**存储引擎**

MyISAM：不支持事务，不支持外键，只支持表级锁，并发程度相对较低。支持表压缩（压缩后，此表为只读，不可以写入。使用 myisampack 压缩）和表损坏修复。支持B+树索引、空间数据索引、全文索引。适用于查询非常频繁，更新不频繁的场景。

InnoDB：MySQL默认存储引擎。支持事务，支持外键，支持行级锁和表级锁，并发程度相对较高。支持在线热备份。支持B+树索引、哈希索引（自适应）、全文索引。适用于更新和查询都相当频繁的场景。

MyISAM和InnoDB的区别：

① InnoDB支持事务，MyISAM不支持事务

② InnoDB支持外键，MyISAM不支持外键

③ InnoDB支持行级锁和表级锁，MyISAM只支持表级锁

④ InnoDB支持在线热备份，MyISAM不支持在线热备份

⑤ MyISAM支持事务空间数据索引、表压缩等，InnoDB不支持

⑥ InnoDB适用于更新和查询都相当频繁的场景。，MyISAM适用于查询非常频繁，更新不频繁的场景。

**索引在不同规格数据表中的性能**

① 对于非常小的表、大部分情况下简单的全表扫描比建立索引更高效。

② 对于中到大型的表，索引就非常有效。

③ 但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术。

**B+树索引**

B+Tree 索引是大多数 MySQL 存储引擎的默认索引类型。

因为不再需要进行全表扫描，只需要对树进行搜索即可，因此查找速度快很多。**除了用于查找，还可以用于排序和分组。**

可以指定多个列作为索引列，多个索引列共同组成键。

B+Tree 索引适用于**全键值、键值范围和键前缀查找**，其中键前缀查找只适用于**最左前缀查找**。

InnoDB 的 B+Tree 索引分为**主索引**和**辅助索引**。

**主索引 & 辅助索引（都是B+Tree索引）**

**主索引：**也叫**聚簇索引**，数据行的物理顺序与索引的逻辑顺序相同，一个表中只能拥有一个聚簇索引。主索引的叶子节点 data 域记录着完整的数据记录本身。

**辅助索引：**也叫**非聚簇索引、二级索引**，不是聚簇索引的索引就是非聚簇索引，数据行的物理顺序与索引的逻辑顺序不一定相同，一个表中可以拥有多个非聚簇索引。辅助索引的叶子节点的 data 域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行二次查找，这个过程称为**回表**。

**主键和聚簇索引的关系**

主键 ≠ 聚簇索引

① 如果一张表定义了主键，那么主键作为聚簇索引。

② 如果没有定义主键，那么该表的第一个唯一非空索引被作为聚簇索引。

③ 如果没有主键也没有唯一非空索引，那么innodb内部会生成一个隐藏的6字节的主键作为聚簇索引，该列的值会随着数据的插入自增。

**哈希索引**

哈希索引能以O（1）时间进行查找，但是失去了有序性：

① 无法用于排序与分组

② 只支持精确查找，无法用于部分查找和范围查找

InnoDB 存储引擎有一个特殊的功能叫“自适应哈希索引”，当某个索引被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找。

**全文索引**

MyISAM 存储引擎支持全文索引，用于**查找文本中的关键词**，而不是直接比较是否相等。

查找条件使用 WHERE  MATCH (columnName) AGAINST ('string')

全文索引使用**倒排索引**实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

**空间数据索引（R-Tree）**

MyISAM 存储引擎支持空间数据索引，可以用于地理数据存储。空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用 GIS 相关的函数来维护数据。

**联合索引**

两个或更多个列上的索引被称作联合索引，联合索引又叫复合索引。对于复合索引：Mysql 从左到右的使用索引中的字段，一个查询可以只使用索引中的一部份，但只能是最左侧部分。

例如索引是key index (a,b,c)，可以支持[a]、[a,b]、[a,b,c] 3种组合进行查找，但不支 [b,c] 进行查找。当最左侧字段是常量引用时，索引就十分有效。

**注意：索引只能创建在表上，不能创建在视图上。**

**索引的优点**

① 创建唯一性索引，保证数据库表中每一行数据的唯一性。

② 大大加快数据的检索速度，这是创建索引的最主要的原因。

③ 加速数据库表之间的连接，特别是在实现数据的参照完整性方面特别有意义。

④ 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

**索引的缺点**

① 索引需要占用物理空间，除了数据表占用数据空间之外，每一个索引还要占一定的物理空间，如果建立聚簇索引，那么需要的空间就会更大。

② 创建索引和维护索引要耗费时间。

**索引在哪些情况下会失效？**

① MySQL估计全表扫描比使用索引快，则不会使用索引。

② 如果条件中有OR，即使其中有条件带索引也不会使用索引。

③ 联合索引不符合最左前缀原则，索引失效。

④ LIKE以%开头，索引失效。

⑤ 索引列是表达式的一部分或函数的参数，索引失效。

⑥ 使用 > < != BETWEEN...AND...，索引失效。

⑦ 条件中字符串不加单引号，索引失效。

**那些列适合建立索引？**

① order by、group by后面的字段适合建立索引。

② 经常用作查询选择 where 后的字段适合建立索引。

③ 经常用作表连接 join 后的字段适合建立索引。

**索引优化策略**

① 使用多列索引。在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。

② 将选择性最强的索引列放在前面。索引的选择性是指：不重复的索引值和记录总数的比值，即COUNT（DISTINCT colName）/ COUNT（\*）。最大值为 1，此时每个记录都有唯一的索引与其对应。选择性越高，查询效率也越高。

③ 使用前缀索引。对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。对于前缀长度的选取需要根据索引选择性来确定，如果选择太短则索引选择性低，太长则造成空间浪费。

④ 使用覆盖索引。索引包含所有需要查询的字段的值。具有以下优点：

1）索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。

2）若辅助索引能够覆盖查询，则无需对主索引进行二次查询。

**查询性能优化**

**1、使用Explain分析查询性能**

Explain 用来分析 SELECT 查询语句，开发人员可以通过分析 Explain 结果来优化查询语句。

比较重要的字段有：

\* select\_type : 查询类型，有简单查询、联合查询、子查询等

\* key : 使用的索引

\* rows : 扫描的行数

1. **优化数据访问**

**2.1 减少请求的数据量**

（一）只返回必要的列

最好不要使用 SELECT \* 语句。

（二）只返回必要的行

使用 WHERE 语句进行查询过滤，有时候也需要使用 LIMIT 语句来限制返回的数据。

（三）缓存重复查询的数据

使用缓存可以避免在数据库中进行查询，特别要查询的数据经常被重复查询，缓存可以带来的查询性能提升将会是非常明显的。

**2.2 减少服务器端扫描的行数**

最有效的方式是使用索引来覆盖查询。

1. **重构查询方式**

**1. 切分大查询**

一个大查询如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

**2.分解大连接查询**

将一个大连接查询（JOIN）分解成对每一个表进行一次单表查询，然后将结果在应用程序中进行关联，这样做的好处有：

① 让缓存更高效。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。而分解后的多个查询，即使其中一个表发生变化，对其它表的查询缓存依然可以使用。

② 分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询。

③ 减少锁竞争。

④ 在应用层进行连接，可以更容易对数据库进行拆分，从而更容易做到高性能和可扩展。

**分库分表**

简单来说，数据的切分就是通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库（主机）中，以达到分散单台设备负载的效果，即分库分表。

数据的切分根据其切分规则的类型，可以分为如下两种切分模式。

垂直（纵向）切分：把单一的表拆分成多个表，并分散到不同的数据库（主机）上。

水平（横向）切分：根据表中数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库（主机）上。水平切分又称为 Sharding，它是将同一个表中的记录拆分到多个结构相同的表中。当一个表的数据不断增多时，Sharding 是必然的选择，它可以将数据分布到集群的不同节点上，从而缓解单个数据库的压力。

**Sharding策略**

① 哈希取模：hash(key) % NUM\_DB。

② 范围：可以是 ID 范围也可以是时间范围。

③ 映射表：使用单独的一个数据库来存储映射关系。

**主从复制和读写分离**

**主从复制**

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

\* binlog 线程 ：负责将主服务器上的数据更改写入二进制文件（binlog）中。

\* I/O 线程 ：负责从主服务器上读取二进制日志文件，并写入从服务器的中继日志中。

\* SQL 线程 ：负责读取从服务器的中继日志并重放其中的 SQL 语句。

### **读写分离**

主服务器用来处理**写操作以及实时性要求比较高的读操作**，而从服务器用来处理**读操作**。

读写分离常用**代理方式**来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。

MySQL 读写分离能提高性能的原因在于：

① 主从服务器负责各自的读和写，极大程度缓解了锁的争用；

② 从服务器可以配置 MyISAM 引擎，提升查询性能以及节约系统开销；

③ 增加冗余，提高可用性。

MySQL索引总结（一 ~ 四）

<http://www.cnblogs.com/crazylqy/p/7615388.html>

# MySQL中的聚集索引和辅助索引知识详解

<https://blog.csdn.net/w892824196/article/details/82661446>

唯一约束就是唯一索引。唯一索引不允许具有索引值相同的行，从而禁止重复的索引或键值。系统在创建该索引时检查是否有重复的键值，并在每次使用 INSERT 或 UPDATE 语句添加数据时进行检查。

**MVCC**

MySql --MVCC实现原理（乐观锁实现原理）

<https://blog.csdn.net/ai_xiangjuan/article/details/78568337>

**快照读&当前读**

**快照读：**快照读读取的是快照版本，也就是历史版本。在事务中使用普通的select语句使用的就是快照读（select \* from table ...）。快照读也就是MVCC的select规则，即事务只能读取创建版本号早于当前事务版本号（包括等于）的行，事务读取的行要么是事务开始前已经提交的，要么是事务自身插入或修改的。行的删除版本号，要么未定义，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。

**当前读：**读取的是最新的数据，也就是忽略版本号的约束表中真正的数据。

在以下情况会使用当前读：

select \* from table where ? lock in share mode;

select \* from table where ? for update;

insert;

update;

delete;

其中第一个语句会加S锁，其他语句均加X锁（只有select语句才加lock in share mode 或 for update，insert、update和delete不加）

<https://www.cnblogs.com/cat-and-water/p/6427612.html>

SELECT ... LOCK IN SHARE MODE 行共享锁（事务结束自动释放）

SELECT ... FOR UPDATE 行排它锁（事务结束自动释放）

LOCK TABLE tb\_name READ 表共享锁（UNLOCK TABLES释放）

LOCK TABLE tb\_name WRITE 表排它锁（UNLOCK TABLES释放）

innodb 的行锁是在有索引的情况下，没有索引的表是锁定全表的。

**Record Lock & Gap Lock & Next-Key Locks**

Mysql加锁过程详解（7）-初步理解MySQL的gap锁

<http://www.cnblogs.com/crazylqy/p/7689447.html>

Mysql加锁过程详解（8）-理解innodb的锁(record,gap,Next-Key lock)

<http://www.cnblogs.com/crazylqy/p/7773492.html>

**MySQL如何解决幻读？**

MySQL使用MVCC + Next-Key Locks解决幻读。除非当前事务中只存在快照读，可以用MVCC解决幻读。但如果当前事务中存在当前读，就必须使用Next-Key Locks来解决幻读。Next-Key Locks对记录索引和索引间的空隙加了锁，使得其他事务无法插入或删除数据，真正避免了幻读的产生。

具体分析见：

MVCC能解决幻读吗？

<https://www.jianshu.com/p/cef49aeff36b>

**MySQL如何定位慢查询？**

# MySQL优化技术：定位慢查询

<https://blog.csdn.net/qzc70919700/article/details/76566846>