# MySQL架构与历史

## MySQL逻辑架构

MySQL服务器逻辑架构图

第二层架构，大多数MySQL的核心服务功能都在这一层，包括查询解析、分析、优化、缓存以及所有的内置函数(例如，日期、时间、数学和加密函数)，所有跨存储引擎的功能都在这一层实现：存储过程、触发器、视图等。

### 1.1.2 优化与执行

用户可以使用特殊的关键字(hint)优化器，影响它的决策过程。也可以请求优化器解释

## 并发控制

### 读写锁

### 锁粒度

## 事务

事务内的语句，要么全部执行成功，要么全部执行失败。

原子性、一致性、隔离性、持久性

### 隔离级别

未提交读、提交读、可重复读、可串行化

### 死锁

## 1.5 MySQL的存储引擎

创建表时，MySQL会在数据库子目录下创建一个和表同名的.frm文件保存表的定义。

可以使用show table status命令显示表的相关信息。

### 1.5.1 InnoDB存储引擎

**InnoDB概览**

InnoDB的数据存储在表空间中，表空间是由InnoDB管理的一个黑盒子，由一系列的数据文件组成。InnoDB使用MVCC来支持高并发，并且实现了4个标准的隔离级别。其默认级别是REPEATABLE READ(可重复读)，并且通过间隙锁(next-key locking)策略防止幻读的出现。间隙锁使得InnoDB不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定，以防止幻影行的插入。

InnoDB是基于聚簇索引建立的。聚簇索引对主键查询有很高的性能。不过它的二级索引(非主键索引)中必须包含主键列，所以如果主键列很大的话，其他的所有索引都会很大。

### 1.5.6 转换表的引擎

**ALTER TABLE**

将表从一个引擎修改为另一个引擎最简单的办法是使用ALTER TABLE语句。例如，将myTable表的引擎修改为InnoDB，ALTER TABLE myTable ENGINE = InnoDB；该语法可以适用于任何引擎。但有一个问题：需要执行很长时间。MySQL会按行将数据从原表复制到一张新的表中，在复制期间会消耗系统所有的I/O能力，同时原表上会加上读锁。一个替代的方案，是采用接下来将讨论的导出与导入的方法，手工进行表的复制。

同一个数据库中不能存在相同的表名，即使使用的是不同的存储引擎。

# MySQL基准测试

# 第3章 服务器性能剖析

# 第4章 Schema与数据类型优化

## 4.1 选择优化的数据类型

### 4.1.1 整数类型

MySQL可以为整数类型指定宽度，例如INT(11)，对大多数应用这是没有意义的：它不会限制值的合法范围，只是规定了MySQL的一些交互工具(例如MySQL命令行客户端)用来显示字符的个数。对于存储和计算来说，INT(1)和INT(20)是相同的。

### 4.1.2 实数类型

实数是带有小数部分的数字。然而，它们不只是为了存储小数部分；也可以使用DECIMAL存储比BIGINT还大的整数。

### 4.4.2 计数器表

如果应用在表中保存计数器，则在更新计数器时可能碰到并发问题。可以用这种表缓存一个用户的朋友数、文件下载次数等。创建一张独立的表存储计数器是一个好主意，这样可使计数器表小且快。

假设只有一行数据来记录网站的点击次数，那么，对于任何想要更新这一行的事务来说，这条记录上都有一个全局的互斥锁。这会使得这些事务只能串行执行。所以要获得更高的并发性能，可以将计数器保存在多行中，每次随机选择一行进行更新，最后使用聚合函数SUM()来统计每行的次数。

## 4.5 加快ALTER TABLE操作的速度

MySQL执行大部分修改表结构操作的方法是用新的结构创建一张空表，从旧表中查出所有数据插入新表，然后删除旧表。

一般而言，大部分ALTER TABLE操作将导致MySQL服务中断。对于常见的场景，能使用的技巧只有两种：一种是先在一台不提供服务的机器上执行ALTER TABLE操作，然后和提供服务的主库进行切换；另外一种技巧是“影子拷贝”。影子拷贝的技巧是用要求的表结构创建一张和源表无关的新表，然后通过重命名和删表操作交换两张表。

不是所有的ALTER TABLE操作都会引起表重建。例如，有两种方法可以改变或者删除一个列的默认值（MODIFY COLUMN很慢，ALTER COLUMN很快）。理论上，MySQL可以跳过创建新表的步骤。列的默认值实际上存在表的.frm文件中，所有可以直接修改这个文件而不需要改动表本身。另外一种方法是通过ALTER COLUMN方法来改变列的默认值。这个语句会直接修改.frm文件而不涉及表结构。

### 4.5.1 只修改.frm文件

# 第5章 创建高性能索引

## 5.1 索引基础

索引可以包含一个或多个列的值。如果索引包含多个列，那么列的顺序也十分重要，因为MySQL只能高效的使用最左前缀列。创建一个包含两个列的索引，和创建两个只包含一个列的索引是大不相同的。

### 5.1.1 索引的类型

#### B-Tree索引

B-Tree通常意味着所有的值都是按顺序存储的，并且每一个叶子页到根的距离相同。

下图表示了B-Tree索引的抽象表示：



B-Tree对索引列是顺序组织存储的，所以很适合查找范围数据。

索引对多个值进行排序的依据是CREATE TABLE语句中定义索引时列的顺序。

B-Tree索引适用于全键值、键值范围或键前缀查找。其中键前缀查找只适用于根据最左前缀的查找。

前面所述的索引对如下类型的查询有效。

**全值匹配**

全值匹配指的是和索引中的所有列进行匹配。

**匹配最左前缀**

**匹配列前缀**

也可以只匹配某一列的值的开头部分。例如查找所有以J开头的姓的人。

**匹配范围值**

例如查找姓在Allen和Barrymore之间的人。

**精确匹配某一列并范围匹配另外一列**

例如查找所有姓为Allen，名字以K开头的人

**只访问索引的查询**

B-Tree通常可以支持“只访问索引的查询”，即查询只需要访问索引，而无需访问数据行。

因为索引树中的节点是有序的，所以除了按值查找以外，索引还可以用于查询中的ORDER BY操作。一般来说，如果B-Tree可以按照某种方式查找到值，那么也可以按照这种方式用于排序。

**B-Tree索引的限制**

* 如果不是按照索引的最左列开始查找，则无法使用索引。
* 不能跳过索引中的列。
* 如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优化查询。

这些限制都和索引列的顺序有关。

#### 哈希索引

哈希索引基于哈希表实现，只有精确匹配索引所有列的查询才有效。对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码

哈希索引的限制：

* 哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。
* 哈希索引数据并不是按照索引值顺序存储的，所以也就无法用于排序。
* 哈希索引也不支持部分索引列匹配查找，因为哈希索引始终是使用索引列的全部内容来计算哈希值的。
* 哈希索引只支持等值比较查询，包括=、IN()、<=>(<>和<=>是不同的操作)。也不支持任何范围查询

InnoDB引擎有一个特殊的功能叫做“自适应哈希索引”。当InnoDB注意到某些值被使用的非常频繁时，它会在内存中基于B-Tree索引之上再创建一个哈希索引，这样就让B-Tree索引也具有哈希索引的一些优点，比如快速的哈希查找。

#### 全文索引

## 5.3 高性能的索引策略

### 5.3.1 单独的列

“独立的列”是指索引列不能是表达式的一部分，也不能是函数的参数。

例如，下面这个查询无法使用actor\_id列的索引：

SELECT actor\_id FROM actor WHERE actor\_id + 1 = 5

MySQL无法自动解析这个方程式

### 5.3.2 前缀索引和索引选择性

有时候需要索引很长的字符列，这会让索引变的大且慢。一种方法是使用前面提到的模拟哈希索引。通常还可以索引开始的部分字符，这样可以大大节约索引空间，从而提高索引效率。但这样也会降低索引的选择性。对于BLOB、TEXT或者很长的VARCHAR类型的列，必需使用前缀索引，因为MySQL不允许索引这些列的完整长度。

前缀索引的缺点：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。

### 5.3.3 多列索引

### 5.3.4 选择合适的索引列顺序

在一个多列B-Tree索引中，索引列的顺序意味着索引首先按照最左列进行排序，其次是第二列。

对于如何选择索引的列顺序有一个经验法则：将选择性最高的列放在索引最前列。

可以跑一些查询来确定在这个表中值的分布情况，并确定哪个值的选择性更高。例如，使用如下的语句预测下：



对应条件值的customer\_id数量更小，应该将索引列customer\_id放在前面。

### 5.3.5 聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。InnoDB的聚簇索引实际上在同一个结构中保存了B-Tree索引和数据行。

当表有聚簇索引时，它的数据行实际上存放在索引的叶子页中(普通索引的叶子页中只是存储了指向数据行的指针)。



InnoDB的二级索引和聚簇索引很不相同。InnoDB的二级索引的叶子节点中存储的不是“行指针”，而是主键值，并以此作为指向行的“指针”。这样的策略减少了当出现行移动或者数据页分裂二级索引的维护工作。使用主键值当作指针会让二级索引占用更多的空间，换来的好处是，InnoDB在移动行时无需更新二级索引中的这个“指针”。

**在InnoDB表中按照主键顺序插入行**

最好避免随机的聚簇索引，特别是对于I/O密集型的应用。随机值作为聚簇索引会使得索引的插入变得完全随机，使得数据没有任何聚集特性。

插入相同的记录，与使用自增的主键相比，使用随机值作为主键会使得插入行花费更长的时间，而且索引占用的空间也更大，毫无疑问其中一个原因是页分裂和碎片化。

下面来看下自增主键插入和随机值插入的过程：

***自增主键***：

因为主键的值是顺序的，所以InnoDB把每一条记录都存储在上一条记录的后面。当达到页的最大填充因子时(InnoDB默认的最大填充因子是页大小的15/16，留出部分空间用于以后的修改)，下一条记录就会写入新的页中。

***随机主键***：

因为新行的主键值不一定比之前的主键值大，所以InnoDB无法简单地总把新行插入索引的最后，而是需要为新行寻找合适的位置------通常是已有数据的中间位置-----并且分配空间。这会导致很多的额外工作，并导致数据分布不够优化。下面是总结的一些缺点：

* 写入的目标页可能已经刷到磁盘上并从缓存中移除，或者是还没有被加载到缓存中，InnoDB在插入之前不得不先找到并从磁盘读取目标页到内存中，这将导致大量的随机I/O。
* 因为写入是乱序的，InnoDB不得不频繁地做页分裂操作(为了保持树的平衡)，以便为新的行分配空间。页分裂会导致移动大量的数据，一次插入最少需要修改三个页而不是一个页。
* 由于频繁的页分裂，页会变得稀疏并被不规则的填充，所以最终数据会有碎片。

### 5.3.6 覆盖索引

如果一个索引包含所有需要查询的字段的值，我们就称之为“覆盖索引”。

不是所有类型的索引都可以成为覆盖索引。覆盖索引必须要存储索引列的值，而哈希索引、空间索引和全文索引等都不存储索引列的值，所以MySQL只能使用B-Tree索引做覆盖索引。

当发起一个被索引覆盖的查询(也叫做索引覆盖查询)时，在EXPLAIN的extra列可以看到“Using index”的信息。

MySQL能在索引中做最左前缀匹配的LIKE比较，因为该操作可以转化为简单的比较操作。

扫描索引本身是很快的，因为只需要从一条索引记录移动到紧接着的下一条记录。但如果索引不能覆盖查询所需要的全部列，那就不得不每扫描一条索引记录就都回表查询一次对应的行。这基本上都是随机I/O，因此按索引顺序读取数据的速度通常要比顺序的全表扫描慢，尤其是在I/O密集型的工作负载时。

## 7.1 分区表

MySQL分区意味着索引也是按照分区的子表定义的，而没有全局索引。这和Oracle不同，在Oracle中可以更加灵活的定义索引和表是否进行分区。

分区本身也有一些限制：

\* 一个表最多只能有1024个分区。

\* 在MySQL5.1中，分区表达式必须是整数，或者是返回整数的表达式。在MySQL5.5中，某些场景中可以直接使用列来进行分区。

\* 如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来。

\* 分区表中无法使用外键约束

### 7.1.1 分区表的原理

所有的底层表都必须使用相同的存储引擎。分区表的索引只是在各个底层表上加上各自加上一个完全相同的索引。

分区表上的操作按照下面的操作逻辑进行：

select查询

当查询一个分区表的时候，分区层先打开并锁住所有底层表，优化器先判断是否可以过滤部分分区，然后再调用对应的存储引擎接口访问各个分区的数据。

insert操作

当写入一条记录时，分区层先打开并锁住所有底层表，然后确定哪个分区接收这条记录，再将记录写入对应底层表。

delete操作

当删除一条记录时，分区层先打开并锁住底层表，然后确定数据对应的分区，最后对相应底层表进行删除操作。

update操作

当更新一条记录时，分区层先打开并锁住所有底层表，MySQL先确定需要更新的记录在哪个分区，然后取出数据并更新，再判断更新后的数据应该放在哪个分区，最后对底层表进行写入操作，并对源数据所在的底层表进行删除操作。

### 7.1.2 分区表的类型

MySQL支持多种分区表。我们看到最多的是根据范围进行分区，每个分区存储落在某个范围的记录，分区表达式可以是列，也可以是包含列的表达式。例如，下表就可以将每一年的销售额存放在不同的分区里：



PARTITION分区子句中可以使用各种函数。但有一个要求，表达式返回的值要是一个确定的整数，而不能是常数。

MySQL还支持键值、哈希和列表分区，这其中有些还支持子分区，不过在生产环境中很少见到。在MySQL5.5中，还可以使用RANGE COLUMNS类型的分区。

### 7.1.3 如何使用分区表

在数据量超大的时候(比如10T)，B-Tree索引就无法起作用了。除非是索引覆盖查询。

为了保证大数据量的可扩展性，一般有下面两个策略：

**全量扫描数据，不要任何索引。**

可以使用简单的分区方式存放表，不要任何索引，根据分区的规则大致定位需要的数据位置。只要能够使用 where条件，将需要的数据限制在少量分区中，则效率是很高的。

**索引数据，并分离热点**

如果数据有明显的“热点”，并且除了这部分数据，其他数据很少被访问到，那么可以将这部分热点数据单独存放在一个分区中，

### 7.1.4 什么情况下会出问题

上面我们介绍的两个分区策略都基于两个非常重要的假设：查询都能够过滤掉很多额外的分区、分区本身并不会带来很多额外的代价。

**NULL值会使分区过滤无效**

分区表达式的值可以是NULL：第一个分区是一个特殊分区。假设按照PARTITION BY RANGE YEAR(order\_date)分区，那么所有order\_date为NULL或者是一个非法值的时候，记录都会被存放到第一个分区。现在假设有下面的查询：where order\_date between “2012-01-01” and “2012-01-31”。实际上，MySQL会检查两个分区：它会先检查2012这个分区，同时还会检查这个表的第一个分区。检查第一个分区是因为YEAR()函数在接收非法值的时候可能会返回，那么这个值就可能会返回到第一个分区。如果第一个分区非常大，特别是当使用“全量扫描数据，不要任何索引”的策略时，代价会非常大。

**分区列和索引列不匹配**

如果定义的索引列和分区列不匹配，会导致查询无法进行分区过滤。假设在列a上定义了索引，而在列b上进行分区。因为每个分区都有其独立的索引，所以扫描列b上的索引就需要扫描每一个分区内对应的索引。要避免这个问题，应该避免建立和分区列不匹配的索引。

**选择分区的成本可能很高**

**打开并锁住所有底层表的成本可能很高**

维护分区的成本可能很高

### 7.1.5 查询优化

对于访问分区表来说，很重要的一点是要在where条件中带入分区列，有时候即使看似多余的也要带上，这样就可以让优化器能够过滤掉无需访问的分区。

使用EXPLAIN PARTITON是否执行了分区过滤，下面是一个示例：



从上面可以看到，这个查询将访问所有分区。下面我们可以在where条件中加入一个时间限制条件：



但是对于下面的例子却不能过滤分区：



MySQL只能在使用分区函数的列本身进行比较时才能过滤分区，而不能根据表达式的值去过滤分区，即使这个表达式是分区函数也不行。这就和查询中使用独立的列才能使用索引的道理是一样的。索引只需要把上面的查询等价的改成如下形式即可：



一个很重要的原则：即使在创建分区时可以使用表达式，但在查询时却只能根据列来过滤分区。

## 7.2 视图