MyISAM**不支持事务**、也**不支持外键**，其优势是**访问的速度快**，**对事务完整性没有要求**或者**以SELECT、INSERT为主**的应用基本上都可以使用这个引擎来创建表。

每个MyISAM在磁盘上存储成3个文件，其文件名都和表名相同，但扩展名分别是：（1）.frm---存储**表的定义**；（2）.MYD---存储**数据**；（3）.MYI---存储**索引**。

数据文件和索引文件可以放置在不同的目录，平均分布IO，获得更快的速度。

MyISAM类型的表可能**会损坏**，原因可能是多种多样的，损坏后的表可能不能被访问，会提示需要修复或者访问后返回错误的结果。

MyISAM的表支持3种不同的存储格式：（1）**静态（固定长度）表**；（2）**动态表**；（3）**压缩表**。

**静态表**是默认的存储格式。静态表中的字段都是非变长字段，这样每个记录都是**固定长度**的，这种存储方式的优点是存储非常迅速，容易缓存，出现故障容易恢复；缺点是占用的空间通常比动态表多。静态表的数据在存储时会按照列的宽度定义补足空格，但是在应用访问的时候并不会得到这些空格，这些空格在返回给应用之前已经去掉。但如果需要保存的内容后面本来就带有空格，那么在返回结果的时候也会被去掉。

**动态表**包含**变长字段**，记录不是固定长度的，这样存储的优点是占用的空间相对较少，但是频繁地更新和删除记录会产生碎片，需要定期执行OPTIMIZE TABLE语句或myisamchk-r命令来改善性能，并且在出现故障时恢复相对比较困难。

**压缩表**由**myisampack**工具创建，占据非常小的磁盘空间。因为每个记录是被单独压缩的，所以只有非常小的访问开支。

InnoDB存储引擎提供了具有**提交**、**回滚**和**崩溃恢复**能力的**事务**安全。但是对比MyISAM的存储引擎，**InnoDB写的处理效率差一些**，并且会**占用更多的磁盘空间以保留数据和索引**。

InnoDB表的**自动增长列必须是索引**。**如果是组合索引，也必须是组合索引的第一列**，但是对于MyISAM表，**自动增长列可以是组合索引的其他列**，这样插入记录后，自动增长列是按照组合索引的前面几列进行排序后递增的。

MySQL支持外键的存储引擎只有InnoDB，在创建外键的时候，要求**父表必须有对应的索引**，**子表在创建外键的时候也会自动创建对应的索引**。在创建外键索引时，可以指定在删除、更新父表时，对子表进行的相应操作，包括RESTRICT、CASCADE、SET NULL和NO ACTION。其中**RESTRICT和NO ACTION相同，是指限制在子表有关联记录的情况下父表不能更新**；**CASCADE表示父表在更新或者删除时，更新或者删除子表对应记录**；**SET NULL则表示父表在更新或者删除的时候，子表的对应字段被SET NULL**。

当某个表被其他表创建了外键参考，那么该表的对应索引或者主键**禁止被删除**。

InnoDB存储表和索引有以下两种方式。

（1）使用**共享表空间存储**，这种方式创建的表的**表结构保存在.frm文件中**，**数据和索引保存在innodb\_data\_home\_dir和innodb\_data\_file\_path定义的表空间中**，可以是多个文件。

（2）使用**多表空间存储**，这种方式创建的表的**表结构仍然保存在.frm文件中，但是每个表的数据和索引单独保存在.ibd中**。**如果是个分区表，则每个分区对应单独的.ibd文件，文件名是“表名+分区名”，可以在创建分区的时候指定每个分区的数据文件的位置，以此来将表的IO均匀分布在多个磁盘上**。

多表空间的数据文件没有大小限制，不需要设置初始大小，也不需要设置文件的最大限制、扩展大小等参数。

对于使用多表空间特性的表，可以比较方便地进行单表备份和恢复操作，但是直接复制.ibd文件是不行的，因为**没有共享表空间的数据字典信息**，**直接复制.bd文件和.frm文件恢复时是不能被正确识别的**。

**即便在多表空间的存储方式下，共享表空间仍然是必须的，InnoDB把内部数据词典和在线重做日志放在这个文件中**。

如何选择合适的存储引擎？

（1）MyISAM：如果应用是**以读操作和插入操作为主**，只有**很少的更新和删除操作**，并且**对事务的完整性、并发性要求不是很高**，那么选择这个存储引擎是非常适合的。MyISAM是在Web、数据仓储和其他应用环境下最常使用的存储引擎之一。

（2）InnoDB：用于**事务**处理应用程序，支持**外键**。如果应用对事务的完整性有比较高的要求，**在并发条件下要求数据的一致性**，数据操作除了插入和查询以外，还包括**很多的更新、删除操作**，那么InnoDB存储引擎应该是比较合适的选择。InnoDB存储引擎除了有效地**降低由于删除和更新导致的锁定**，还可以确保事务的完整提交和回滚，对于类似计费系统或者财务系统等对数据准确性要求比较高的系统，InnoDB都是合适的选择。

由于CHAR是固定长度，所以它的**处理速度**比VARCHAR快得多，但是其缺点是浪费存储空间，程序需要对行尾空格进行处理，所以对于那些长度变化不大并且对查询速度有较高要求的数据可以考虑使用CHAR类型来存储。在MySQL中，不同的存储引擎对CHAR和VARCHAR的使用原则有所不同，这里简单概括如下：

（1）MyISAM存储引擎：建议使用**固定长度的数据列代替可变长度的数据列**。

（2）InnoDB存储引擎：建议使用**VARCHAR类型**。**对于InnoDB数据表，内部的行存储格式没有区分固定长度和可变长度列（所有数据行都使用指向数据列值的头指针），因此在本质上，使用固定长度的CHAR列不一定比使用可变长度VARCHAR列性能要好**。因而，主要的性能因素是**数据行使用的存储总量**。由于CHAR平均占用的空间多于VARCHAR，因此使用VARCHAR来最小化需要处理的数据行的存储总量和磁盘I/O是比较好的。

BLOB能用来保存**二进制**数据，比如照片；而TEXT只能保存**字符**数据，比如一篇文章或者日记。

（1）BLOB和TEXT值会引起一些性能问题，特别是**在执行了大量的删除操作时**。

删除操作会在数据表中留下很大的“空洞”，以后填入这些“空洞”的记录在插入的性能上会有影响。为了提高性能，建议定期使用OPTIMIZE TABLE功能对这类表进行碎片整理，避免因为“空洞”导致性能问题。

（2）可以使用**合成的索引**来提高大文本字段（BLOB或TEXT）的查询性能。

简单来说，合成索引就是根据大文本字段的内容建立一个**散列值**，并把这个值存储在单独的数据列中，接下来就可以通过检索散列值找到数据行了。但是，要注意这种技术只能用于精确匹配的查询（散列值对于类似“<”或“>=”等范围搜索操作符是没有用处的）。如果需要对BLOB或者CLOB字段进行模糊查询，MySQL提供了前缀索引，也就是只为字段的前n列创建索引。

（3）**在不必要的时候避免检索大型的BLOB或TEXT值**，防止在网络上传输大量的值。

（4）**把BLOB或TEXT列分离到单独的表中**。

在某些环境中，如果把这些数据列移动到第二张数据表中，可以把原数据表中的数据列转换为固定长度的数据行格式，那么它就是有意义的。这会减少主表中的碎片，可以得到固定长度数据行的性能优势。它还可以使主数据表在运行SELECT \* 查询的时候不会通过网络传输大量的BLOB或TEXT值。

浮点数一般用于表示含有小数部分的数值。当一个字段被定义为浮点类型后，如果插入数据的精度超过该列定义的实际精度，则插入值会被**四舍五入**到实际定义的精度值，然后插入，四舍五入的过程不会报错。在MySQL中**float、double**（或real）用来表示浮点数。

定点数不同于浮点数，定点数实际上是以**字符串**形式存放的，所以定点数可以更精确地保存数据。如果实际插入的数值精度大于实际定义的精度，则MySQL会进行警告（默认的SQLMode下），但是数据按照实际精度四舍五入后插入；如果SQLMode是在TRADITIONAL（传统模式）下，则系统会直接报错，导致数据无法插入。在MySQL中，**decimal**（或numberic）用来表示定点数。

在精度要求比较高的应用中（比如货币）要使用定点数而不是浮点数来保存数据。

设计索引的原则：

（1）**搜索的索引列，不一定是所要选择的列**。换句话说，最适合索引的列是出现在**WHERE子句中的列**，或**连接子句中指定的列**，而不是出现在SELECT关键字后的选择列表中的列。

（2）**使用唯一索引**。考虑某列中值的分布。索引的列的基数越大，索引的效果越好。例如，存放出生日期的列具有不同的值，很容易区分各行。而用来记录性别的列，只含有“M”和“F”，则对此列进行索引没有多大用处，因为不管搜索哪个值，都会得出大约一半的行。

（3）**使用短索引**。如果对字符串列进行索引，应该指定一个前缀长度，只要有可能就应该这样做。例如，有一个CHAR(200)列，如果在前10个或20个字符内，多数值是唯一的，那么就不要对整个列进行索引。对前10个或20个字符进行索引能够节省大量索引空间，也可能会使查询更快。较小的索引设计的磁盘IO较少，较短的值比较起来更快。更为重要的是，对于较短的键值，索引高速缓存中的块能容纳更多的键值，因此，MySQL也可以在内存中容纳更多的值。这样就增加了找到行而不用读取索引中较多块的可能性。

（4）**利用最左前缀**。在创建一个n列的索引时，实际是创建了MySQL可利用的n个索引。多列索引可起几个索引的作用，因为可利用索引最左边的列集来匹配行。这样的列集称为最左前缀。

（5）**不要过度索引**。不要以为索引“越多越好”，什么东西都用索引是错误的。每个额外的索引都要**占用额外的磁盘空间**，并**降低写操作的性能**。**在修改表的内容时，索引必须进行更新，有时可能需要重构，因此，索引越多，所花的时间越长**。如果有一个索引很少利用或从不使用，那么会不必要地减缓表的修改速度。此外，MySQL在生成一个执行计划时，要考虑各个索引，这也要花费时间。**创建多余的索引给查询优化带来了更多的工作**。索引太多，也可能**会使MySQL选择不到所要使用的最好索引**。只保持所需的索引有利于查询优化。

（6）**对于InnoDB存储引擎的表，记录默认会按照一定的顺序保存**，如果有明确定义的主键，则**按照主键顺序保存**。如果没有主键，但是有唯一索引，那么就是**按照唯一索引的顺序保存**。如果既没有主键又没有唯一索引，那么表中会**自动生成一个内部列，按照这个列的顺序保存**。按照主键或者内部列进行的访问是最快的，所以InnoDB表尽量自己指定主键，当表中同时有几个列都是唯一的，都具有作为主键的时候，要**选择最常作为访问条件的列作为主键**，提高查询的效率。另外，还需要注意，InnoDB表的普通索引都会保存主键的键值，所以**主键要尽可能选择较短的数据类型，可以有效地减少索引的磁盘空间，提高索引的缓存效果**。

HASH索引的特征：

（1）**只能使用=或<=>操作符的等式比较**。

（2）优化器**不能使用HASH索引来加速ORDER BY操作**。

（3）**MySQL不能确定在两个值之间大约有多少行**。如果将一个MyISAM表改为HASH索引的MEMORY表，会影响一些查询的执行效率。

（4）**只能使用整个关键字来搜索一行**。

而对于BTREE索引，当使用>、<、>=、<=、BETWEEN、!=或者<>，或者LIKE ‘pattern’（其中**‘pattern’不以通配符开始**）操作符时，都可以使用相关列上的索引。

视图是一种虚拟存在的表，对于使用视图的用户来说基本上是透明的。视图并不在数据库中实际存在，行和列数据来自定义视图的查询中使用的表，并且是在使用视图时动态生成的。

视图相对于普通的表的优势主要包括以下几项。

（1）**简单**：使用视图的用户完全**不需要关心后面对应的表的结构、关联条件和筛选条件**，对用户来说已经是**过滤好的符合条件的结果集**。

（2）**安全**：**使用视图的用户只能访问他们被允许查询的结果集**，**对表的权限管理并不能限制到某个行某个列，但是通过视图就可以简单地实现**。

（3）**数据独立**：**一旦视图的结构确定了，可以屏蔽表结构变化对用户的影响，源表增加列对视图没有影响**；**源表修改列名，则可以通过修改视图来解决，不会造成对访问者的影响**。

存储过程和函数是**事先经过编译并存储在数据库中的一段SQL语句的集合**，调用存储过程和函数可以简化应用开发人员的很多工作，减少数据在数据库和应用服务器之间的传输，对于提高数据处理的效率是有好处的。

存储过程和函数的区别在于**函数必须有返回值，而存储过程没有**，**存储过程的参数可以使用IN、OUT、INOUT类型，而函数的参数只能是IN类型的**。**如果有函数从其他类型的数据库迁移到MySQL，那么就可能因此需要将函数改造成存储过程**。

触发器是**与表有关的数据库对象**，在满足定义条件时触发，并执行触发器中定义的语句集合。触发器的这种特性可以协助应用**在数据库端确保数据的完整性**。触发器是行触发的，每次增加、修改或者删除记录都会触发进行处理，**编写过于复杂的触发器或者增加过多的触发器对记录的插入、更新、删除操作肯定会有比较严重的影响**。

**SQL注入**就是**利用某些数据库的外部接口将用户数据插入到实际的数据库操作语言（SQL）当中**，从而达到入侵数据库乃至操作系统的目的。

如何预防SQL注入？

（1）使用PreparedStatement+Bind-variable可以防止SQL注入。PreparedStatement是由JDBC驱动来支持的，在使用PreparedStatement语句的时候，仅仅做了简单的替换和转义，并不是MySQL提供了PreparedStatement的特性。

（2）**对特殊字符进行转换**。

（3）对用户提交或可能改变的数据进行简单分类，分别**应用正则表达式来对用户提供的输入数据进行严格的检测和验证**。

分区是指根据一定的规则，数据库把一个表分解成多个更小的、更容易管理的部分。就访问数据库的应用而言，逻辑上只有一个表或一个索引，但是实际上这个表可能由数10个物理分区对象组成，每个分区都是一个独立的对象，可以独自处理，可以作为表的一部分进行处理。分区对应用来说是完全透明的，不影响应用的业务逻辑。

MySQL分区的优点主要包括以下4方面。

1. 和单个磁盘或者文件系统分区相比，可以存储更多数据。

（2）优化查询。在WHERE子句中包含分区条件时，可以只扫描必要的一个或多个分区来提高查询效率；同时在涉及SUM()和COUNT()这类聚合函数的查询时，可以容易地在每个分区上并行处理，最终只需要汇总所有分区得到的结果。

（3）对于已经过期或者不需要保存的数据，可以通过删除与这些数据有关的分区来快速删除数据。

（4）跨多个磁盘来分散数据查询，以获得更大的查询吞吐量。

优化SQL语句的一般步骤：

（1）**通过show status命令了解各种SQL的执行频率**

show session status like ‘Com\_%’;

show global status like ‘Com\_%’;

1. **定位执行效率较低的SQL语句**

* 通过慢查询日志定位那些执行效率较低的SQL语句。
* 慢查询日志在查询结束以后才记录，所以在应用反映执行效率出现问题的时候查询慢查询日志并不能定位问题，可以使用show processlist命令查看当前MySQL在进行的线程，包括线程的状态、是否锁表等，可以实时地查看SQL的执行情况，同时对一些锁表操作进行优化。

（3）**通过EXPLAIN分析低效的SQL的执行计划**

type：表示MySQL在表中找到所需行的方式：

ALL < index < range < ref < eq\_ref < const, system < NULL

ALL，全表扫描，MySQL遍历全表来找到匹配的行。

index，索引全扫描，MySQL遍历整个索引来查询匹配的行。

range，索引范围扫描，常见于<、<=、>、>=、between等操作符。

ref，使用非唯一索引扫描或唯一索引的前缀扫描，返回匹配某个单独值的记录行。

eq\_ref，类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配；简单来说，就是多表连接中使用primary key或者unique index作为关联条件。

const/system：单表中最多有一个匹配行，查询起来非常迅速，所以**这个匹配行中的其他列的值可以被优化器在当前查询中当作常量来处理**。例如，根据主键primary key或者唯一索引unique index进行的查询。

NULL：MySQL不用访问表或索引，直接就能够得到结果。

Extra列为Using where，表示需要根据索引回表查询数据。

Extra列为Using index，表示现在直接访问索引就足够获取到所需要的数据，不需要通过索引回表，Using index也就是平常说的**覆盖索引扫描**。

（4）**通过show profile分析SQL**

Sending data状态表示MySQL线程开始访问数据行并把结果返回给客户端，而不仅仅是返回结果给客户端。由于在Sending data状态下，MySQL线程往往需要做**大量的磁盘读取操作**，所以经常是整个查询中耗时最长的状态。

（5）**通过trace分析优化器如何选择执行计划**

MySQL 5.6提供了对SQL的跟踪trace，通过trace文件能够进一步了解为什么优化器选择A执行计划而不选择B执行计划，帮助我们更好地理解优化器的行为。

（6）**确定问题并采取相应的优化措施**

索引是在MySQL的**存储引擎**层中实现的，而不是在服务器层实现的。

**MySQL中能够使用索引的典型场景**：

（1）**匹配全值**，对索引中所有列都指定具体值，即是对索引中的所有列都有等值匹配的条件。

（2）匹配值的范围查询，对索引的值能够进行范围查找。

（3）匹配最左前缀，仅仅使用索引中的最左边列进行查找，比如在col1 + col2 + col3字段上的联合索引能够被包含col1、(col1 + col2)、(col1 + col2 + col3)的等值查询利用到，可是不能够被col2、(col2 + col3)的等值查询利用到。

（4）**仅仅对索引进行查询，当查询的列都在索引的字段中时，查询的效率更高**。

（5）**匹配列前缀，仅仅使用索引中的第一列，并且只包含索引第一列的开头一部分进行查找**。

（6）**能够实现索引匹配部分精确而其他部分进行范围匹配**。

（7）**如果列名是索引，那么使用column\_name is null就会使用索引**。

（8）MySQL 5.6引入了Index Condition Pushdown（ICP）的特性，进一步优化了查询。Pushdown表示操作下放，**某些情况下的条件过滤操作下放到存储引擎**。

**存在索引但不能使用索引的典型场景**：

（1）**以%开头的LIKE查询不能够利用B-TREE索引**。

因为B-TREE索引的结构，所以以%开头的查询很自然就没法利用索引了，一般都推荐使用全文索引（Fulltext）来解决类似的全文检索问题。或者考虑利用InnoDB的表都是聚簇表的特点，采用一种轻量级别的解决方式：一般情况下，**索引都会比表小，扫描索引要比扫描表更快**（某些特殊情况下，索引比表更大，不在本例讨论范围内），而InnoDB表上二级索引idx\_last\_name实际上存储字段last\_name还有主键actor\_id，那么理想的访问方式应该是首先扫描二级索引idx\_last\_name获得满足条件last\_name like ‘%IN%’的主键actor\_id列表，之后根据主键回表去检索记录，这样访问避开了全表扫描演员表actor产生的大量IO请求。

（2）**数据类型出现隐式转换的时候也不会使用索引**，特别是当列类型是字符串，那么一定记得在WHERE条件中把字符常量值用引号引起来，否则即便这个列上有索引，MySQL也不会用到，因为MySQL默认把输入的常量值进行转换以后才进行检索。

（3）**复合索引的情况下，假如查询条件不包含索引列最左边部分，即不满足最左原则Leftmost，是不会使用复合索引的**。

（4）**如果MySQL估计使用索引比全表扫描更慢，则不使用索引**。

在查询的时候，**筛选性越高越容易使用到索引，筛选性越低越不容易使用索引**。

（5）**用or分割开的条件，如果or前的条件中的列有索引，而后面的列中没有索引，那么涉及的索引都不会被用到**。

**因为or后面的条件列中没有索引，那么后面的查询肯定要走全表扫描，在存在全表扫描的情况下，就没有必要多一次索引扫描增加I/O访问**，一次全表扫描过滤条件就足够了。