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.ibd 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，都可以作为主键的时候，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_%’ ;

（2） **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。

ANALYZE、CHECK、OPTIMIZE、ALTER TABLE 执行期间将对表进程锁定，因此一定注意要在数据库不繁忙的时候执行相关的操作。

优化 INSERT 语句：

（1）**如果同时从同一客户插入很多行，应尽量使用多个值表的 INSERT 语句**，这种方式将**大大缩减客户端与数据库之间的连接、关闭等消耗**，使得效率比分开执行的单个 INSERT 语句快。

（2）**如果从不同客户插入很多行，可以通过使用 INSERT DELAYED 语句得到更高的速度**。DELAYED 的含义是让 INSERT 语句马上执行，其实数据都被放在内存的队列中，并没有真正写入磁盘，这比每条语句分别插入要快得多。

（3）**将索引文件和数据文件分在不同的磁盘上存放**。

优化 ORDER BY 语句：

MySQL 中有两种排序方式：

（1）**通过有序索引顺序扫描直接返回有序数据**，这种方式在使用 EXPLAIN 分析查询的时候显示为 Using index，不需要额外的排序，操作效率较高。

（2）**通过对返回数据进行排序**，也就是通常说的 **Filesort 排序**，**所有不是通过索引直接返回排序结果的排序都叫 Filesort 排序**。Filesort 并不代表通过磁盘文件进行排序，而只是说明进行了一个排序操作，至于排序操作是否使用了磁盘文件或临时表等，则取决于 MySQL 服务器对排序参数的设置和需要排序数据的大小。

Filesort 是通过相应的排序算法，**将取得的数据在 sort_buffer_size 系统变量设置的内存排序区中进行排序，如果内存装载不下，它就会将磁盘上的数据进行分块，再对各个数据块进行排序，然后将各个块合并成有序的结果集**。sort_buffer_size 设置的排序区是每个**线程独占**的，所以同一时刻，MySQL 中存在多个 sort_buffer 排序区。

了解了 MySQL 排序的方式，优化目标就清晰了：尽量**减少额外的排序**，**通过索引直接返回有序数据**。**WHERE 条件和 ORDER BY 使用相同的索引**，并且 **ORDER BY 的顺序和索引顺序相同**，并且 **ORDER BY 的字段都是升序或者都是降序**。否则肯定需要额外的排序操作，这样就会出现 Filesort。

对于 Filesort，MySQL 有两种排序算法：

（1）两次扫描算法：首先**根据条件取出排序字段和行指针信息**，之后**在排序区 sort buffer 中排序**。**如果排序区 sort buffer 不够，则在临时表 Temporary Table 中存储排序结果**。**完成排序后根据行指针回表读取记录**。该算法是 MySQL4.1 之前采用的算法，需要两次访问数据，**第一次获取排序字段和行指针信息**，**第二次根据行指针获取记录**，尤其是第二次读取操作可能导致大量随机 I/O 操作；优点是排序的时候内存开销较少。

（2）一次扫描算法：一次性取出满足条件的行的所有字段，然后在排序区 sort buffer 中排序后直接输出结果集。排序的时候**内存开销比较大**，但是排序效率比两次扫描算法要高。

优化 GROUP BY 语句：

默认情况下，MySQL 对所有 GROUP BY col1，col2，……的字段进行排序。这与在查询中指定 ORDER BY col1，col2，……类似。因此，如果显式包括一个包含相同列的 ORDER BY 子句，则对 MySQL 的实际执行性能没有什么影响。**如果查询包括 GROUP BY 但用户想要避免排序结果的消耗，则可以指定 ORDER BY NULL 禁止排序**。

优化嵌套查询：

子查询可以被更有效率的连接（JOIN）替代。连接（JOIN）之所以更有效率一些，是因为 **MySQL 不需要在内存中创建临时表**来完成这个逻辑上需要两个步骤的查询工作。

优化分页查询：

一般分页查询时，通过创建覆盖索引能够比较好地提高性能。一个常见又非常头痛的分页场景是“LIMIT 1000, 20”，此时 MySQL 排序出前 1020 条记录后仅仅需要返回地 1001 到 1020 条记录，前 1000 条记录都会被抛弃，查询和排序的代价非常高。

（1）第一种优化思路：**在索引上完成分页的操作，最后根据主键关联回原表查询所需要的其他列内容**。

（2）第二种优化思路：**把 LIMIT 查询转换成某个位置的查询**。

使用 SQL 提示：

SQL 提示（SQL HINT）是优化数据库的一个重要手段，简单来说就是在 SQL 语句中加入一些人为的提示来达到优化操作的目的。下面是一个使用 SQL 提示的例子：

```
SELECT SQL_BUFFER_RESULTS * FROM ...
```

这个语句将强制 MySQL 生成一个临时结果集。只要临时结果集生成后，所有表上的锁定均被释放。这能**在遇到表锁定问题时或要花很长时间将结果传给客户端时有所帮助，因为可以尽快释放锁资源**。

（1）USE INDEX

在查询语句中表名的后面，**添加 USE INDEX 来提供希望 MySQL 去参考的索引列表，就可以让 MySQL 不再考虑其他可用的索引**。

（2）IGNORE INDEX

如果用户只是单纯地想**让 MySQL 忽略一个或者多个索引，则可以使用 IGNORE INDEX 作为 HINT**。

（3）FORCE INDEX

为**强制 MySQL 使用一个特定的索引，可在查询中使用 FORCE INDEX 作为 HINT**。

通过拆分提高表的访问效率

（1）垂直拆分：把主码和一些列放到一个表，然后把主码和另外的列放到另一个表中。

如果一个表中某些列常用，而另一些列不常用，则可以采用垂直拆分，另外，垂直拆分可以使得数据行变小，一个数据页就能存放更多的数据，在查询时就会减少 I/O 次数。其缺点是需要管理**冗余列**，**查询所有数据需要联合（JOIN）操作**。

（2）水平拆分：根据一列或多列数据的值把数据行放到两个独立的表中。

水平拆分通常在以下几种情况下使用。

- 表很大，分割后可以**降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，提高查询速度**。
- **表中的数据本来就有独立性**，例如，表中分别记录各个地区的数据或不同时期的数据，特别是**有些数据常用，而另外一些数据不常用**。
- 需要把数据存放到**多个介质**上。

水平拆分会给应用增加复杂度，它通常在查询时需要多个表名，查询所有数据需要 UNION 操作。在许多数据库应用中，这种复杂性会超过它带来的优点，因为只要索引关键字不大，则在索引用于查询时，表中增加 2~3 倍数据率，查询时也就增加读一个索引层的磁盘次数，所以水平拆分要考虑数据量的增长速度，根据实际情况决定是否需要对表进行水平拆分。

数据库规范化越高，那么产生的关系就越多，关系过多的直接结果就是导致表之间的连接操作越频繁，而表之间的连接操作是性能较低的操作，直接影响到查询的速度，所以，对于查询较多的应用，就需要根据实际情况运用**逆规范化**对数据进行设计，通过逆规范化来提高查询的性能。

反规范的好处是**降低连接操作的需求**、**降低外码和索引的数目**，还可能**减少表的数目**，相应带来的问题是可能出现**数据的完整性问题**。**加快查询速度，但会降低修改速度**。

常用的反规范技术：

（1）**增加冗余列**：指在多个表中具有相同的列，它常用来在查询时避免连接操作。

（2）**增加派生列**：指增加的列来自其他表中的数据，由其他表中的数据经过计算生成。增加的派生列其作用是在查询时减少连接操作，避免使用集函数。

（3）**重新组表**：指如果许多用户需要查看两个表连接出来的结果数据，则把这两个表重新组成一个表来减少连接而提高性能。

（4）**分割表**。

逆规范技术需要维护数据的完整性。无论使用何种反规范技术，都需要一定的管理来维护数据的完整性，常用的方法是批处理维护、应用逻辑和触发器。

（1）**批处理维护**是指**对复制列或派生列的修改积累一定的时间后，运行一批处理作业或存储过程对复制或派生列进行修改**，这只能在对实时性要求不高的情况下使用。

（2）数据的完整性也可由**应用逻辑**来实现，这就要求必须**在同一事务中对所有涉及的表进行增、删、改操作**。用应用逻辑来实现数据的完整性风险较大，因为同一逻辑必须在所有的应用中使用和维护，容易遗漏，特别是在需求变化时，不易于维护。

（3）使用**触发器**，对数据的任何修改立即触发对复制列或派生列的相应修改。触发器是**实时**的，而且相应的处理逻辑只在一个地方出现，易于维护。一般来说，是解决这类问题比较好的办法。

对于**数据量较大的表，在其上进行统计查询通常会效率很低**，并且还要**考虑统计查询是否会对在线的应用产生负面影响**。通常在这种情况下，使用**中间表**可以提高统计查询的效率。

中间表在统计查询中经常会用到，其优点如下：

（1）中间表**复制源表部分数据，并且与源表相“隔离”，在中间表上做统计查询不会对在线应用产生负面影响**。

（2）中间表上可以**灵活地添加索引或增加临时用的新字段**，从而达到提高统计查询效率和辅助统计查询作用。

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。

行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般。

MySQL 的表级锁有两种模式：**表共享读锁**和**表独占写锁**。对 MyISAM 表的读操作，不会阻塞其他用户对同一表读请求，但会阻塞对同一表的写请求；对 MyISAM 表的写操作，会阻塞其他用户对同一表的读和写操作；MyISAM 表的读操作和写操作之间，以及写操作之间是串行的。

MyISAM 在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行更新操作（UPDATE、DELETE、INSERT 等）前，会自动给涉及的表加写锁，这个过程并不需要用户干预，因此，用户一般不需要直接用 LOCK TABLE 命令给 MyISAM 表显式加锁。给 MyISAM 表显式加锁，一般是为了在一定程度模拟事务操作，实现对某一时间点多个表的一致性读取。

在用 LOCK TABLES 给表显式加表锁时，必须同时取得所有涉及表的锁，并且 MySQL 不支持锁升级。也就是说，**在执行 LOCK TABLES 后，只能访问显式加锁的这些表，不能访问未加锁的表**；同时，如果加的是读锁，那么只能执行查询操作，而不能执行更新操作。在自动加锁的情况下也是如此，MyISAM 总是一次获得 SQL 语句所需要的全部锁。这也正是 **MyISAM 表不会出现死锁（Deadlock Free）的原因**。

在一定条件下，MyISAM 表也支持查询和插入操作的并发执行。MyISAM 存储引擎有一个系统变量 **concurrent_insert**，专门用以控制其并发插入的行为，其值分别可以为 0、1 或 2。

（1）当 concurrent_insert 设置为 0 时，**不允许并发插入**。

（2）当 concurrent_insert 设置为 1 时，**如果 MyISAM 表中没有空间（即表的中间没有被删除的行），MyISAM 允许在一个进程读表的同时，另一个进程从表尾插入记录**。这也是 MySQL 的默认设置。

（3）当 concurrent_insert 设置为 2 时，**无论 MyISAM 表中有没有空洞，都允许在表尾并发插入记录**。

MyISAM 存储引擎的读锁和写锁是互斥的，读写操作是串行的。那么，一个进程请求某个 MyISAM 表的读锁，同时另一个进程也请求同一表的写锁，MySQL 如何处理呢？答案是**写进程先获得锁**。不仅如此，**即使读请求先到锁等待队列，写请求后到，写锁也会插到读锁请求之前**！这是因为 MySQL 认为写请求一般比读请求更重要。这也正是 MyISAM 表不太适合于有大量更新操作和查询操作应用的原因，因为，**大量的更新操作会造成查询操作很难获得读锁，从而可能永远阻塞**。

一些需要长时间运行的查询操作，也会使写进程“饿死”！因此，应用中应尽量**避免出现长时间运行的查询操作**，不要总想用一条 SELECT 语句来解决问题，因为这种看似巧妙的 SQL 语句，往往比较复杂，执行时间较长，在可能的情况下可以通过使用中间表等措施对 SQL 语句做一定的“分解”，使每一步查询都能在较短时间完成，从而减少锁冲突。如果复杂查询不可避免，应尽量安排在数据库空闲时段执行，比如一些定期统计可以安排在夜间执行。

InnoDB 与 MyISAM 的最大不同有两点：一是支持**事务**（TRANSACTION）；二是采用了**行级锁**。

原子性：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。

一致性：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以保持数据的完整性；事务结束时，所有的内部数据结构（如 B 树索引或双向链表）也都必须是正确的。

隔离性：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。

持久性：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

更新丢失：当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在，就会发生丢失更新问题——最后的更新覆盖了由其他事务所做的更新。

脏读：一个事务正在对一条记录做修改，在这个事务完成并提交前，这条记录的数据就处于不一致状态；这时，另一个事务也来读取同一条记录，如果不加

控制，第二个事务读取了这些“脏”数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。

不可重复读：一个事务在读取某些数据后的某个时间，再次读取以前读过的数据，却发现其读出的数据已经发生了改变或某些记录已经被删除了。

幻读：一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据。

在上面讲到的并发事务处理带来的问题中，“更新丢失”通常是应该完全避免的。但**防止更新丢失，并不能单靠数据库事务控制器来解决，需要应用程序对要更新的数据加必要的锁来解决**，因此，**防止更新丢失应该是应用的责任**。

“脏读”、“不可重复读”和“幻读”，其实都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。数据库实现事务隔离的方式，基本上可分为以下两种：

（1）在读取数据前，对其加锁，阻止其他事务对数据进行修改。

（2）不用加任何锁，通过一定机制生成一个数据请求时间点的一致性数据快照，并用这个快照来提供一定级别（语句级或事务级）的一致性读取。从用户的角度来看，好像是数据库可以提供同一数据的多个版本，因此，这种技术叫做数据**多版本并发控制**（MultiVersion Concurrency Control，简称 MVCC 或 MCC），也经常称为**多版本数据库**。

未提交读（Read uncommitted）：最低级别，只能保证不读取物理上损坏的数据。

已提交读（Read committed）：语句级。

可重复读（Repeatable read）：事务级

可序列化（Serializable）：最高级别，事务级。

InnoDB 实现了以下两种类型的行锁：

（1）**共享锁（S）**：**允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁**。

（2）**排他锁（X）**：**允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁**。

另外，为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB 还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁：

（1）**意向共享锁（IS）**：**事务打算给数据行加行共享锁，事务在给一个数据**

行加共享锁前必须先取得该表的 IS 锁。

（2）**意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加行排他锁前必须先取得该表的 IX 锁。**

如果一个事务请求的锁模式与当前的锁兼容，InnoDB 就将请求的锁授予该事务；反之，如果两者不兼容，该事务就要等待锁释放。

意向锁是 InnoDB 自动加的，不需用户干预。对于 UPDATE、DELETE 和 INSERT 语句，InnoDB 会自动给涉及数据集加排他锁（X）；对于普通 SELECT 语句，InnoDB 不会加任何锁。事务可以通过以下语句显式给记录集加共享锁或排他锁。

（1）**共享锁（S）**：SELECT * FROM table_name WHERE ... LOCK IN SHARE MODE。

（2）**排他锁（X）**：SELECT * FROM table_name WHERE ... FOR UPDATE

用 SELECT ... IN SHARE MODE 获得共享锁，主要用在需要数据依存关系时来确认某行记录是否存在，并确保没有人对这个记录进行 UPDATE 或者 DELETE 操作。但是**如果当前事务也需要对该记录进行更新操作，则很有可能造成死锁，对于锁定行记录后需要进行更新操作的应用，应该使用 SELECT ... FOR UPDATE 方式获得排他锁。**

InnoDB 行锁是通过给索引上的索引项加锁来实现的，如果没有索引，InnoDB 将通过隐藏的聚簇索引来对记录加锁。InnoDB 行锁分为 3 种情形。

（1）**Record lock**：对索引项加锁。

（2）**Gap lock**：对索引项之间的“间隙”、第一条记录前的“间隙”或最后一条记录后的“间隙”加锁。

（3）**Next-key lock**：前两种的组合，对记录及其前面的间隙加锁。

InnoDB 这种行锁实现特点意味着：**如果不通过索引条件检索数据，那么 InnoDB 将对表中的所有记录加锁，实际效果跟表锁一样。**

（1）**在不通过索引条件查询时，InnoDB 会锁定表中的所有记录。**

（2）**由于 MySQL 的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。**

（3）**当表有多个索引的时候，不同的事务可以使用不同的索引锁定不同的行，不论是使用主键索引、唯一索引或者普通索引，InnoDB 都会使用行锁来对数据加锁。**

（4）**即使在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的，如果 MySQL 认为全表扫描效**

率更高，比如对一些很小的表，它就不会使用索引，这种情况下 InnoDB 也会对所有记录加锁。因此，在分析锁冲突时，别忘了检查 SQL 的执行计划，以确认是否真正使用了索引。

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB 会给符合条件的已有数据记录的索引项加锁；**对于键值在条件范围内但并不存在的记录，叫做“间隙（GAP）”，InnoDB 也会对这个“间隙”加锁，这种锁机制就是所谓的 Next-Key 锁**。

举例来说，假如 emp 表中只有 101 条记录，其 empid 的值分别是 1、2、...、100、101，下面的 SQL：

SELECT * FROM emp WHERE empid > 100 FOR UPDATE;

是一个范围条件的检索，InnoDB 不仅会对符合条件的 empid 值为 101 的记录加锁，也会对 empid 大于 101（这些记录不存在）的“间隙”加锁。

InnoDB 使用 Next-Key 锁的目的，一方面是为了防止幻读，以满足相关隔离级别的要求，对于上面的例子，要是不使用间隙锁，如果其他事务插入了 empid 大于 100 的任何记录，那么本事务如果再次执行上述语句，就会发生幻读；另一方面，是为了满足其恢复和复制的需要。

很显然，**在使用范围条件检索并锁定记录时，InnoDB 这种加锁机制会阻塞符合条件范围内键值的并发插入，这往往会造成严重的锁等待**。因此，在实际应用开发中，尤其是并发插入比较多的应用，我们要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。

对基于语句日志格式（SBL）的恢复和复制而言，由于 MySQL 的 BINLOG 是按照事务提交的先后顺序记录的，因此要正确恢复或复制数据，就必须满足：**在一个事务未提交前，其他并发事务不能插入满足其锁定条件的任何记录，也就是不允许出现幻读**。

对于“INSERT INTO target_tab SELECT * FROM source_tab WHERE ...”和“CREATE TABLE new_tab ... SELECT ... FROM source_tab WHERE ...”时要给 source_tab 加锁，而不是使用对并发影响最小的多版本数据来实现一致性读。如果上述语句的 SELECT 是范围条件，InnoDB 还会给源表加 Next-Key 锁。

因此，INSERT ... SELECT ...和 CREATE TABLE ... SELECT ...语句，可能会阻止对源表的并发更新。如果查询比较复杂，会造成严重的性能问题。实际上，MySQL 将这种 SQL 叫做**不确定（non-deterministic）的 SQL**，属于“**Unsafe SQL**”。

如果应用中一定要用这种 SQL 来实现业务逻辑，又不希望对源表的并发更新产生影响，可以采取以下 3 种措施：

（1）将 innodb_locks_unsafe_for_binlog 的值设置为 “on”，强制 MySQL 使用多版本数据一致性读。但付出的代价是可能无法使用 BINLOG 正确地恢复或复制数据。

（2）通过使用 “SELECT * FROM source_tab … INTO outfile” 和 “load data infile …” 语句组合来间接实现，采用这种方式 MySQL 不会给 source_tab 加锁。

（3）使用基于行的 BINLOG 格式和基于行数据的复制。

对于 InnoDB，在绝大部分情况下都应该使用行级锁，因为事务和行锁往往是我们选择 InnoDB 表的理由。但在个别特殊事务中，也可以考虑使用表级锁。

（1）**事务需要更新大部分或全部数据，表又比较大**，如果使用默认的行锁，不仅这个事务执行效率低，而且可能造成其他事务长时间锁等待和锁冲突，这种情况下可以考虑使用表锁来提高该事务的执行速度。

（2）**事务涉及多个表，比较复杂，很可能引起死锁，造成大量事务回滚**。这种情况也可以考虑一次性锁定事务涉及的表，从而避免死锁，减少数据库因事务回滚带来的开销。

当然，应用中这两种事务不能太多，否则，就应该考虑使用 MyISAM 表了。在 InnoDB 下，使用表锁要注意以下两点：

（1）使用 LOCK TABLES 虽然可以给 InnoDB 加表级锁，但必须说明的是，**表锁不是由 InnoDB 存储引擎层管理的，而是由其上一层——MySQL Server 负责的**，仅当 autocommit=0、innodb_table_locks=1（默认设置）时，InnoDB 层才能知道 MySQL 加的表锁，MySQL Server 也才能感知 InnoDB 加的行锁，这种情况下，InnoDB 才能自动识别涉及表级锁的死锁；否则，InnoDB 将无法自动检测并处理这种死锁。

（2）在用 LOCK TABLES 对 InnoDB 表加锁时要注意，要将 AUTOCOMMIT 设为 0，否则 MySQL 不会给表加锁；**事务结束前，不要用 UNLOCK TABLES 释放表锁，因为 UNLOCK TABLES 会隐含地提交事务；COMMIT 或 ROLLBACK 并不能释放用 LOCK TABLES 加的表级锁，必须用 UNLOCK TABLES 释放表锁**。

发生死锁后，InnoDB 一般都能自动检测到，并使一个事务释放锁并回退，另一个事务获得锁，继续完成事务。但**在涉及外部锁或涉及表锁的情况下，InnoDB 并不能完全自动检测到死锁，这需要通过设置锁等待超时参数**

innodb_lock_wait_timeout 来解决。需要说明的是，这个参数并不是只用来解决死锁问题，在并发访问比较高的情况下，如果大量事务因无法立即获得所需的锁而挂起，会占用大量计算机资源，造成严重性能问题，甚至拖垮数据库。我们通过设置合适的锁等待超时阈值，可以避免这种情况发生。

避免死锁的方法：

（1）在应用中，如果不同的程序会并发存取多个表，应尽量约定以**相同的顺序**来访问表，这样可以大大降低产生死锁的机会。

（2）在程序以批量方式处理数据的时候，如果**事先对数据排序，保证每个线程按固定的顺序来处理记录**，也可以大大降低出现死锁的可能。

（3）在事务中，**如果要更新记录，应该直接申请足够级别的锁，即排他锁**，而不应先申请共享锁，更新时再申请排他锁，因为当用户申请排他锁时，其他事务可能又已经获得了相同记录的共享锁，从而造成锁冲突，甚至死锁。

（4）在 REPEATABLE-READ 隔离级别下，如果两个线程同时对相同条件记录用 SELECT ... FOR UPDATE 加排他锁，在没有符合该条件记录情况下，两个线程都会加锁成功。程序发现记录尚不存在，就试图插入一条新记录，如果两个线程都这么做，就会出现死锁。

（5）当隔离级别为 READ COMMITTED 时，如果两个线程都先执行 SELECT ... FOR UPDATE，判断是否存在符合条件的记录，如果没有，就插入记录。此时，只有一个线程能插入成功，另一个线程会出现锁等待，当第 1 个线程提交后，第 2 个线程会因主键重出错，但虽然这个线程出错了，却会获得一个排他锁！这时如果有第 3 个线程又来申请排他锁，也会出现死锁。

如果出现死锁，可以用 SHOW INNODB STATUS 命令来**确定最后一个死锁产生的原因**。返回结果中包括**死锁相关事务的详细信息**，如**引发死锁的 SQL 语句**，**事务已经获得的锁**，**正在等待什么锁**，以及**被回滚的事务**等。

（1）尽量使用**较低的隔离级别**。

（2）精心设计索引，并**尽量使用索引访问数据**，使加锁更精确，从而减少锁冲突的机会。

（3）选择合理的事务大小，**小事务发生锁冲突的几率也更小**。

（4）给记录集显式加锁时，最好**一次性请求足够级别的锁**。比如要修改数据，最好直接申请排他锁，而不是先申请共享锁，修改时再申请排他锁，这样容

易产生死锁。

(5) **不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行**。这样可以大大减少死锁的机会。

(6) 尽量**用相等条件访问数据**，这样可以**避免 Next-Key 锁对并发插入的影响**。

(7) **不要申请超过实际需要的锁级别；除非必须，查询时不要显式加锁**。

(8) **对于一些特定的事务，可以使用表锁来提高处理速度或减少发生死锁的几率**。