Mysql 技术内幕：InnoDB 学习（1）

jask

2024-08-11

Mysql 技术内幕

各种存储引擎

InnoDB

InnoDB 通过使用多版本并发控制（MVCC）来获得高并发性，并且实现了 SQL 标准的 4 种隔离级别，默认为 REPEATABLE 级别。同时，使用一种被称为 next-key locking 的策略来避免幻读（phantom）现象的产生。除此之外，InnoDB 储存引擎还提供了插入缓冲（insert buffer）、二次写（double write）、自适应哈希索引（adaptive hash index）、预读（read ahead）等高性能和高可用的功能。

对于表中数据的存储，InnoDB 存储引擎采用了聚集（clustered）的方式，因此每张表的存储都是按主键的顺序进行存放。如果没有显式地在表定义时指定主键，InnoDB 存储引擎会为每一行生成一个 6 字节的 ROWID，并以此作为主键。

MyIsam

MyISAM 存储引擎不支持事务、表锁设计，支持全文索引，主要面向一些 OLAP 数据库应用。在 MySQL 5.5.8 版本之前 MyISAM 存储引擎是默认的存储引擎（除 Windows 版本外）

NDB

NDB 存储引擎是一个集群存储引擎，类似于 Oracle 的 RAC 集群，不过与 Oracle RAC share everything 架构不同的是，其结构是 share nothing 的集群架构，因此能提供更高的可用性。NDB 的特点是数据全部放在内存中（从 MySQL 5.1 版本开始，可以将非索引数据放在磁盘上），因此主键查找（primary key lookups）的速度极快，并且通过添加 NDB 数据存储节点（Data Node）可以线性地提高数据库性能，是高可用、高性能的集群系统。

Memory

Memory 存储引擎（之前称 HEAP 存储引擎）将表中的数据存放在内存中，如果数据库重启或发生崩溃，表中的数据都将消失。它非常适合用于存储临时数据的临时表，以及数据仓库中的纬度表。Memory 存储引擎默认使用哈希索引，而不是我们熟悉的 B+ 树索引。

InnoDB 架构

后台线程

Master Thread, IO Thread, Purge Thread, Page Cleaner Thread.

内存

缓冲池 缓冲池简单来说就是一块内存区域，通过内存的速度来弥补磁盘速度较慢对数据库性能的影响。在数据库中进行读取页的操作，首先将从磁盘读到的页存放在缓冲池中，这个过程称为将页“FIX”在缓冲池中。下一次再读相同的页时，首先判断该页是否在缓冲池中。若在缓冲池中，称该页在缓冲池中被命中，直接读取该页。否则，读取磁盘上的页。

对于数据库中页的修改操作，则首先修改在缓冲池中的页，然后再以一定的频率刷新到磁盘上。这里需要注意的是，页从缓冲池刷新回磁盘的操作并不是在每次页发生更新时触发，而是通过一种称为 Checkpoint 的机制刷新回磁盘。同样，这也是为了提高数据库的整体性能。

具体来看，缓冲池中缓存的数据页类型有：索引页、数据页、undo 页、插入缓冲（insert buffer）、自适应哈希索引（adaptive hash index）、InnoDB 存储的锁信息（lock info）、数据字典信息（data dictionary）等。不能简单地认为，缓冲池只是缓存索引页和数据页，它们只是占缓冲池很大的一部分而已

LRU List, Free List 和 Flush List 通常来说，数据库中的缓冲池是通过 LRU（Latest Recent Used，最近最少使用）算法来进行管理的。即最频繁使用的页在 LRU 列表的前端，而最少使用的页在 LRU 列表的尾端。当缓冲池不能存放新读取到的页时，将首先释放 LRU 列表中尾端的页。

在 InnoDB 存储引擎中，缓冲池中页的大小默认为 16KB，同样使用 LRU 算法对缓冲池进行管理。稍有不同的是 InnoDB 存储引擎对传统的 LRU 算法做了一些优化。在 InnoDB 的存储引擎中，LRU 列表中还加入了 midpoint 位置。新读取到的页，虽然是最新访问的页，但并不是直接放入到 LRU 列表的首部，而是放入到 LRU 列表的 midpoint 位置。这个算法在 InnoDB 存储引擎下称为 midpoint insertion strategy。在默认配置下，该位置在 LRU 列表长度的 5/8 处。

为什么不采用朴素的 LRU 算法？这是因为若直接将读取到的页放入到 LRU 的首部，那么某些 SQL 操作可能会使缓冲池中的页被刷新出，从而影响缓冲池的效率。常见的这类操作为索引或数据的扫描操作。这类操作需要访问表中的许多页，甚至是全部的页，而这些页通常来说又仅在这次查询操作中需要，并不是活跃的热点数据。如果页被放入 LRU 列表的首部，那么非常可能将所需要的热点数据页从 LRU 列表中移除，而在下一次需要读取该页时，InnoDB 存储引擎需要再次访问磁盘。

LRU 列表用来管理已经读取的页，但当数据库刚启动时，LRU 列表是空的，即没有任何的页。这时页都存放在 Free 列表中。当需要从缓冲池中分页时，首先从 Free 列表中查找是否有可用的空闲页，若有则将该页从 Free 列表中删除，放入到 LRU 列表中。否则，根据 LRU 算法，淘汰 LRU 列表末尾的页，将该内存空间分配给新的页。当页从 LRU 列表的 old 部分加入到 new 部分时，称此时发生的操作为 page made young，而因为 innodb_old_blocks_time 的设置而导致页没有从 old 部分移动到 new 部分的操作称为 page not made young。可以通过命令 SHOW ENGINE INNODB STATUS 来观察 LRU 列表及 Free 列表的使用情况和运行状态。

重做日志缓冲 在通常情况下，8MB 的重做日志缓冲池足以满足绝大部分的应用，因为重做日志在下列三种情况下会将重做日志缓冲中的内容刷新到外部磁盘的重做日志文件中。

Master Thread 每一秒将重做日志缓冲刷新到重做日志文件；

每个事务提交时会将重做日志缓冲刷新到重做日志文件；

当重做日志缓冲池剩余空间小于 1/2 时，重做日志缓冲刷新到重做日志文件。

额外的内存池 额外的内存池通常被 DBA 忽略，他们认为该值并不十分重要，事实恰恰相反，该值同样十分重要。在 InnoDB 存储引擎中，对内存的管理是通过一种称为内存堆（heap）的方式进行的。在对一些数据结构本身的内存进行分配时，需要从额外的内存池中进行申请，当该区域的内存不够时，会从缓冲池中进行申请。例如，分配了缓冲池（innodb_buffer_pool），但是每个缓冲池中的帧缓冲（frame buffer）还有对应的缓冲控制对象（buffer control block），这些对象记录了一些诸如 LRU、锁、等待等信息，而这个对象的内存需要从额外内存池中申请。因此，在申请了很大的 InnoDB 缓冲池时，也应考虑相应地增加这个值。

Checkpoint

目的：缩短数据库的恢复时间；

缓冲池不够用时，将脏页刷新到磁盘；

重做日志不可用时，刷新脏页。

InnoDB 有两种 Checkpoint： 1.Sharp Checkpoint 2.Fuzzy Checkpoint

Sharp Checkpoint 发生在数据库关闭时将所有的脏页都刷新回磁盘，这是默认的工作方式。但是若数据库在运行时也使用 Sharp Checkpoint，那么数据库的可用性就会受到很大的影响。故在 InnoDB 存储引擎内部使用 Fuzzy Checkpoint 进行页的刷新，即只刷新一部分脏页，而不是刷新所有的脏页回磁盘。

在 InnoDB 存储引擎中可能发生如下几种情况的 Fuzzy Checkpoint：

Master Thread Checkpoint

FLUSH_LRU_LIST Checkpoint

Async/Sync Flush Checkpoint

Dirty Page too much Checkpoint

Master Thread 工作方式

1.0x 版本之前的 Master Thread Master Thread 具有最高的线程优先级别。其内部由多个循环（loop）组成：主循环（loop）、后台循环（backgroup loop）、刷新循环（flush loop）、暂停循环（suspend loop）。

Master Thread 会根据数据库运行的状态在 loop、background loop、flush loop 和 suspend loop 中进行切换。Loop 被称为主循环，因为大多数的操作是在这个循环中，其中有两大部分的操作——每秒钟的操作和每 10 秒的操作。

loop 循环通过 thread sleep 来实现，这意味着所谓的每秒一次或每 10 秒一次的操作是不精确的。在负载很大的情况下可能会有延迟（delay），只能说大概在这个频率下。当然，InnoDB 源代码中还通过了其他的方法来尽量保证这个频率。

每秒一次的操作包括：

日志缓冲刷新到磁盘，即使这个事务还没有提交（总是）；

合并插入缓冲（可能）；

至多刷新 100 个 InnoDB 的缓冲池中的脏页到磁盘（可能）；

如果当前没有用户活动，则切换到 background loop（可能）。

即使某个事务还没有提交，InnoDB 存储引擎仍然每秒会将重做日志缓冲中的内容刷新到重做日志文件。这一点是必须要知道的，因为这可以很好地解释为什么再大的事务提交（commit）的时间也是很短的。

接着来看每 10 秒的操作，包括如下内容：

刷新 100 个脏页到磁盘（可能的情况下）；

合并至多 5 个插入缓冲（总是）；

将日志缓冲刷新到磁盘（总是）；

删除无用的 Undo 页（总是）；

刷新 100 个或者 10 个脏页到磁盘（总是）。

在以上的过程中，InnoDB 存储引擎会先判断过去 10 秒之内磁盘的 IO 操作是否小于 200 次，如果是，InnoDB 存储引擎认为当前有足够的磁盘 IO 操作能力，因此将 100 个脏页刷新到磁盘。接着，InnoDB 存储引擎会合并插入缓冲。不同于每秒一次操作时可能发生的合并插入缓冲操作，这次的合并插入缓冲操作总会在这个阶段进行。之后，InnoDB 存储引擎会再进行一次将日志缓冲刷新到磁盘的操作。这和每秒一次时发生的操作是一样的。

接着 InnoDB 存储引擎会进行一步执行 full purge 操作，即删除无用的 Undo 页。对表进行 update、delete 这类操作时，原先的行被标记为删除，但是因为一致性读（consistent read）的关系，需要保留这些行版本的信息。但是在 full purge 过程中，InnoDB 存储引擎会判断当前事务系统中已被删除的行是否可以删除，比如有时候可能还有查询操作需要读取之前版本的 undo 信息，如果可以删除，InnoDB 会立即将其删除。从源代码中可以发现，InnoDB 存储引擎在执行 full purge 操作时，每次最多尝试回收 20 个 undo 页。

然后，InnoDB 存储引擎会判断缓冲池中脏页的比例（buf_get_modified_ratio_pct），如果有超过 70% 的脏页，则刷新 100 个脏页到磁盘，如果脏页的比例小于 70%，则只需刷新 10% 的脏页到磁盘。

接着来看 background loop，若当前没有用户活动（数据库空闲时）或者数据库关闭（shutdown），就会切换到这个循环。background loop 会执行以下操作：删除无用的 Undo 页（总是）；

合并 20 个插入缓冲（总是）；

跳回到主循环（总是）；

不断刷新 100 个页直到符合条件（可能，跳转到 flush loop 中完成）。

若 flush loop 中也没有什么事情可以做了，InnoDB 存储引擎会切换到 suspend_loop，将 Master Thread 挂起，等待事件的发生。若用户启用（enable）了 InnoDB 存储引擎，却没有使用任何 InnoDB 存储引擎的表，那么 Master Thread 总是处于挂起的状态。

1.2x 之前的 Master Thread 改善了 IO 性能

1.2x 版本的 Master Thread 伪代码如下：

```
if InnoDB is idle

srv_master_do_idle_tasks();

else

srv_master_do_active_tasks();
```

其中 srv_master_do_idle_tasks() 就是之前版本中每 10 秒的操作，srv_master_do_active_tasks() 处理的是之前每秒中的操作。同时对于刷新脏页的操作，从 Master Thread 线程分离到一个单独的 Page Cleaner Thread，从而减轻了 Master Thread 的工作，同时进一步提高了系统的并发性。

InnoDB 关键特性

插入缓冲（Insert Buffer）

两次写（Double Write）

自适应哈希索引（Adaptive Hash Index）

异步 IO（Async IO）

刷新邻接页（Flush Neighbor Page）

插入缓冲 Insert Buffer 是物理页的一个组成部分。其中 a 列是自增长的，若对 a 列插入 NULL 值，则由于其具有 AUTO_INCREMENT 属性，其值会自动增长。同时页中的行记录按 a 的值进行顺序存放。在一般情况下，不需要随机读取另一个页中的记录。因此，对于这类情况下的插入操作，速度是非常快的。

在 InnoDB 存储引擎中，主键是行唯一的标识符。通常应用程序中行记录的插入顺序是按照主键递增的顺序进行插入的。因此，插入聚集索引（Primary Key）一般是顺序的，不需要磁盘的随机读取。

注意 并不是所有的主键插入都是顺序的。若主键类是 UUID 这样的类，那么插入和辅助索引一样，同样是随机的。即使主键是自增类型，但是插入的是指定的值，而不是 NULL 值，那么同样可能导致插入并非连续的情况。

InnoDB 存储引擎开创性地设计了 Insert Buffer，对于非聚集索引的插入或更新操作，不是每一次直接插入到索引页中，而是先判断插入的非聚集索引页是否在缓冲池中，若在，则直接插入；若不在，则先放入到一个 Insert Buffer 对象中，好似欺骗。数据库这个非聚集的索引已经插到叶子节点，而实际并没有，只是存放在另一个位置。然后再以一定的频率和情况进行 Insert Buffer 和辅助索引页子节点的 merge（合并）操作，这时通常能将多个插入合并到一个操作中（因为在一个索引页中），这就大大提高了对于非聚集索引插入的性能。

使用 Insert Buffer 需要满足两个条件：

索引是辅助索引；

索引不是唯一的；

Change Buffer InnoDB 存储引擎可以对 DML 操作——INSERT、DELETE、UPDATE 都进行缓冲，他们分别是：Insert Buffer、Delete Buffer、Purge buffer。

对一条记录进行 UPDATE 操作可能分为两个过程：

将记录标记为已删除；

真正将记录删除。

因此 Delete Buffer 对应 UPDATE 操作的第一个过程，即将记录标记为删除。Purge Buffer 对应 UPDATE 操作的第二个过程，即将记录真正的删除。同时，InnoDB 存储引擎提供了参数 innodb_change_buffering，用来开启各种 Buffer 的选项。该参数可选的值为：inserts、deletes、purges、changes、all、none。inserts、deletes、purges 就是前面讨论过的三种情况。changes 表示启用 inserts 和 deletes，all 表示启用所有，none 表示都不启用。该参数默认值为 all。

Insert Buffer 的内部实现 Insert Buffer 的数据结构是一棵 B+ 树。因此其也由叶节点和非叶节点组成。非叶节点存放的是查询的 search key（键值）。

search key 一共占用 9 个字节，其中 space 表示待插入记录所在表的表空间 id，在 InnoDB 存储引擎中，每个表有一个唯一的 space id，可以通过 space id 查询得知是哪张表。space 占用 4 字节。marker 占用 1 字节，它是用来兼容老版本的 Insert Buffer。offset 表示页所在的偏移量，占用 4 字节。

space | marker | offset |

非叶节点的 search key 当一个辅助索引要插入到页（space，offset）时，如果这个页不在缓冲池中，那么 InnoDB 存储引擎首先根据上述规则构造一个 search key，接下来查询 Insert Buffer 这棵 B+ 树，然后再将这条记录插入到 Insert Buffer B+ 树的叶子节点中。

Merge Insert Buffer Merge Insert Buffer 的操作可能发生在以下几种情况下：辅助索引页被读取到缓冲池时；

Insert Buffer Bitmap 页追踪到该辅助索引页已无可用空间时；

Master Thread。

两次写

Insert Buffer 带给 InnoDB 存储引擎的是性能上的提升，那么 doublewrite（两次写）带给 InnoDB 存储引擎的是数据页的可靠性。

当发生数据库宕机时，可能 InnoDB 存储引擎正在写入某个页到表中，而这个页只写了一部分，比如 16KB 的页，只写了前 4KB，之后就发生了宕机，这种情况被称为部分写失效（partial page write）。在 InnoDB 存储引擎未使用 doublewrite 技术前，曾经出现过因为部分写失效而导致数据丢失的情况。

doublewrite 由两部分组成，一部分是内存中的 doublewrite buffer，大小为 2MB，另一部分是物理磁盘上共享表空间中连续的 128 个页，即 2 个区（extent），大小同样为 2MB。在对缓冲池的脏页进行刷新时，并不直接写磁盘，而是会通过 memcpy 函数将脏页先复制到内存中的 doublewrite buffer，之后通过 doublewrite buffer 再分两次，每次 1MB 顺序地写入共享表空间的物理磁盘上，然后马上调用 fsync 函数，同步磁盘，避免缓冲写带来的问题。在这个过程中，因为 doublewrite 页是连续的，因此这个过程是顺序写的，开销并不是很大。在完成 doublewrite 页的写入后，再将 doublewrite buffer 中的页写入各个表空间文件中，此时的写入则是离散的。

自适应哈希索引

InnoDB 存储引擎会监控对表上各索引页的查询。如果观察到建立哈希索引可以带来速度提升，则建立哈希索引，称之为自适应哈希索引（Adaptive Hash Index，AHI）。AHI 是通过缓冲池的 B+ 树页构造而来，因此建立的速度很快，而且不需要对整张表构建哈希索引。InnoDB 存储引擎会自动根据访问的频率和模式来自动地为某些热点页建立哈希索引。

AHI 有一个要求，即对这个页的连续访问模式必须是一样的。例如对于（a，b）这样的联合索引页，其访问模式可以是以下情况：

WHERE a=xxx

WHERE a=xxx and b=xxx

异步 IO

当前的数据库系统都采用异步 IO（Asynchronous IO，AIO）的方式来处理磁盘操作。

AIO 的另一个优势是可以进行 IO Merge 操作，也就是将多个 IO 合并为 1 个 IO，这样可以提高 IOPS 的性能。、

刷新邻接页

AIO 的另一个优势是可以进行 IO Merge 操作，也就是将多个 IO 合并为 1 个 IO，这样可以提高 IOPS 的性能。