# 高性能MySQL笔记

[数据库索引原理及优化](https://blog.csdn.net/suifeng3051/article/details/52669644)

### MVCC

[MVCC机制详解](https://www.codercto.com/a/88775.html)

[必看MVCC解析](https://segmentfault.com/a/1190000012650596#item-1)

[辅助视频理解](https://www.bilibili.com/video/av83518115?from=search&seid=12695218876063854516)

为MVCC是行级锁的一个变种，但是它在很多情况下避免了加锁操作，因此开 销更低。虽然实现机制有所不同，但大都实现了非阻塞的读操作，写操作也只锁定必要的行。

MVCC的实现,是通过保存数据在某个时间点的快照来实现的。也就是说，不管需要执行多长时间，每个事务看到的数据都是一致的。根据事务开始的时间不同，每个事务对同一张表，同一时刻看到的数据可能是不一样的

（行创建时间可以认为是最近一次事务id指针）

（行删除时间可以认为是回归指针，指向上一个回滚版本）

**SELECT**

InnoDB会根据以下两个条件检查每行记录:

* InnoDB只查找版本早于当前事务版本的数据行(也就是,行的系统版本号小于或等于事务的系统版本号),这样可以确保事务读取的行，要么是在事务开始前已经存在的，要么是事务自身插入或者修改过的。
* 行的删除版本要么未定义（*没有回滚记录）*，要么大于当前事务版本号（）。这可以确保事务读取到的行，在事务开始之前未被删除。

**INSERT**

* InnoDB为新播入的每-一行保存当前系统版本号作为行版本号。

**DELETE**

* InnoDB为删除的每-\*行保存当前系统版本号作为行删除标识。

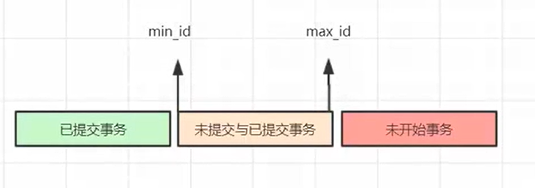
**UPDATE**

* InnoDB为插入-行新记录，保存当前系统版本号作为行版本号，同时保存当前系统版本号到原来的行作为行删除标识。

上述存疑

**InnoDB存储引擎在数据库每行数据的后面添加了三个字段**

* 6字节的事务ID(DB\_TRX\_ID)字段: 用来标识**最近一次**对本行记录做修改(insert|update)的事务的标识符, 即**最后一次修改(insert|update)本行记录的事务id**。 至于delete操作，在innodb看来也不过是一次update操作，更新行中的一个特殊位将行表示为deleted, **并非真正删除**。
* 7字节的回滚指针(DB\_ROLL\_PTR)字段: 指写入回滚段(rollback segment)的 undo log record (撤销日志记录记录)。 如果一行记录被更新, 则 undo log record 包含 '重建该行记录被更新之前内容' 所必须的信息。
* 6字节的DB\_ROW\_ID字段: 包含一个随着新行插入而单调递增的行ID, 当由innodb自动产生聚集索引时，聚集索引会包括这个行ID的值，否则这个行ID不会出现在任何索引中。 结合聚簇索引的相关知识点, 我的理解是, 如果我们的表中没有主键或合适的唯一索引, 也就是无法生成聚簇索引的时候, InnoDB会帮我们自动生成聚集索引, 但聚簇索引会使用DB*ROW*ID的值来作为主键; 如果我们有自己的主键或者合适的唯一索引, 那么聚簇索引中也就不会包含 DB*ROW*ID 了



当执行sql语句的时候会生成一致性试图read-view，他由执行查询时所有**未提交的事务id**数组（数组里最小的id位min\_id,已创建的最大事务id为max\_id（当前事务的稳定id）组成，查询的数据结果需要跟read-view做对比从而得到快照结果·

*trx\_id就是遍历版本链的当前事务id，只要匹配到一条可见的事务id就可以返回*

版本链对比规则：

1. 如果落在绿色部分（trx*id < min*id) 表示这个版本是已提交的事务生成，这个数据是可见的
2. 如果落在红色部分（trx*id > max*id) 表示这个版本是由将来启动的事务生成的，是肯定不可见的
3. 如果时落在黄色不跟（min*id <= trx*id <= max\_id)那就包括两种情况
   * 如果row的trx\_id在数组中，表示这个版本室友还没提交的事务生成的，不可见，当前自己的事务时可见的
   * 若row的rtx\_id不再数组中，表示这个版本是已经提交了的事务生成的，可见。

对于删除的情况可以认为是update的特殊情况，会将版本链上最新的数据复制一份，然后将trx*id修改成修改成删除操作的trx*id，同时在该条记录的头信息（record header)里的（deleted flag）标记位写上true，来表示当前记录已经被删除，在查询时按照上面的规则查到对应的记录如果delete flag标记为true，意味着记录已被删除，则不返回数据

**总结**

事务执行更新操作的时候会更新**undo log**

执行查找操作的时候会新建一致性视图（RR级别下是第一次select，rc下是每次select）

当查找的时候通过回滚指针遍历历史版本，找到一条可见操作记录就按照此操作查询

**一个事务快照的创建过程可以概括为：** 查看当前所有的未提交并活跃的事务，存储在数组中 选取未提交并活跃的事务中最小的XID，记录在快照的xmin中 **选取所有已提交事务中最大的XID，加1后记录在xmax中**

## 创建高性能索引

基础

Mysql只能高效地使用最左前缀列

### 索引类型

索引是在存储层实现的，不是在服务器层

#### B-Tree索引

默认索引，除了Archive不支持之外

不同引擎使用的存储结构不同，NDB集群存储引擎使用T-Tree结构，innoDB使用B+Tree结构

不同引擎使用性能也不同，MyISAM使用前缀压缩技术使索引更小，通过数据的物理位置引用被索引的列；InonDB按照原数据格式压缩存储，根据主键引用被索引的列

B-Tree表示值都是按顺序存储，每一个叶子到根的距离相同

查询从索引的根节点开始，根节点中存放的是指向叶子节点的指针

叶子节点的指针指向被索引数据

B-Tree索引适用于全键型、键值范围和键前缀查找中的最左前缀查找

* 全值匹配
* 与所有列进行匹配
* 最左前缀匹配
* 只使用索引的第一列
* 列前缀匹配
* 某列的开头部分
* 范围值匹配
* 精确匹配某一列并范围匹配另外一列
* 只访问索引的查询
* 即覆盖查询
* 顺序查询，比如Order By

使用限制

* 非最左匹配查询
* 不能跳过索引的列
* 如果存在某列的范围查询，右边所有列都无法使用索引优化查询

#### 哈希索引

基于hash表实现，必须精确匹配才有效

只有memory引擎显式支持且默认支持

分为Slot槽和Value值

查找方法：计算hash值->找到指向指针->查找数据行

限制

* 只包含hash值和指针，不存储字段值，所以必须读取数据行。不过内存访问行速度较快，可以不考虑
* 索引数据不是按照索引值顺序存储，无法应用于排序
* 不支持部分匹配索引
* 只支持等值查询
* 存在哈希冲突，即不同索引列值具有相同的哈希值，当存在哈希冲突的时候必须遍历链表中的所有行指针，逐行比较
* 索引维护代价随着哈希冲突的增加越来越大

InnDB存在自适应哈希索引，当判断出查找频繁的索引值时，在B-Tree的基础上再创建一个Hash索引

可以创建自定义hash索引，进行伪hash索引，针对索引值较大时，但是必须保证索引值的唯一性

#### R-Tree空间索引

MyISAM引擎支持空间索引，用作地理位置存储

#### 全文索引

查找文本中的关键字

...

### 索引的优点

* 减少服务器扫描数据量
* 帮助服务器避免排序和临时表
* 将随机IO变成3顺序IO

三星原则

* 索引是否将相关记录放在一起
* 索引中的数据顺序和排序顺序一致
* 包含查询中需要的全部列

### 高性能索引策略

#### 使用独立的列

将索引列单独放在比较符号的一侧

#### 前缀索引

使用前缀索引索引很长的字符串，能让索引更小更快但是Mysql无法使用前缀索引做ORDER BY和GROUP BY，也无法使用前缀索引做覆盖扫描

**索引的选择性**：不重复的索引值和数据表的记录总数的比值；唯一索引的选择性是1

尽可能要让前缀索引的选择性和全值素银的选择性相匹配

#### 多列索引

出现合并索引（Using Union）意味着索引建立糟糕

* 多个AND条件可以使用一个包含所有列的多列索引代替
* 多个OR条件会消耗大量CPU和内存资源

#### 选择合适顺序

一般思路：

1. 将选择性最高的列放在索引最前列（不需要考虑排序或分组时）

* 计算选择性，选择性低的放在最前面

1. 注意具体情况，留心WHERE字句中的排序、分组和范围条件等因素对查询的影响

#### 聚簇索引

聚簇索引是一种数据存储方式，不是索引类型

InnoDB中竖数据行存放在索引的叶子页中，聚簇即数据行和相邻键值紧凑存储在一起

优点

* 数据访问更快
* 使用覆盖索引扫扫描查询时可以直接使用主键值
* 数据相关保存

缺点

* 数据放在内存就没有优势了
* 插入速度依赖于插入顺序，按照主键插入最快
* 更新聚簇索引列代价较高
* 更新时存在列分裂的问题，即当页面插满的时候会将该页分裂容下该行，从而占用更多磁盘空间
* 聚簇索引可能导致全表扫描变慢，尤其是行比较稀疏，或者由于页分裂导致数据存储不连续的时候
* 二级索引（非聚簇索引）很大
* 二级索引访问需要两次索引查找，因为二级索引节点保存的是行主键值而不是物理位置指向的指针

#### InnoDB和MyISAM的数据分布

* MyISAM更加简单，二级索引存放的是指针；InnoDB存放的索引的主键值

#### 覆盖索引

* 只读取索引即可以获取数据行，减少数据访问量；也更容易放入内存中
* 索引是顺序存储的，适合IO密集型
* 聚簇索引+覆盖索引比较高效

无法执行覆盖索引的原因

* 索引必须存索引列的值，mysql中一般只用B-Tree
* 没有任何索引能覆盖这个查询
* 不能使用LIKE操作

改良方法

* 使用延迟关联

#### 索引扫描作为排序

explain中type值为"index"说明使用索引扫描

当索引的列顺序和ORDERBY子句顺序一致并且所以列的排序方向一样时，mysql才能使用索引做结果排序

#### 压缩（前缀压缩）索引

MyISAM可以压缩索引大小

方法

* 先保存索引块的第一个值
* 其他值和第一个值比较得到相同前缀的字节数和剩余的不通过后缀部分

优势

* 占用更小的空间

缺点

* 无法使用二分查找等，有些操作如倒序操作速度不行
* 不适合CPU密集型应用

#### 冗余和重复索引

杜绝重复索引

冗余索引出现情况

* 需要扩展索引又要保证原索引性能不被影响
* 注意索引越多维护成本越高

#### 索引和锁

索引可以让査询锁定更少的行。

InnoDB在二级索引上使用共享(读）锁，但访问主键索引需要排他（写）锁。这消除了使用覆盖索引的可能性，并且使得SELECT FOR UPDATE比LOCK IN SHARE MODE或非锁定査询要慢很多。

### 索引案例

#### 支持多种过滤条件

* 字段选择少的放在前面
* 范围查找的字段放在后面

#### 避免多个范围查询

范围列后面无法索引

可以尝试将一个范围列变成等值列进行比较

#### 优化排序

* 使用延迟关联，通使用覆盖索引查询返回需要的主键，再根据这些主键关联原表获得需要的行

### 维护索引和表

维护目的

* 找到并修复损坏的表
* 维护准确而的索引统计信息
* 较少碎片

#### 更新索引统计信息

MySQL的查询优化器有两个API

* records*in*range() InnoDB传入两个边界值得到估算值
* info() 返回个各种类型数据

MySQL优化器使用的是基于成本的模型，而衡量成本的主要指标就是一个查询需要扫描多少行

**ANALYZE TABLE：**重新生成统计信息

#### 减少索引和数据碎片

**行碎片**

这种碎片指的是数据行被存储为多个地方的多个片段中。即使査询只从索引中访问一行记录，行碎片也会导致性能下降。

**行间碎片**

行间碎片是指逻辑上顺序的页，或者行在磁盘上不是顺序存储的。行间碎片对诸如全表扫描和聚簇索引扫描之类的操作有很大的影响，因为这些操作原本能够从磁盘上顺序存储的数据中获益。

**剩余空间碎片**

剩余空间碎片是指数据页中有大量的空余空间。这会导致服务器读取大量不需要的数据，从而造成浪费。

MylSAM表，这三类碎片化都可能发生。但`InnoDB不会出现短小的行碎片；InnoDB会移动短小的行并重写到一个片段中。

解决方法

* OPTIMIZE TABLE或者导出再导入的方式来重新整理数据
* MylSAM引擎以通过排序算法重建索引的方式来消 除碎片
* InnoDB可以先删除，然后再重新创建索引的方式来消除索引的碎片化。
* mysql> ALTER TABLE <table> ENGINE=<efigiwe>;

总结

## 查询性能优化

### 为什么查询速度会变慢

查询的生命周期

查询的时间花费

### 优化数据访问

査询性能低下最基本的原因是访问的数据太多

1. 确认应用程序是否在检索大量超过需要的数据。这通常意味着访问了太多的行，但有时候也可能是访问了太多的列。
2. 确认MySQL服务器层是否在分析大量超过需要的数据行。

举例

* 查询不需要记录
* 使用LIMIT限制查询数量
* 多表关联时返回全部列
* 总是取出全部列
* 重复查询相同的数据

#### 查看是否扫描额外的记录

影响开销的三指标

* 响应时间
* 扫描行数
* 返回行数

**响应时间**

由服务时间和排队时间组成

**扫描行数**

理想情况应该和返回行数相同

访问类型

* 全表扫描(最慢)
* 索引扫描
* 范围扫描
* 唯一索引查询
* 常数引用（最快）

mysql使用where的方式

* 在索引中使用where条件来过滤不匹配的记录。
* 使用索引覆盖扫描返回记录
* 从数据表中返回数据，然后过滤不满足条件的记录

解决方案

* 使用索引覆盖扫描
* 改变库表结构
* 重写查询

### 重构查询方式

#### 切分查询

按时清理数据

#### 分解关联查询

* 缓存效率更高
* 执行单个查询时减少锁的竞争
* 数据库的高性能和可扩展

### 查询执行

#### 查询执行的过程

1. 客户端发送一条査询给服务器。
2. 服务器先检査査询缓存，如果命中了缓存，则立刻返回存储在缓存中的结果。否则进入下一阶段。
3. 服务器端进行SQL解析、预处理，再由优化器生成对应的执行计划。
4. MySQL根据优化器生成的执行计划，调用存储引擎的API来执行査询。
5. 将结果返回给客户端。

#### MySQL客户端/服务器通信协议

半双工，优点简单快速，缺点数据包越长性能越低

**查询状态**

**Sleep** 线程正在等待客户端发送新的请求。 **Query** 线程正在执行查询或者正在将结果发送给客户端。 **Locked** 在MySQL服务器层，该线程正在等待表锁。在存储引擎级别实现的锁，例如InnoDB的行锁，并不会体现在线程状态中。对于MylSAM来说这是一个比较典型的状态，但在其他没有行锁的引擎中也经常会出现。 **Analyzing and statistics** 线程正在收集存储引擎的统计信息，并生成査询的执行计划。 **Copying to tmp table [on disk]** 线程正在执行査询，并且将其结果集都复制到一个临时表中，这种状态一般要么是在做GROUP BY操作，要么是文件排序操作，或者是UNION操作。如果这个状态后面还有 “on disk”标记，那表示MySQL正在将一个内存临时表放到磁盘上。 **Sorting result** 线程正在对结果集进行排序。

**Sending data** 这表示多种情况：线程可能在多个状态之间传送数据，或者在生成结果集，或者在向客户端返回数据。

#### 查询优化处理

解析SQL、预处理、优化SQL执行计划

**语法解析器和预处理**

通过关键字解析SQL生成一棵解析树，使用mysql的语法规则验证和解析查询

预处理器根据MySQL规则进一步检查解析树是否合法，下一步还会验证权限

**查询优化器**

优化器将语法树转化成执行计划，其作用就是找到最合适的执行计划

MySQL使用基于成本的优化，使用了复杂的优化策略

优化策略

* 静态优化：直接对解析树分析，完成优化
* 动态优化：将WHERE转化 ，运行时优化，和查询的上下文有关

MySQL对查询的静态优化只做一次，但是对动态优化则每次执行时都需要重新评估

**能处理的优化类型**

重新定义关联表的顺序

外连接转化成内连接

使用等价变换原则

优化COUNT()、MIN()、MAX()

预估并转化成常数表达式

覆盖索引扫描

子查询优化

提前终止查询

等值传播

列表IN()的比较

....

**数据和索引的统计信息**

统计信息由存储引擎实现，MySQL査询优化器在生成査询的执行计划时，需要向存储引擎获取相应的统计信息

**执行计划**

mysql是一棵左侧深度优先的树而不是平衡树

**关联查询优化器**

MySQL会选择合适的关联顺序让查询成本京可能降低

但是等关联表的个数增加的时候，执行计划的搜索空间(需要检查的阶乘种关联顺序)增长迅速，此时优化器就会使用贪婪模式寻找最优关联顺序

**排序优化**

索引排序不起作用的时候MySQL需要自己进行排序，如果数据量小则在内存中进行，如果数据量大则需要使用磁盘，这个过程统一称为文件排序(**filesort**)

排序算法

* 两次传输排序
* 读取行指针和需要排序的字段，对其进行排序，然后再根据排序结果读取所需要的数据行。
* 单次传输排序
* 先读取査询所需要的所有列，然后再根据给定列进行排序，最后直接返回排序结果

#### 查询执行引擎

查询优化阶段每一个表创建了一个handler实例，优化器调用实例接口（`handler API）获取表相关信息

#### 返回结果

* 即使不需要返回结果集，mysql任然会返沪查询的一些信息
* 查询如果可以别缓存，mysql也会在这个阶段将结果存放到查询缓存中
* 返回是一个增量、逐步返回的过程，好处是服务器无需存储大量洁后果，也不会因为要返回太多接轨而消耗大量内存；而且，客户端也呢个第一时间获得返回的结果

### 查询优化器的局限性

**关联子查询**

MySQL会将相关的外层表压到子査询中，它认为这样可以更高效率地查找到数据行

优化

* 使用函数GR0UP\_C0NCAT()在IN()中构造一个由逗号分隔的列表(性能不佳)
* 等效改写

一般使用左外连接比子查询稍快

去重时会产生临时中间表，此时子查询比关联更快

具体问题具体分析

#### UNION限制

有时，MySQL无法将限制条件从外层“下推”到内层，这使得原本能够限制部分返回结果的条件无法应用到内层査询的优化上。

#### 索引合并优化

当WHERE子句中包含多个复杂条件的 时候，MySQL能够访问单个表的多个索引以合并和交叉过滤的方式来定位需要査找的行

#### 等值传递

例如，有一个非常大的IN()列表，而MySQL优化器发现存在WHERE、ON或者USING的子句，将这个列表的值和另一个表的某个列相关联。那么优化器会将IN()列表都复制应用到关联的各个表中。

#### 并行执行

MySQL无法并行执行查询

#### 哈希关联

mysql不支持哈希关联，所有关联都是嵌套循环关联

#### 松散索引扫描

MySQL不支持松散索引扫描，无法按照不连续的方式扫描一个索引

#### 最大值和最小值优化

使用LIMIT重写查询，从而使得扫描记录数尽可能少

#### 在同一张表上查询更新

一般i情况MySQL不允许对哦同一张表同时进行查询和更新。

但是可以通过生成表的形式绕过限制

### 查询优化器的提示

略

### 优化特定类型的查询

#### 优化COUNT()查询

COUNT()：

* 它可以统计某个列值的数量，也可以统计行数。在统计列值时要求列值是非空的（不统计NULL)。如果在C0UNT()的括号中指定了列或者列的表达式，则统计的就是这个表达式有值的结果数
* 另一个作用是统计结果集的行数。当MySQL确认括号内的表达式值不可能为空时，实际上就是在统计行数

MylSAM的C0UNTO函数总是非常快，不过这是有前提条件的，即只有没有任何WHERE条件的COUNT(\*)才非常快

**优化策略**

* 使用近似值
* 索引覆盖扫描

#### 优化关联查询

* 确保ON或者USING子句中的列上有索引
* **确保任何的GROUP BY和ORDER BY中的表达式只涉及到一个表中的列，这样MySQL才有可能使用索引来优化这个过程。**

#### 优化GROUP BY和DISTINCT

无法使用索引的时候，Group BY使用两种策略完成

* 使用临时表
* 使用文件排序进行分组

在分组査询的SELECT中直接使用非分组列通常导致结果是不定的，当索引改变，或者优化器选择不同的优化策略时都可能导致结果不一样

如果没有通过ORDER BY子句显式地指定排序列，当査询使用GROUP BY子句的时候，**结果集会自动按照分组的字段进行排序**。如果不关心结果集的顺序，而这种默认排序又导致了需要文件排序，则可以使用ORDER BY NULL,让MySQL不再进行文件排序。也可以在GROUP BY子句中直接使用DESC或者ASC关键字，使分组的结果集按需要的方向排序

#### 优化GROUP BY WITH ROLLUP

尽可能将该功能转移到应用程序中

#### 优化LIMIT分页

* 尽可能地使用索引覆盖扫描
* 做一次关联操作再返回所需的列（关联延迟）
* 利用offset记录上次查询点直接向下查询

#### 优化 SQL*CALC*FOUND\_ROWS

分页的时候，另一个常用的技巧是在LIMFT语句中加上`SQL\_CALC\_FOUND\_ROWS提示(hint)，这样就可以获得去掉LIMIT以后满足条件的行数，因此可以作为分页的总数

另一种做法是先获取并缓存较多的数据，然后每次分页从缓存中获取

#### 优化UNION查询

否则就一定要使用UNION ALL,这一点很重要。如果没有ALL关键字，MySQL会给临时表加上DISTINCT选项，这会导致对整个临时表的数据做唯一性检査。这样做的代价非常高。

#### 静态查询分析

#### 使用用户自定义变量

不能使用的情况

* 使用自定义变量的査询，无法使用査询缓存。
* 不能在使用常量或者标识符的地方使用自定义变量,例如表名、列名和LIMIT子句中。
* 用户自定义变量的生命周期是在一个连接中有效，所以不能用它们来做连接间的通信。
* 如果使用连接池或者持久化连接，自定义变量可能让看起来毫无关系的代码发生交互（如果是这样，通常是代码bug或者连接池bug,这类情况确实可能发生）。
* 在5.0之前的版本，是大小写敏感的，所以要注意代码在不同MySQL版本间的兼容性问题。
* 不能显式地声明自定义变量的类型。确定未定义变量的具体类型的时机在不同
* MySQL版本中也可能不一样。如果你希望变量是整数类型，那么最好在初始化的时候就赋值为0,如果希望是浮点型则赋值为0.0,如果希望是字符串则赋值为"，用<M户自定义变量的类型在赋值的时候会改变。MySQL的用户自定义变量是一个动态 类型。
* MySQL优化器在某些场景下可能会将这些变量优化掉，这可能导致代码不按预想的方式运行。
* 赋值的顺序和赋值的时间点并不总是固定的，这依赖于优化器的决定。实际情况可能很让人困惑，后面我们将看到这一点。
* 赋值符号：=的优先级非常低，所以需要注意，赋值表达式应该使用明确的括号。
* 使用未定义变量不会产生任何语法错误，如果没有意识到这一点，非常容易犯错。

### 案例

#### 队列表

#### 计算两点之间距离

#### 使用用户自定义的函数

### 总结