changes to secondary index entries when the relevant `page` is not in the `buffer pool`, thus avoiding I/O operations by not reading in the page from the disk. The buffered changes are merged when the page is loaded to the buffer pool, and the updated page is later flushed to disk using the normal mechanism. The InnoDB main thread merges buffered changes when the server is nearly idle, and during a `slow shutdown`.

Because it can result in fewer disk reads and writes, this feature is most valuable for workloads that are I/O-bound, for example applications with a high volume of DML operations such as bulk inserts.

However, the insert buffer occupies a part of the buffer pool, reducing the memory available to cache data pages. If the working set almost fits in the buffer pool, or if your tables have relatively few secondary indexes, it may be useful to disable insert buffering. If the working set entirely fits in the buffer pool, insert buffering does not impose any extra overhead, because it only applies to pages that are not in the buffer pool.

You can control the extent to which InnoDB performs insert buffering with the system configuration parameter `innodb_change_buffering`. You can turn on and off buffering for inserts, delete operations (when index records are initially marked for deletion) and purge operations (when index records are physically deleted). An update operation is represented as a combination of an insert and a delete. In MySQL 5.5 and higher, the default value is changed from `inserts` to `all`.

1.1.17 控制自旋锁 Spin Lock 轮训间隔

先了解下何谓自旋锁？它是为实现保护共享资源而提出一种锁机制。其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就是说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，“自旋”一词就是因此而得名。

为了防止自旋锁循环过快，耗费 `cpu`，在 MySQL 5.5.X 版本里，引入了 `innodb_spin_wait_delay` 参数，作用是控制轮训间隔，也就是说每次轮训时，休息一会然后再轮训。比如我们用一个死循环监控服务状态，那么每次 `sleep 5` 秒，然后再检查。例如：

```
#!/bin/bash
while true
do
    pstree -p mysql > /dev/null
    if [ $? -eq 0 ];then
        echo "OK."
    else
        echo "Mysql is down." | mail -s "aleat" hechunyang@139.com
    fi
    sleep 5
done
```

指的就是这个意思。

注：innodb_spin_wait_delay 参数，默认是 6，可动态可调。

```
set global innodb_spin_wait_delay=6;
```

参见 MySQL5.5 手册：

14.4.7.14. Control of Spin Lock Polling

Many InnoDB mutexes and rw-locks are reserved for a short time. On a multi-core system, it can be more efficient for a thread to continuously check if it can acquire a mutex or rw-lock for a while before sleeping. If the mutex or rw-lock becomes available during this polling period, the thread can continue immediately, in the same time slice. However, too-frequent polling by multiple threads of a shared object can cause "cache ping pong", different processors invalidating portions of each others' cache. InnoDB minimizes this issue by waiting a random time between subsequent polls. The delay is implemented as a busy loop.

1.1.18 快速创建、删除、更改索引

在 MySQL5.1.X 版本里，聚集索引创建和删除的过程：

- 1、创建一个和原表结构一样的空表，然后创建聚集索引；
- 2、拷贝原表的数据到新表，这时会对原表加一个排他锁，其他的会话 dml 操作会阻塞，从而保证数据的一致性；
- 3、复制完毕后删除掉原表，并把新表改名为原表。

非聚集索引创建和删除的过程：

- 1、创建一个和原表结构一样的空表，然后创建非聚集索引；
- 2、拷贝原表的数据到新表，这时会对原表加一个共享锁，其他的会话不能更新，但可以查询数据，从而保证数据的一致性；
- 3、复制完毕后删除掉原表，并把新表改名为原表。

在 MySQL5.5.X 版本开始，创建和删除非聚集索引不用复制整个表的内容，只需更新索引页，和之前相比，速度会更快。但创建聚集索引（主键）时，或者是外键时，还是需要复制整个表的内容，因为聚集索引是把 primary key 以及 row data 保存在一起的，而 secondary index 则是单独存放，然后有个指针指向 primary key。

下面是创建和删除非聚集索引的速度对比：

速度对比	MySQL5.5	MySQL5.1
------	----------	----------

表信息	<pre>mysql> select @@version; +-----+ @@version +-----+ 5.5.19-log +-----+ 1 row in set (0.01 sec) mysql> select count(*) from test1; +-----+ count(*) +-----+ 999999 +-----+ 1 row in set (0.94 sec)</pre>	<pre>mysql> select @@version; +-----+ @@version +-----+ 5.1.59-log +-----+ 1 row in set (0.00 sec) mysql> select count(*) from test1; +-----+ count(*) +-----+ 999999 +-----+ 1 row in set (2.60 sec)</pre>
增加索引	<pre>mysql> select @@version; +-----+ @@version +-----+ 5.5.19-log +-----+ 1 row in set (0.01 sec) mysql> create index IX_tid on test1(tid); Query OK, 0 rows affected (22.40 sec) Records: 0 Duplicates: 0 Warnings: 0 mysql></pre>	<pre>mysql> select @@version; +-----+ @@version +-----+ 5.1.59-log +-----+ 1 row in set (0.00 sec) mysql> create index IX_tid on test1(tid); Query OK, 999999 rows affected (45.79 sec) Records: 999999 Duplicates: 0 Warnings: 0 mysql></pre>
删除索引	<pre>mysql> select @@version; +-----+ @@version +-----+ 5.5.19-log +-----+ 1 row in set (0.01 sec) mysql> drop index IX_tid on test1; Query OK, 0 rows affected (0.22 sec) Records: 0 Duplicates: 0 Warnings: 0 mysql></pre>	<pre>mysql> select @@version; +-----+ @@version +-----+ 5.1.59-log +-----+ 1 row in set (0.00 sec) mysql> drop index IX_tid on test1; Query OK, 999999 rows affected (15.82 sec) Records: 999999 Duplicates: 0 Warnings: 0 mysql></pre>

从结果上看，MySQL5.5.X 的速度是较快的。

参见 MySQL5.5 手册：

14.4.2. Fast Index Creation in the InnoDB Storage Engine

In MySQL 5.5 and higher, or in MySQL 5.1 with the InnoDB Plugin, creating and dropping [secondary indexes](#) does not copy the contents of the entire table, making this operation much more efficient than with prior releases.

InnoDB has two types of indexes: the clustered index and secondary indexes. Since the clustered index contains the data values in its B-tree nodes, adding or dropping a clustered index does involve copying the data, and creating a new copy of the table. A secondary index, however, contains only the index key and the value of the primary key. This type of index can be created or dropped without copying the data in the clustered index. Because each secondary index contains copies of the primary key values (used to access the clustered index when needed), when you change the definition of the primary key, all secondary indexes are recreated as well.

While an InnoDB secondary index is being created or dropped, the table is locked in shared mode. Any writes to the table are blocked, but the data in the table can be read. When you alter the clustered index of a table, the table is locked in exclusive mode, because the data must be copied. Thus, during the creation of a new clustered index, all operations on the table are blocked.

Once a `CREATE INDEX` or `ALTER TABLE` statement that creates an InnoDB secondary index begins executing, queries can access the table for read access, but cannot update the table. If an `ALTER TABLE` statement is changing the clustered index for an InnoDB table, all queries wait until the operation completes.

MySQL 5.5 does not support efficient creation or dropping of `FOREIGN KEY` constraints. Therefore, if you use `ALTER TABLE` to add or remove a `REFERENCES` constraint, the child table is copied, rather than using Fast Index Creation.

1.1.19 InnoDB 支持创建压缩数据页

从 MySQL 5.5.X 版本开始，支持 InnoDB 数据页压缩，数据页的压缩使数据文件体积变小，减少磁盘 I/O，提高吞吐量，小成本的提高 CPU 利用率。尤其是对读多写少的应用，最为有效，同样的内存可以存储更多的数据，充分的“榨干”内存利用率。

它的工作原理是：当用户获取数据时，如果压缩的页没有在 `Innodb_Buffer_Pool` 缓冲池里，那么会从磁盘加载进去，并且在 `Innodb_Buffer_Pool` 缓冲池里开辟一个新的未压缩 16K 的数据页来解压缩加载进来的压缩页，为了减少磁盘 I/O 以及对页的解压，在缓冲池里同时存在压缩和未压缩的页。为了给其他需要的数据页腾出空间，缓冲池里会把未压缩的数据页踢出去，而保留压缩的页在内存，未压缩的页在一段时间内没有被访问，那么会直接写入磁盘里，因此缓冲池里中可能有压缩和未压缩的页，或者只有压缩页。

InnoDB 采用最近最少使用(LRU)算法，将经常被访问的热数据放入内存里。当访问一个压缩表时，InnoDB 使用一个自适应的 LRU 算法来实现内存中压缩页和未压缩页一个适当的平衡，其目的是为了避免当 CPU 繁忙时花费太多的时间用在解压缩上，也为了避免当 CPU 空闲时做过多的 I/O 操作在解压缩上。所以，当系统处于 I/O 瓶颈时，这个算法会踢出未压缩的页，而不是未压缩和压缩的页，为了让更多的页注入内存腾出空间。当系统处于 CPU 瓶颈时，这个算法会同时踢出未压缩的页和压缩的页，让更多的内存存放热数据，减少解压缩带来的开销。

在以前的版本，一个数据页是 16K，现在可以在建表时指定压缩的页是 1,2,4,8K，设置过小，会导致消耗更多的 CPU，通常设置为 8K。

注：必须采用文件格式 Barracuda，且独立表空间才支持数据页压缩。

```
innodb_file_format = Barracuda
```

```
innodb_file_per_table = 1
```

在建表的时候加入 `ROW_FORMAT=COMPRESSED KEY_BLOCK_SIZE=8` 即可，如：

```
CREATE TABLE `compressed` (  
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
```

```

`k` int(10) unsigned NOT NULL DEFAULT '0',
`c` char(120) NOT NULL DEFAULT "",
`pad` char(60) NOT NULL DEFAULT "",
PRIMARY KEY (`id`),
KEY `k` (`k`)
) ENGINE=InnoDB
DEFAULT CHARSET=gbk
ROW_FORMAT=COMPRESSED
KEY_BLOCK_SIZE=8

```

针对数据页 8K 和 16K，进行了一次压力测试，虚拟机内存 1G，Buffer_Pool 为 600M。

```

/usr/local/bin/sysbench --test=oltp --mysql-table-engine=innodb --oltp-table-size=10000000
--max-requests=1000 --num-threads=10 --mysql-host=192.168.110.140 --mysql-port=3306
--mysql-user=admin --mysql-password=123456 --mysql-db=test
--oltp-table-name=uncompressed --mysql-socket=/tmp/mysql.sock run

```

```

/usr/local/bin/sysbench --test=oltp --mysql-table-engine=innodb --oltp-table-size=10000000
--max-requests=1000 --num-threads=10 --mysql-host=192.168.110.140 --mysql-port=3306
--mysql-user=admin --mysql-password=123456 --mysql-db=test --oltp-table-name=compressed
--mysql-socket=/tmp/mysql.sock run

```



先创建 1 百万行记录的表，经过压缩的 8K 数据页的表要比未压缩 16K 的数据页体积小一半。

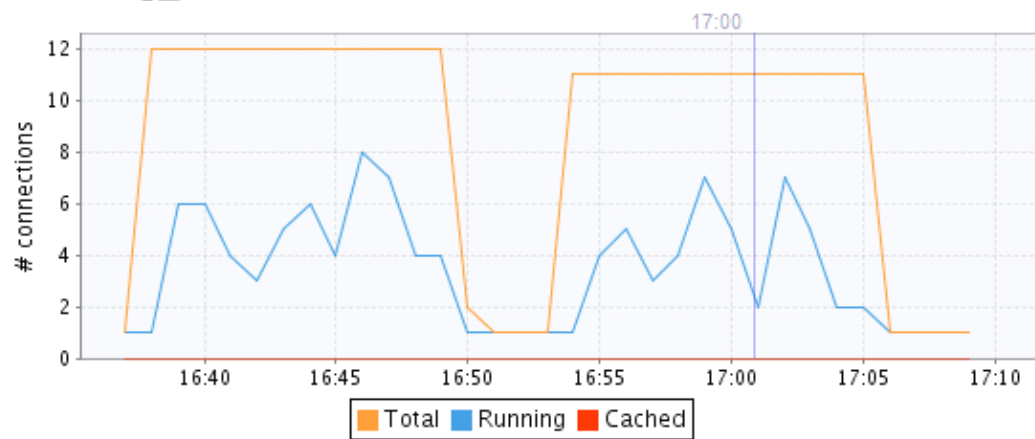
```



[root@hadoop-datanode5 test]# ll -h
总用量 3.5G
-rw-rw---- 1 mysql mysql 8.5K 7月 9 13:54 compressed.frm
-rw-rw---- 1 mysql mysql 1.2G 7月 9 17:06 compressed.ibd
-rw-rw---- 1 mysql mysql 8.5K 7月 9 13:40 uncompressed.frm
-rw-rw---- 1 mysql mysql 2.3G 7月 9 16:49 uncompressed.ibd
[root@hadoop-datanode5 test]#

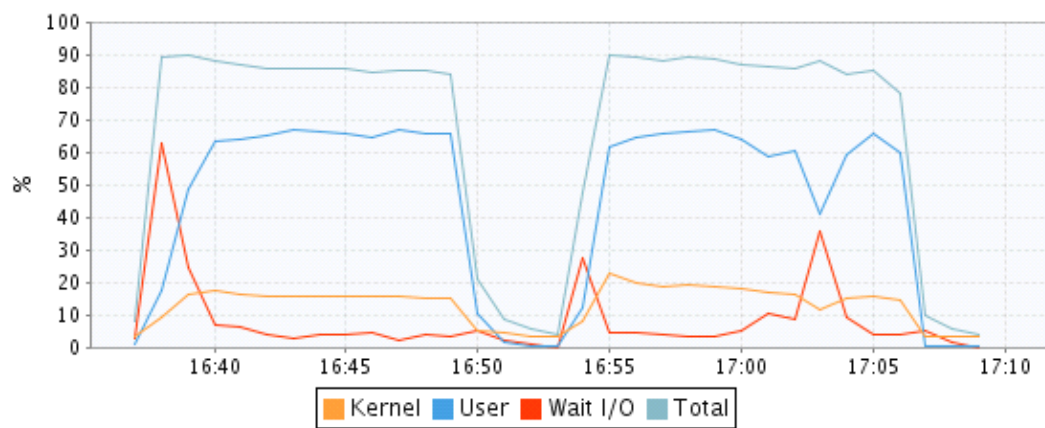
```

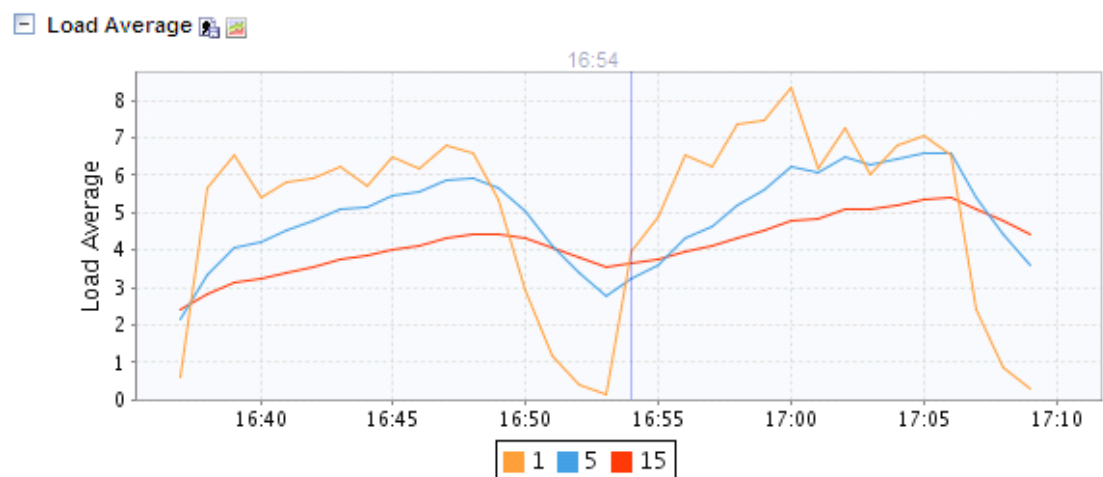
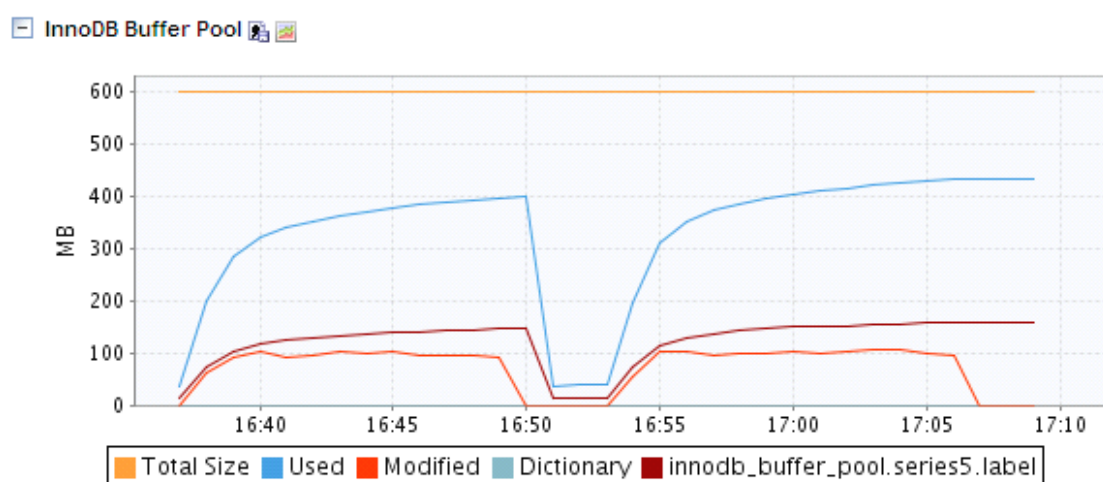
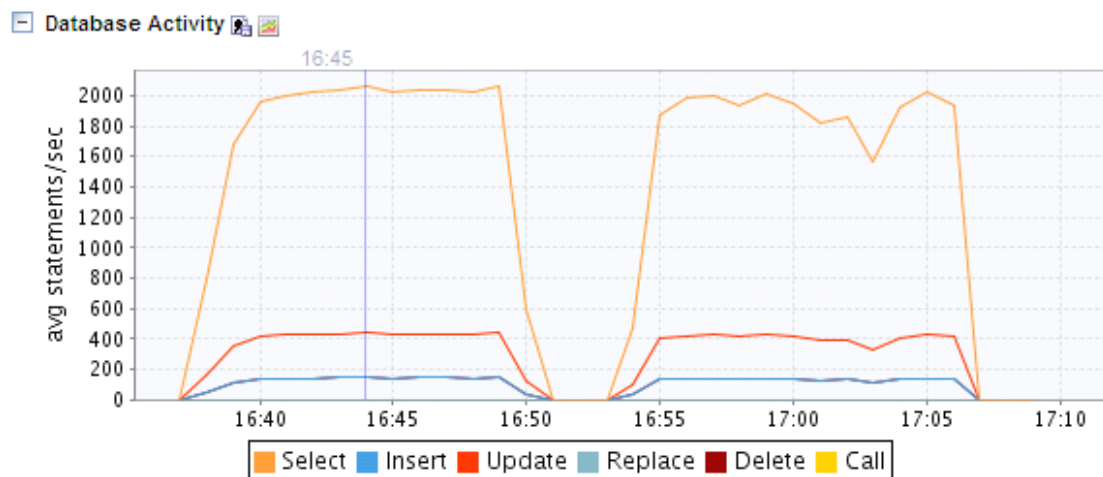
左图为未压缩，右图为压缩

Connections  



CPU Utilization  





从图中所看，被请求的数据页小于 InnoDB_Buffer_Pool 缓冲池大小，未压缩的性能要稍好于压缩过的，因为压缩会带来额外的 CPU 消耗，总体上差异不大。

下面把 Sysbench 参数调大，再压一次

```
/usr/local/bin/sysbench --test=oltp --mysql-table-engine=innodb --oltp-table-size=10000000
--max-requests=10000 --num-threads=10 --mysql-host=192.168.110.140 --mysql-port=3306
```

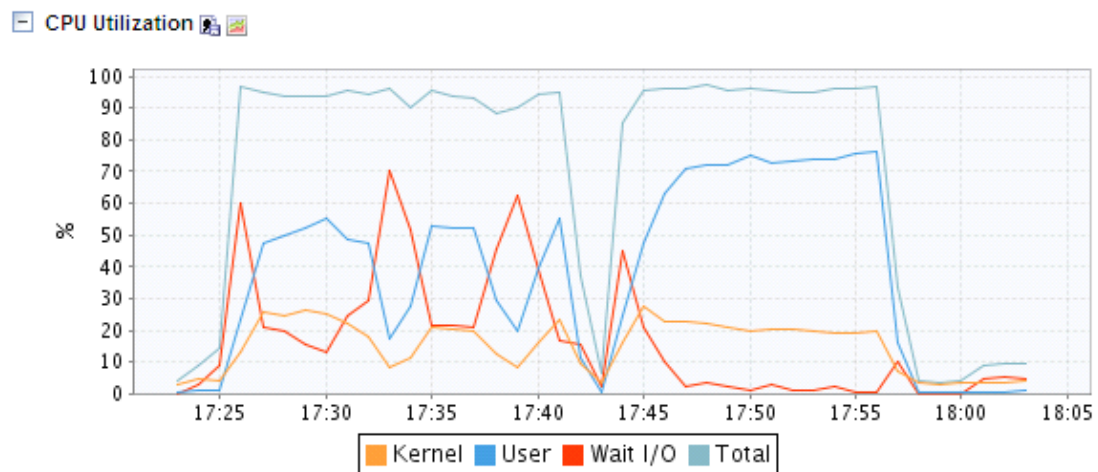
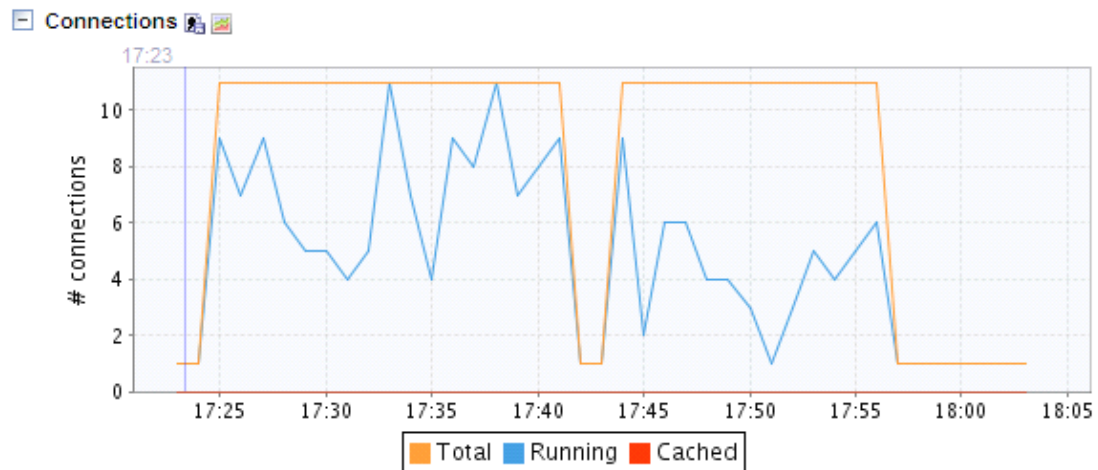
```

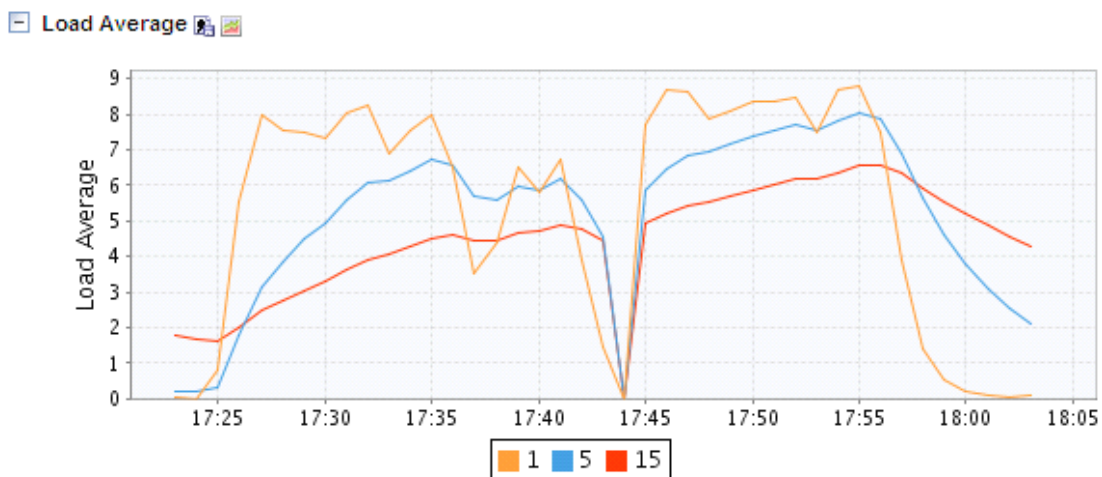
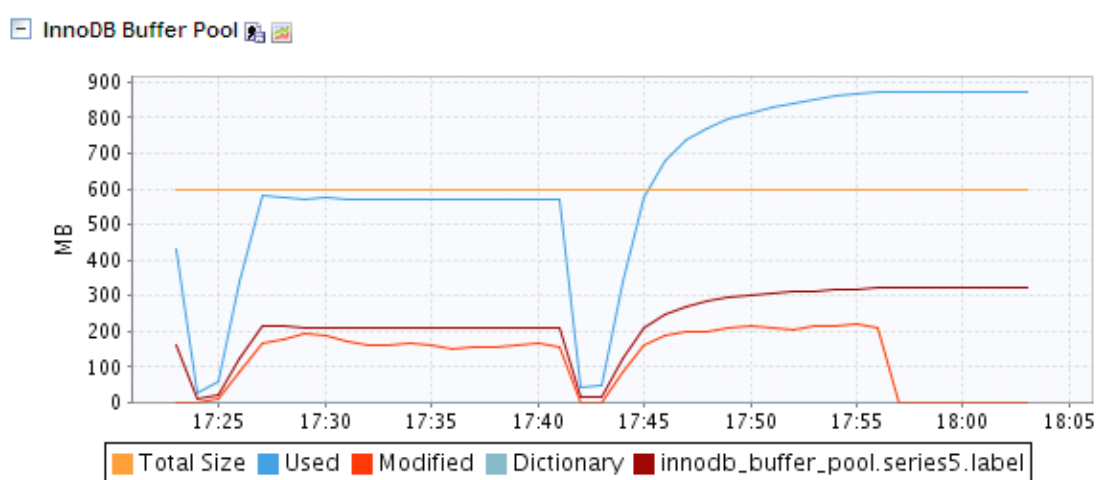
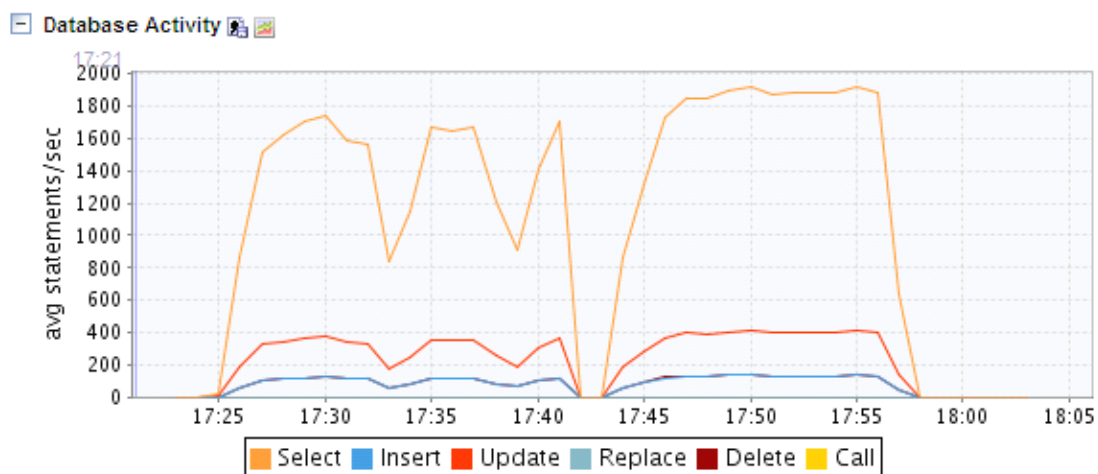
--mysql-user=admin                --mysql-password=123456                --mysql-db=test
--oltp-table-name=uncompressed --mysql-socket=/tmp/mysql.sock run

/usr/local/bin/sysbench  --test=oltp  --mysql-table-engine=innodb  --oltp-table-size=10000000
--max-requests=10000  --num-threads=10  --mysql-host=192.168.110.140  --mysql-port=3306
--mysql-user=admin --mysql-password=123456 --mysql-db=test --oltp-table-name=compressed
--mysql-socket=/tmp/mysql.sock run

```

左图为未压缩 16K，右图为压缩 8K





从图中所看，被请求的数据页大于 InnoDB_Buffer_Pool 缓冲池大小，压缩的性能要好于压缩过的，吞吐量也提高，最为明显 CPU Wait/IO 降低很多。

根据以上两种情况，你可根据自身的业务情况，来选择是否开启数据页压缩功能。

另附上大批量插入时的测试：

16K
mysql> show create table test1\G;
***** 1. row ***** Table: test1 Create Table: CREATE TABLE `test1` (`id` int(11) NOT NULL, `tid` int(11) DEFAULT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB DEFAULT CHARSET=latin1 1 row in set (0.04 sec)

8K
mysql> show create table test3\G;
***** 1. row ***** Table: test3 Create Table: CREATE TABLE `test3` (`id` int(11) NOT NULL, `tid` int(11) DEFAULT NULL, `name` varchar(10) DEFAULT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB DEFAULT CHARSET=gbk ROW_FORMAT=COMPRESSED KEY_BLOCK_SIZE=8 1 row in set (0.79 sec)

```

mysql> call test3();
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id:      8
Current database: test

Query OK, 1 row affected (7 min 38.60 sec)

mysql> use test;
No connection. Trying to reconnect...
Connection id:      3
Current database: *** NONE ***

Database changed
mysql> call test1();
Query OK, 1 row affected (4 min 24.86 sec)

```

在大批量插入的时候，几乎慢了 1 倍，8K 数据页性能要比 16K 数据页性能有所下降。

参见 MySQL5.5 手册：

14.4.3.1. Overview of Table Compression

Because processors and cache memories have increased in speed more than disk storage devices, many workloads are I/O-bound. Data [compression](#) enables smaller database size, reduced I/O, and improved throughput, at the small cost of increased CPU utilization. Compression is especially valuable for read-intensive applications, on systems with enough RAM to keep frequently-used data in memory.

Setting the compressed page size too large wastes some space, but the pages do not have to be compressed as often. If the compressed page size is set too small, inserts or updates may require time-consuming recompression, and the B-tree nodes may have to be split more frequently, leading to bigger data files and less efficient indexing.

Typically, you set the compressed page size to 8K or 4K bytes. Given that the maximum InnoDB record size is around 8K, `KEY_BLOCK_SIZE=8` is usually a safe choice.

Fundamentally, compression works best when the CPU time is available for compressing and uncompressing data. Thus, if your workload is I/O bound, rather than CPU-bound, you might find that compression can improve overall performance. When you test your application performance with different compression configurations, test on a platform similar to the planned configuration of the production system.

Compression and the InnoDB Buffer Pool

In a compressed InnoDB table, every compressed page (whether 1K, 2K, 4K or 8K) corresponds to an uncompressed page of 16K bytes. To access the data in a page, InnoDB reads the compressed page from disk if it is not already in the buffer pool, then uncompresses the page to its original 16K byte form. This section describes how InnoDB manages the buffer pool with respect to pages of compressed tables.

To minimize I/O and to reduce the need to uncompress a page, at times the buffer pool contains both the compressed and uncompressed form of a database page. To make room for other required database pages, InnoDB may “evict” from the buffer pool an uncompressed page, while leaving the compressed page in memory. Or, if a page has not been accessed in a while, the compressed form of the page may be written to disk, to free space for other data. Thus, at any given time, the buffer pool may contain both the compressed and uncompressed forms of the page, or only the compressed form of the page, or neither.

InnoDB keeps track of which pages to keep in memory and which to evict using a least-recently-used (LRU) list, so that “hot” or frequently accessed data tends to stay in memory. When compressed tables are accessed, InnoDB uses an adaptive LRU algorithm to achieve an appropriate balance of compressed and uncompressed pages in memory. This adaptive algorithm is sensitive to whether the system is running in an I/O-bound or CPU-bound manner. The goal is to avoid spending too much processing time uncompressing pages when the CPU is busy, and to avoid doing excess I/O when the CPU has spare cycles that can be used for uncompressing compressed pages (that may already be in memory). When the system is I/O-bound, the algorithm prefers to evict the uncompressed copy of a page rather than both copies, to make more room for other disk pages to become memory resident. When the system is CPU-bound, InnoDB prefers to evict both the compressed and uncompressed page, so that more memory can be used for “hot” pages and reducing the need to uncompress data in memory only in compressed form.

1.1.20 可动态关闭 InnoDB 更新元数据统计功能

`innodb_stats_on_metadata` 参数的作用是：每当查询 `information_schema` 元数据库里的表时，InnoDB 还会随机提取其他数据库每个表索引页的部分数据，来更新 `information_schema.STATISTICS` 表，来返回刚才你查询的结果。当你的表很大，并且数量很多时，耗费的时间就会很长，很多经常不访问的数据也会进入到 `InnoDB_Buffer_Pool` 缓冲池里，那么就会把缓冲池所污染。并且 `ANALYZE TABLE` 和 `SHOW TABLE STATUS` 语句也会造成 InnoDB 随机提取数据。

从 MySQL5.5.X 版本开始，你可以动态关闭 `innodb_stats_on_metadata`，默认是开启的。

```
set global innodb_stats_on_metadata = OFF;
```

那么有人问，如果我关闭了该功能，会不会造成数据统计的不精确？

答案是：不会的。下面我用一个例子，来加以证实：

#查看所有数据库大小

```
select sum(data_length+index_length)/1024/1024/1024 from  
information_schema.tables;
```

```
mysql> set global innodb_stats_on_metadata = ON;
Query OK, 0 rows affected (0.01 sec)

mysql> show variables like 'innodb_stats_on_metadata';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| innodb_stats_on_metadata | ON    |
+-----+-----+
1 row in set (0.05 sec)

mysql> select sum(data_length+index_length)/1024/1024/1024 from information_schema.tables;
+-----+-----+
| sum(data_length+index_length)/1024/1024/1024 |
+-----+-----+
| 0.231541509740 |
+-----+-----+
1 row in set (1.16 sec)

mysql> set global innodb_stats_on_metadata = OFF;
Query OK, 0 rows affected (0.01 sec)

mysql> show variables like 'innodb_stats_on_metadata';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| innodb_stats_on_metadata | OFF   |
+-----+-----+
1 row in set (0.04 sec)

mysql> select sum(data_length+index_length)/1024/1024/1024 from information_schema.tables;
+-----+-----+
| sum(data_length+index_length)/1024/1024/1024 |
+-----+-----+
| 0.231541509740 |
+-----+-----+
1 row in set (0.66 sec)
```

关闭元数据统计后，查询快了近 2 倍，结果跟之前的一样。如果你的表很大，并且很多，那么这个结果相差的就会更大。

参见 MySQL5.5 手册：

14.4.8.5. Controlling Optimizer Statistics Estimation

The MySQL query optimizer uses estimated statistics about key distributions to choose the indexes for an execution plan, based on the relative [selectivity](#) of the index. Certain operations cause InnoDB to sample random pages from each index on a table to estimate the [cardinality](#) of the index. (This technique is known as [random dives](#).) These operations include the `ANALYZE TABLE` statement, the `SHOW TABLE STATUS` statement, and accessing the table for the first time after a restart.

1.2 安全性、稳定性显著的改变

1.2.1 复制功能(Replication)加强

默认情况下，MySQL5.5 的复制功能是异步的，这意味着当谈到数据一致性时，主服务器及其从服务器是独立的。异步复制可以提供最佳的性能，因为主服务器在将更新的数据写入它的二进制日志（Binlog）文件中后，无需等待验证更

新数据是否已经复制到至少一台拓扑从服务器中，就可以自由处理其它进入的事务处理请求。虽然快，但这也同时带来了很高的风险，如果在主服务器或从服务器端发生故障，会造成主服务器/从服务器数据的不一致，甚至在恢复时造成数据丢失。

MySQL5.5 引入了一种半同步复制功能，该功能可以确保主服务器和访问链中至少一台从服务器之间的数据一致性和冗余。在这种配置结构中，一台主服务器和其许多从服务器都进行了配置，这样在复制拓扑中，至少有一台从服务器在父主服务器进行事务处理前，必须确认更新已经收到并写入了其中继日志 (Relay Log)。当出现超时，源主服务器必须暂时切换到异步复制模式重新复制，直到至少有一台设置为半同步复制模式的从服务器及时收到信息。

在后面的章节中，会对半复制同步的安装测试加以详细描述，这里不再叙述。

1.2.2 中继日志 relay-log 自我修复。

在 MySQL5.5.X 版本开始，增加了 relay_log_recovery 参数，这个参数的作用是：当 slave 从库宕机后，假如 Relay-Log 损坏了，导致一部分中继日志没有处理，则自动放弃所有未执行的 relay-log，并且重新从 MASTER 上获取日志，这样保证 relay-log 的完整。默认情况下该功能是关闭的，将 relay_log_recovery 的值设置为 1 时，可在 slave 从库上开启该功能，建议开启。

参见 MySQL5.5 手册：

Enables automatic relay log recovery immediately following server startup, which means that the replication slave discards all unprocessed relay logs and retrieves them from the replication master. This should be used following a crash on the replication slave to ensure that no possibly corrupted relay logs are processed. The default value is 0 (disabled). This global variable can be changed dynamically, or by starting the slave with the `--relay-log-recovery` option.

1.2.3 开启 InnoDB 严格检查模式

从 MySQL5.5.X 版本开始，你可以开启 InnoDB 严格检查模式，尤其采用了页数据压缩功能，建议开启。此功能是当 CREATE TABLE, ALTER TABLE and CREATE INDEX 语句时，如果写法有错误，不会有警告信息，而是直接抛出错误，好处是直接将问题扼杀在摇篮里。

innodb_strict_mode 参数默认为 OFF，建议开启，并支持动态开启。

```
set global innodb_strict_mode=1;
```

参见 MySQL5.5 手册：

The setting of InnoDB strict mode affects the handling of syntax errors on the `CREATE TABLE`, `ALTER TABLE` and `CREATE INDEX` statements. The strict mode also enables a record size check, so that an `INSERT` or `UPDATE` never fails due to the record being too large for the selected page size.

We recommend running in strict mode when using the `ROW_FORMAT` and `KEY_BLOCK_SIZE` clauses on `CREATE TABLE`, `ALTER TABLE`, and `CREATE INDEX` statements. Without strict mode, InnoDB ignores conflicting clauses and creates the table or index, with only a warning in the message log. The resulting table might have different behavior than you intended, such as having no compression when you tried to create a compressed table. When InnoDB strict mode is on, such problems generate an immediate error and the table or index is not created, avoiding a troubleshooting session later.

1.3 动态更改系统配置参数

1.3.1 支持动态更改独立表空间

在 MySQL5.1.X 版本里，如果想采用独立表空间，那么就必须修改 `my.cnf`，将 `innodb_file_per_table = 1`，然后重启 `mysql` 服务。

从 MySQL5.5.X 版本开始起，可以动态修改，该参数默认是关闭的，采用共享表空间。

```
set global innodb_file_per_table = 1;
```

注：如果使用 Barracuda 格式的表压缩功能，必须启动独立表空间，共享表空间不支持该功能。

参见 MySQL5.5 手册：

In MySQL 5.5 and higher, the configuration parameter `innodb_file_per_table` is dynamic, and can be set ON or OFF using the `SET GLOBAL`. Previously, the only way to set this parameter was in the MySQL option file (`my.cnf` or `my.ini`), and changing it required shutting down and restarting the server.

The default setting is OFF, so new tables and indexes are created in the system tablespace. Dynamically changing the value of this parameter requires the `SUPER` privilege and immediately affects the operation of all connections.

Tables created when `innodb_file_per_table` is enabled can use the Barracuda file format, and `TRUNCATE` returns the disk space for those tables to the operating system. The Barracuda file format in turn enables features such as table compression and the `DYNAMIC` row format. Tables created when `innodb_file_per_table` is off cannot use these features. To take advantage of those features for an existing table, you can turn on the file-per-table setting and run `ALTER TABLE t ENGINE=INNODB` for that table.

1.3.2 支持动态更改 InnoDB 锁超时时间

在 MySQL5.1.X 版本里，如果想更改 InnoDB 锁超时时间，那么就必须修改 `my.cnf`，将 `innodb_lock_wait_timeout = 10`，然后重启 `mysql` 服务。

在遇到锁等待时，经常会遇到如下所示，如果当初这个参数设置的过大，那么锁等待的时间就会很长。

ERROR HY000: Lock wait timeout exceeded; try restarting transaction

从 MySQL5.5.X 版本开始起，可以动态修改，该参数默认是 50 秒。

```
set global innodb_lock_wait_timeout = 10;
```

参见 MySQL5.5 手册：

In MySQL 5.5 and higher, the configuration parameter `innodb_lock_wait_timeout` can be set at runtime with the `SET GLOBAL` or `SET SESSION` statement. Changing the `GLOBAL` setting requires the `SUPER` privilege and affects the operation of all clients that subsequently connect. Any client can change the `SESSION` setting for `innodb_lock_wait_timeout`, which affects only that client.

In MySQL 5.1 and earlier, the only way to set this parameter was in the MySQL option file (`my.cnf` or `my.ini`), and changing it required shutting down and restarting the server.

1.4 Innodb 新参数汇总

`innodb_adaptive_flushing`

Command-Line Format	--innodb_adaptive_flushing=#	
Config-File Format	innodb_adaptive_flushing	
Option Sets Variable	Yes, innodb_adaptive_flushing	
Variable Name	innodb_adaptive_flushing	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	boolean
	Default	ON

解释：自适应刷新脏页，在没有达到 `innodb_max_dirty_pages_pct` 设置的值也会刷新，默认为 ON，可动态更改，建议不要更改。

`innodb_buffer_pool_instances`

Command-Line Format	-- innodb_buffer_pool_instances=#	
Config-File Format	innodb_buffer_pool_instances	
Option Sets Variable	Yes, innodb_buffer_pool_instances	
Variable Name	innodb_buffer_pool_instances	

Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values	
	Type	numeric
	Default	1
	Range	1 .. 64

解释：可以设置多个 `buffer_pool` 管理各自的缓冲池，这样会减少某个线程正在更新缓冲池而造成其它线程必须等待的瓶颈。默认为 1，不支持动态更改，`innodb_buffer_pool_size` 必须大于 1G，生成 `InnoDB_Buffer_Pool` 多实例才有效，最多支持 64 个 `InnoDB_Buffer_Pool` 实例。可根据实际内存大小，加以设置多个缓冲池。

`innodb_change_buffering`

Command-Line Format	<code>--innodb_change_buffering=#</code>	
Config-File Format	<code>innodb_change_buffering</code>	
Option Sets Variable	Yes, <code>innodb_change_buffering</code>	
Variable Name	<code>innodb_change_buffering</code>	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values (<= 5.5.3)	
	Type	enumeration
	Default	inserts
	Valid Values	inserts, none
	Permitted Values (>= 5.5.4)	
	Type	enumeration
	Default	all
	Valid Values	inserts, deletes, purges, changes, all, none

解释：可以设置插入 `inserts` 合并缓冲、删除 `deletes` 合并缓冲、清除 `purges` 合并缓冲，`changes`(Buffer both inserts and delete-marking)，`none`（Do not buffer any operations）默认是 `all`（buffer insert, delete-marking, and purge operations），可动态更改。一般情况下，采用默认值 `all` 即可。

innodb_file_format

Command-Line Format	--innodb_file_format=#	
Config-File Format	innodb_file_format	
Option Sets Variable	Yes, <u>innodb_file_format</u>	
Variable Name	innodb_file_format	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	string
	Default	Antelope

解释：innodb 的文件格式为 Antelope，innodb-plugin 的文件格式为 Barracuda，开启数据压缩页 8K，那么就必须设置为 Barracuda，以后会是 Cheetah，以动物的名字命名，默认为 Antelope，可动态更改，建议改为 Barracuda。

innodb_file_format_check

Command-Line Format	--innodb_file_format_check=#	
Config-File Format	innodb_file_format_check	
Option Sets Variable	Yes, <u>innodb_file_format_check</u>	
Variable Name	innodb_file_format_check	
Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values (<= 5.5.0)	
	Type	string
	Default	Antelope
	Permitted Values (>= 5.5.4)	
	Type	string
	Default	Barracuda
	Permitted Values (>= 5.5.5)	
	Type	boolean
	Default	ON

解释： 用来检查共享表空间文件格式，如果共享表空间的文件格式高于当前版

本，并且 `innodb_file_format_check` 设置为 `on`，在服务器启动时会报告错误：
InnoDB: Error: the system tablespace is in a file format that this version doesn't support.如果设置为 `off`，服务器可以启动，并收到一个警告信息，但这样很不安全，数据有可能会损坏。默认是 `on`，不支持动态更改，建议不要修改。

innodb_file_format_max

Version Introduced	5.5.5	
Command-Line Format	<code>--innodb_file_format_max=#</code>	
Config-File Format	<code>innodb_file_format_max</code>	
Option Sets Variable	Yes, <code>innodb_file_format_max</code>	
Variable Name	<code>innodb_file_format_max</code>	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	string
	Default	Antelope

解释：设置文件格式最高版本，默认是 Antelope，可动态更改，建议更改为 Barracuda。

innodb_io_capacity

Command-Line Format	<code>--innodb_io_capacity=#</code>	
Config-File Format	<code>innodb_io_capacity</code>	
Option Sets Variable	Yes, <code>innodb_io_capacity</code>	
Variable Name	<code>innodb_io_capacity</code>	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	numeric
	Default	200
	Min Value	100

解释：5.1.X 版本，由于代码写死，每秒只能刷新 100 个脏页，现在可以设置刷

新脏页的数量，提高磁盘 I/O 的吞吐率，默认是 200，可动态更改，一般六块 15000 转的磁盘做 RAID10，设置 2000 较合适。

innodb_old_blocks_pct

Command-Line Format	--innodb_old_blocks_pct=#	
Config-File Format	innodb_old_blocks_pct	
Variable Name	innodb_old_blocks_pct	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	numeric
	Default	37
	Range	5-95

解释：控制进入缓冲区 sublist of old blocks 区域的数量，默认是 37，占整个缓冲池的比例为 3/8，可动态更改，除非有很多全表扫描语句，一般采用默认即可。

innodb_old_blocks_time

Command-Line Format	--innodb_old_blocks_time=#	
Config-File Format	innodb_old_blocks_time	
Variable Name	innodb_old_blocks_time	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	numeric
	Default	0

解释：控制进入缓冲区 sublist of old blocks 区域停留的时间，超过后移动到 sublist of new blocks 区域，默认是 0，有访问即移动，可动态更改，除非有很多全表扫描语句，一般采用默认即可。

innodb_purge_batch_size

Version Introduced	5.5.4	
Command-Line Format	--innodb_purge_batch_size=#	
Config-File Format	innodb_purge_batch_size	
Variable Name	innodb_purge_batch_size	
Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values	
	Type	numeric
	Default	20
	Range	1-5000

解释： 当开启独立线程清除 undo 页时，表示一次删除多少个页。默认是 20，不支持动态修改，一般不需要更改。

innodb_purge_threads

Version Introduced	5.5.4	
Command-Line Format	--innodb_purge_threads=#	
Config-File Format	innodb_purge_threads	
Variable Name	innodb_purge_threads	
Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values	
	Type	numeric
	Default	0
	Range	0-1

解释： 是否开启开启独立线程清除 undo 页。默认是 0，不开启，不支持动态修改。

innodb_read_ahead_threshold

Command-Line Format	--innodb_read_ahead_threshold=#	
Config-File Format	innodb_read_ahead_threshold	
Option Sets Variable	Yes, <u>innodb_read_ahead_threshold</u>	
Variable Name	innodb_read_ahead_threshold	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	numeric
	Default	56
	Range	0-64

解释：当顺序读取 extent 块（包含 64 个 page）innodb_read_ahead_threshold 设置的 page 页数量时，触发一个异步读取请求，将下一个页提前读取到 buffer pool 中。默认为 56，不支持动态修改，一般采用默认即可。

innodb_read_io_threads

Command-Line Format	--innodb_read_io_threads=#	
Config-File Format	innodb_read_io_threads	
Option Sets Variable	Yes, <u>innodb_read_io_threads</u>	
Variable Name	innodb_read_io_threads	
Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values	
	Type	numeric
	Default	4

解释：控制后台线程处理数据页上的读请求。默认是 4，不支持动态修改。建议根据服务器的核数以及读写请求的比例加以调整。

innodb_write_io_threads

Command-Line Format	--innodb_write_io_threads=#	
Config-File Format	innodb_write_io_threads	

Option Sets Variable	Yes, <u>innodb_write_io_threads</u>	
Variable Name	innodb_write_io_threads	
Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values	
	Type	numeric
	Default	4

解释：控制后台线程处理数据页上的写请求。默认是 4，不支持动态修改。建议根据服务器的核数以及读写请求的比例加以调整。

innodb_replication_delay

Command-Line Format	--innodb_replication_delay=#	
Config-File Format	innodb_replication_delay	
Option Sets Variable	Yes, <u>innodb_replication_delay</u>	
Variable Name	innodb_replication_delay	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	numeric
	Default	0

解释：当 innodb_thread_concurrency 线程已满，slave 端复制线程的延迟时间(ms)，默认是 0，不延迟，可动态修改。

innodb_spin_wait_delay

Command-Line Format	--innodb_spin_wait_delay=#	
Config-File Format	innodb_spin_wait_delay	
Option Sets Variable	Yes, <u>innodb_spin_wait_delay</u>	
Variable Name	innodb_spin_wait_delay	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	

	Type	numeric
	Default	6
	Min Value	0

解释：控制自旋锁轮训时间间隔，默认是 6 秒，可动态修改，一般默认即可。

innodb_stats_method

Command-Line Format	-- innodb_stats_method=#	
Config-File Format	innodb_stats_method	
Option Sets Variable	Yes, innodb_stats_method	
Variable Name	innodb_stats_method	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	enumeration
	Default	nulls_equal
	Valid Values	nulls_equal
		nulls_unequal nulls_ignored

解释：收集 InnoDB 表索引值分布统计时，服务器如何处理 NULL 值，该参数有 3 个可选值，nulls_equal=所有的 null 被看成相等的一个值，nulls_unequal=每个 null 被看成单独的一个值，nulls_ignored=null 被忽略，默认值为 nulls_equal。

innodb_stats_sample_pages

Command-Line Format	--innodb_stats_sample_pages=#	
Config-File Format	innodb_stats_sample_pages	
Option Sets Variable	Yes, innodb_stats_sample_pages	
Variable Name	innodb_stats_sample_pages	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	

	Type	numeric
	Default	8

解释：每当查询 `information_schema` 元数据库里的表时，InnoDB 还会随机提取其他数据库每个表 8 个索引页的部分数据，来更新 `information_schema.STATISTICS` 表，来返回刚才你查询的结果。默认是 8 个页，可动态更改。如果你关闭了 `innodb_stats_on_metadata`，那么这个参数被忽略。

innodb_strict_mode

Command-Line Format	--innodb_strict_mode=#	
Config-File Format	innodb_strict_mode	
Option Sets Variable	Yes, <code>innodb_strict_mode</code>	
Variable Name	innodb_strict_mode	
Variable Scope	Both	
Dynamic Variable	Yes	
	Permitted Values	
	Type	boolean
	Default	OFF

解释：开启 InnoDB 严格检查模式，尤其采用了页数据压缩功能，建议开启。此功能是当 `CREATE TABLE`, `ALTER TABLE` and `CREATE INDEX` 语句时，如果写法有错误，不会有警告信息，而是直接抛出错误，好处是直接将问题扼杀在摇篮里。默认是关闭。

innodb_use_native_aio

Version Introduced	5.5.4	
Command-Line Format	--innodb_use_native_aio=#	
Config-File Format	innodb_use_native_aio	
Option Sets Variable	Yes, <code>innodb_use_native_aio</code>	
Variable Name	innodb_use_native_aio	
Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values	

	Type	boolean
	Default	ON

解释：选择是否启用 Linux 系统的异步 I/O，默认是 ON，开启的，此参数不支持动态修改。如果在启动时 InnoDB 检测到一个潜在的问题，例如不支持 AIO 在 tmpfs 文件系统上，会自动将其关闭，一般采用默认即可。

innodb_use_sys_malloc

Command-Line Format	--innodb_use_sys_malloc=#	
Config-File Format	innodb_use_sys_malloc	
Option Sets Variable	Yes, innodb_use_sys_malloc	
Variable Name	innodb_use_sys_malloc	
Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values	
	Type	boolean
	Default	ON

解释：使用 MySQL 自带的内存分配程序，还是使用当前部署的操作系统中现有的更高效的内存分配程序。默认设置值为 1，不支持动态更改，表示 InnoDB 使用操作系统的内存分配程序。

1.5 同步复制新参数汇总

relay_log_recovery

Command-Line Format	-- relay_log_recovery =#	
Config-File Format	relay_log_recovery	
Option Sets Variable	Yes, relay_log_recovery	
Variable Name	relay_log_recovery	
Variable Scope	Global	
Dynamic Variable	No	
	Permitted Values	
	Type	boolean

	Default	FALSE
--	---------	-------

解释：当 slave 从库宕机后，假如 Relay-Log 损坏了，导致一部分中继日志没有处理，则自动放弃所有未执行的 relay-log，并且重新从 MASTER 上获取日志，这样保证 relay-log 的完整。默认情况下该功能是关闭的，不支持动态修改，建议开启。

sync_relay_log

Command-Line Format	-- sync_relay_log =#	
Config-File Format	sync_relay_log	
Option Sets Variable	Yes, sync_relay_log	
Variable Name	sync_relay_log	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Platform Bit Size	32
	Type	numeric
	Default	0
	Range	0 .. 4294967295
	Permitted Values	
	Platform Bit Size	64
	Type	numeric
	Default	0
	Range	0 .. 18446744073709547520

解释：这个参数和 sync_binlog 是一样的，当设置为 1 时，Slave 的 IO 线程每次接收到 Master 发送过来的 binlog 日志都要写入系统缓冲区然后刷入到 relay log 中继日志里，这样是最安全的，因为在崩溃的时候，你最多会丢失一个事务，但会造成磁盘的大量 I/O。当设置为 0 时，并不是马上就刷入中继日志里，而且由操作系统决定何时来写入，虽然安全性降低，但减少了大量的磁盘 I/O 操作。这个值默认是 0，可动态修改，建议采用默认值。

sync_relay_log_info

Command-Line Format	-- sync_relay_log_info =#	
Config-File Format	sync_relay_log_info	
Option Sets Variable	Yes, sync_relay_log_info	
Variable Name	sync_relay_log_info	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Platform Bit Size	32
	Type	numeric
	Default	0
	Range	0 .. 4294967295
	Permitted Values	
	Platform Bit Size	64
	Type	numeric
	Default	0
	Range	0 .. 18446744073709547520

解释：这个参数和 sync_relay_log 参数一样，当设置为 1 时，Slave 的 IO 线程每次接收到 Master 发送过来的 binlog 日志都要写入系统缓冲区然后刷入到 relay-log.info 里，这样是最安全的，因为在崩溃的时候，你最多会丢失一个事务，但会造成磁盘的大量 I/O。当设置为 0 时，并不是马上就刷入 relay-log.info 里，而且由操作系统决定何时来写入，虽然安全性降低，但减少了大量的磁盘 I/O 操作。这个值默认是 0，可动态修改，建议采用默认值。

sync_master_info

Command-Line Format	-- sync_master_info =#	
Config-File Format	sync_master_info	
Option Sets Variable	Yes, sync_master_info	
Variable Name	sync_master_info	
Variable Scope	Global	
Dynamic Variable	Yes	

	Permitted Values	
	Platform Bit Size	32
	Type	numeric
	Default	0
	Range	0 .. 4294967295
	Permitted Values	
	Platform Bit Size	64
	Type	numeric
	Default	0
	Range	0 .. 18446744073709547520

解释：这个参数和 `sync_relay_log_info` 参数一样，当设置为 1 时，Slave 的 IO 线程每次接收到 Master 发送过来的 binlog 日志都要写入系统缓冲区然后刷入 `master.info` 里，这样是最安全的，因为在崩溃的时候，你最多会丢失一个事务，但会造成磁盘的大量 I/O。当设置为 0 时，并不是马上就刷入 `master.info` 里，而且由操作系统决定何时来写入，虽然安全性降低，但减少了大量的磁盘 I/O 操作。这个值默认是 0，可动态修改，建议采用默认值。

`rpl_semi_sync_master_enabled`

Variable Name	<code>rpl_semi_sync_master_enabled</code>	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	boolean
	Default	OFF

解释：表示是否开启半同步复制功能，默认是关闭的，采用异步复制，可动态修改。

rpl_semi_sync_master_timeout

Variable Name	rpl_semi_sync_master_enabled	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	boolean
	Default	OFF

解释：表示主库在某次事务中，如果等待时间超过 10 秒，那么则降级为异步复制模式，不再等待 SLAVE 从库。如果主库再次探测到，SLAVE 从库恢复了，则会自动再次回到半同步复制模式。默认为 10000 毫秒，等于 10 秒，这个参数动态可调。

rpl_semi_sync_master_trace_level

Variable Name	rpl_semi_sync_master_trace_level	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	numeric
	Default	32

解释：在 Master 上，开启半同步复制模式时的调试级别，默认是 32，可动态修改，一般采用默认值即可。

1 = general level (for example, time function failures)

16 = detail level (more verbose information)

32 = net wait level (more information about network waits)

64 = function level (information about function entry and exit)

rpl_semi_sync_master_wait_no_slave

Variable Name	rpl_semi_sync_master_wait_no_slave	
Variable Scope	Global	
Dynamic Variable	Yes	

	Permitted Values	
	Type	boolean
	Default	ON

解释：是否允许 master 每个事务提交后都要等待 slave 的接收确认信号。默认为 on，每一个事务都会等待。如果为 off，则 slave 追赶上后，也不会开启半同步复制模式，需要手工开启，可动态修改。

rpl_semi_sync_slave_enabled

Variable Name	rpl_semi_sync_slave_enabled	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	boolean
	Default	OFF

解释：表示在 slave 上已经是否开启半同步复制模式，默认是不开启，可动态修改。

rpl_semi_sync_slave_trace_level

Variable Name	rpl_semi_sync_slave_trace_level	
Variable Scope	Global	
Dynamic Variable	Yes	
	Permitted Values	
	Type	numeric
	Default	32

解释：在 Slave 上，开启半同步复制模式时的调试级别，默认是 32，可动态修改，一般采用默认值即可。

1 = general level (for example, time function failures)

16 = detail level (more verbose information)

32 = net wait level (more information about network waits)

64 = function level (information about function entry and exit)

1.6 SQL 语句写法的改变

1.6.1 delete 表连接语法改变

在 MySQL5.5 里，规范了 sql 语句表连接的写法。

```
delete user a,user2 b from user a join user2 b on a.id=b.id;
```

这样写在 MySQL5.1 里，可以正常执行，但在 MySQL5.5 里就报错，下面是一个测试：

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.1.59-log |
+-----+
1 row in set (0.01 sec)

mysql> delete user a,user2 b from user a join user2 b on a.id=b.id;
Query OK, 18 rows affected (0.05 sec)

mysql>
```

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.19    |
+-----+
1 row in set (0.03 sec)

mysql> delete user a,user2 b from user a join user2 b on a.id=b.id;
ERROR 1064 (42000): You have an error in your SQL syntax; check the m
he right syntax to use near 'a,user2 b from user a join user2 b on a.
mysql>
```

可以看到，执行时报错。那么规范后的写法是：

```
delete a,b from user a join user2 b on a.id=b.id;
```

```
mysql> select version();
+-----+
| version() |
+-----+
| 5.5.19    |
+-----+
1 row in set (0.03 sec)

mysql>
mysql> delete a,b from user a join user2 b on a.id=b.id;
Query OK, 18 rows affected (0.11 sec)

mysql>
```