一. 优化查询的方法？

1.使用索引

应尽量避免全表扫描，首先应考虑在 where 及 order by ,group by 涉及的列上建立索引。

2.优化 SQL 语句

2.1 通过 explain(查询优化神器)用来查看 SQL 语句的执行效果，可以帮助选择更好的索引和优化查询语句，写出更好的优化语句。通常我们可以对比较复杂的尤其是涉及到多表的 SELECT 语句，把关键字 EXPLAIN 加到前面，查看执行计划。例如：explain select \* fromnews;

2.2 任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

2.3 不在索引列做运算或者使用函数。

2.4 查询尽可能使用 limit 减少返回的行数，减少数据传输时间和带宽浪费。

3.优化数据库对象

3.1 优化表的数据类型

使用 procedure analyse()函数对表进行分析，该函数可以对表中列的数据类型提出优化建议。能小就用小。表数据类型第一个原则是：使用能正确的表示和存储数据的最短类型。这样可以减少对磁盘空间、内存、cpu 缓存的使用。

使用方法：select \* from 表名 procedure analyse();

3.2 对表进行拆分

通过拆分表可以提高表的访问效率。有 2 种拆分方法：

1.垂直拆分

把主键和一些列放在一个表中，然后把主键和另外的列放在另一个表中。如果一个表中某些列常用，而另外一些不常用，则可以采用垂直拆分。

2.水平拆分

根据一列或者多列数据的值把数据行放到二个独立的表中。

3.3 使用中间表来提高查询速度创建中间表，表结构和源表结构完全相同，转移要统计的数据到中间表，然后在中间表上进行统计，得出想要的结果。

4.硬件优化

4.1 CPU 的优化

选择多核和主频高的 CPU。

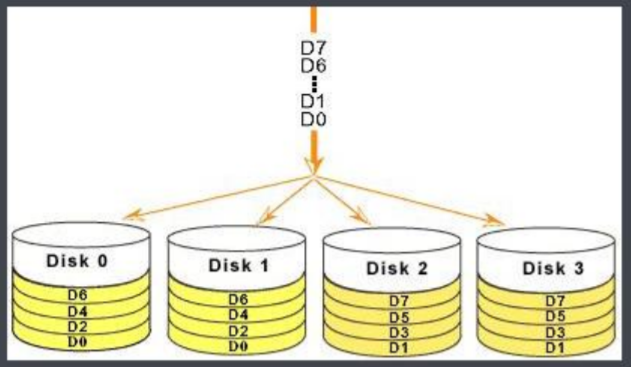
4.2 内存的优化

使用更大的内存。将尽量多的内存分配给 MYSQL 做缓存。

4.3 磁盘 I/O 的优化

4.3.1 使用磁盘阵列

RAID 0 没有数据冗余，没有数据校验的磁盘陈列。实现 RAID 0至少需要两块以上的硬盘，它将两块以上的硬盘合并成一块，数据连续地分割在每块盘上。RAID1 是将一个两块硬盘所构成 RAID 磁盘阵列，其容量仅等于一块硬盘的容量，因为另一块只是当作数据“镜像”。使用 RAID-0+1 磁盘阵列。RAID 0+1 是 RAID 0 和 RAID 1 的组合形式。它在提供与 RAID 1 一样的数据安全保障的同时，也提供了与 RAID 0 近似的存储性能。



4.3.2 调整磁盘调度算法

选择合适的磁盘调度算法，可以减少磁盘的寻道时间。

5.MySQL 自身的优化

对 MySQL 自身的优化主要是对其配置文件 my.cnf 中的各项参数进行优化调整。如指定 MySQL 查询缓冲区的大小，指定 MySQL 允许的最大连接进程数等。

6.应用优化

6.1 使用数据库连接池

6.2 使用查询缓存

它的作用是存储 select 查询的文本及其相应结果。如果随后收到一个相同的查询，服务器会从查询缓存中直接得到查询结果。查询缓存适用的对象是更新不频繁的表，当表中数据更改后，查询缓存中的相关条目就会被清空。 二．如果有一个特别大的访问量到数据库上，怎么做优化？

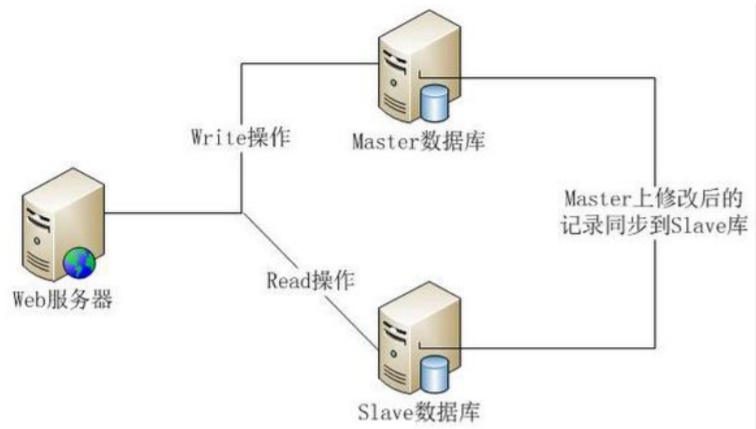
1.使用优化查询的方法（见上面）

2.主从复制，读写分离，负载均衡

目前，大部分的主流关系型数据库都提供了主从复制的功能，通过配置两台（或多台）

数据库的主从关系，可以将一台数据库服务器的数据更新同步到另一台服务器上。网站可以利用数据库的这一功能，实现数据库的读写分离，从而改善数据库的负载压力。一个系统的读操作远远多于写操作，因此写操作发向 master，读操作发向 slaves 进行操作（简单的轮循算法来决定使用哪个 slave）。

利用数据库的读写分离，Web 服务器在写数据的时候，访问主数据库（Master），主数据库通过主从复制机制将数据更新同步到从数据库（Slave），这样当 Web 服务器读数据的时候，就可以通过从数据库获得数据。这一方案使得在大量读操作的 Web 应用可以轻松地读取数据，而主数据库也只会承受少量的写入操作，还可以实现数据热备份，可谓是一举两得的方案。



主从复制的原理：

影响 MySQL-A 数据库的操作，在数据库执行后，都会写入本地的日志系统 A 中。假设，实时的将变化了的日志系统中的数据库事件操作，通过网络发给 MYSQL-B。MYSQL-B 收到后，写入本地日志系统 B，然后一条条的将数据库事件在数据库中完成。那么，MYSQL-A 的变化，MYSQL-B 也会变化，这样就是所谓的 MYSQL 的复制。

在上面的模型中，MYSQL-A 就是主服务器，即 master，MYSQL-B 就是从服务器，即slave。日志系统 A，其实它是 MYSQL 的日志类型中的二进制日志，也就是专门用来保存修改数据库表的所有动作，即 bin log。【注意 MYSQL 会在执行语句之后，释放锁之前，写入二进制日志，确保事务安全】日志系统 B，并不是二进制日志，由于它是从 MYSQL-A 的二进制日志复制过来的，并不是自己的数据库变化产生的，有点接力的感觉，称为中继日志，即 relay log。可以发现，通过上面的机制，可以保证 MYSQL-A 和 MYSQL-B 的数据库数据一致，但是时间上肯定有延迟，即 MYSQL-B 的数据是滞后的。

简化版：

mysql 主(称 master)从(称 slave)复制的原理：

1. master 将数据改变记录到二进制日志(binary log)中,也即是配置文件 log-bin 指定的文件(这些记录叫做二进制日志事件，binary log events)

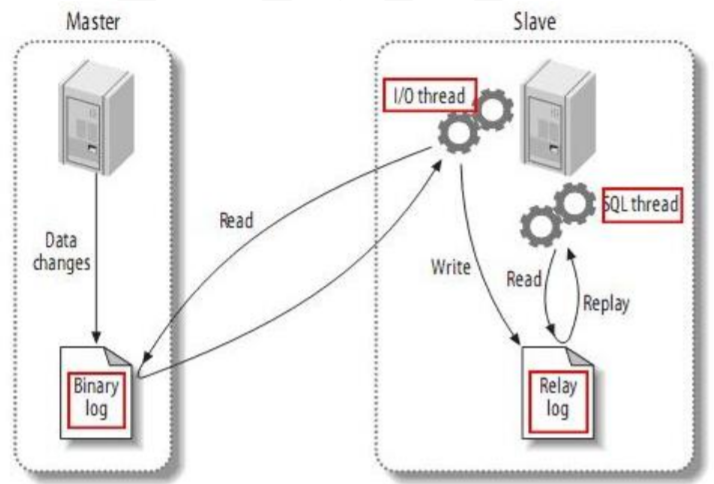
PS：从图中可以看出，Slave 服务器中有一个I/O线程(I/O Thread)在不停地监听 Master的二进制日志(Binary Log)是否有更新：如果没有它会睡眠等待 Master 产生新的日志事件；如果有新的日志事件(Log Events)，则会将其拷贝至 Slave 服务器中的中继日志(RelayLog)。

(2).slave 将 master 的二进制日志事件（binary log events）拷贝到它的中继日志(relaylog)

(3).slave 重做中继日志中的事件,将 Master 上的改变反映到它自己的数据库中。，所以两端的数据是完全一样的。

PS：从图中可以看出，Slave 服务器中有一个 SQL 线程(SQL Thread)从中继日志读取事件，并重做其中的事件，从而更新 Slave 的数据，使其与 Master 中的数据一致。只要该线程与 I/O 线程保持一致，中继日志通常会位于 OS 的缓存中，所以中继日志的开销很小。

附简要原理图：



主从复制的几种方式：

1.同步复制

主服务器在将更新的数据写入它的二进制日志（Binlog）文件中后，必须等待验证所有的从服务器的更新数据是否已经复制到其中，之后才可以自由处理其它进入的事务处理请求。

2.异步复制

主服务器在将更新的数据写入它的二进制日志（Binlog）文件中后，无需等待验证更新数据是否已经复制到从服务器中，就可以自由处理其它进入的事务处理请求。

3.半同步复制

主服务器在将更新的数据写入它的二进制日志（Binlog）文件中后，只需等待验证其中一台从服务器的更新数据是否已经复制到其中，就可以自由处理其它进入的事务处理请求，其他的从服务器不用管。

3.数据库分表，分区，分库

分表见上面描述。

分区就是把一张表的数据分成多个区块，这些区块可以在一个磁盘上，也可以在不同的磁盘上，分区后，表面上还是一张表，但数据散列在多个位置，这样一来，多块硬盘同时处理不同的请求，从而提高磁盘 I/O 读写性能，实现比较简单。 包括水平分区和垂直分区。

分库是根据业务不同把相关的表切分到不同的数据库中，比如 web、bbs、blog 等库。

**sql 注入的问题**

一．sql 语句应该考虑哪些安全性？

1.防止 sql 注入，对特殊字符进行过滤、转义或者使用预编译的 sql 语句绑定变量。

2.当 sql 语句运行出错时，不要把数据库返回的错误信息全部显示给用户，以防止泄漏服务器和数据库相关信息。 二．什么叫做 SQL 注入，如何防止？请举例说明。

举个例子：

你后台写的 java 代码拼的 sql 如下：

// 该 ename 为前台传过来的一个查询条件

public List getInfo(String ename){

StringBuffer buf = new StringBuffer();

buf.append("select empno,ename,deptno from emp where ename =

'").append(ename).append("'");

...

...

... }

而前台页面有个输入框如下：

职员姓名：\_\_\_\_\_\_\_\_\_\_ 该文本域对应上面方法的 ename 参数。

如果用户在查询时向职员姓名文本域中输入的是如下信息：

' or '1'='1

那么这时就会涉及到 sql 注入这个概念了。

上面的字符串传到后台后，与其它 select 等字符串拼成了如下的语句：

select empno,ename,deptno from emp where ename = '' or '1'='1' 上面的 where 条件是永远成立的，如果你的表中有权限限制，比如只能查询本地市的信息，过滤条件中有地市过滤，不过因为输入 ' or '1'='1 字符串后，条件永远成立，导致你能看到所有城市的职员信息，那就会产生权限问题了，用户能看到不该看到的信息。同理，如果是 insert 或者 update 等语句的话，通过 sql 注入会产生不可估量的问题。

这种不安全的情况是在 SQL 语句在拼接的情况下发生。

解决方法：

1.参数绑定。为了防范这样”SQL 注入安全“可以用预编译解决（不要用拼接 SQL 字符串,可以用 prepareStatement,参数用 set 方法进行填装 ）。

String sql= "insert into userlogin values(?,?)";

try {

PreparedStatement ps=conn.prepareStatement(sql);

for(int i=1;i<100;i++){

ps.setInt(1, i);

ps.setInt(2, 8888);

ps.executeUpdate();

}

ps.close();

conn.close();

} catch (SQLException e) {

e.printStackTrace();

}

2、检查变量的数据类型和格式

如果你的 SQL 语句是类似 where id={$id}这种形式，数据库里所有的 id 都是数字，那么就应该在 SQL 被执行前，检查确保变量 id 是 int 类型；如果是接受邮箱，那就应该检查并严格确保变量一定是邮箱的格式，其他的类型比如日期、时间等也是一个道理。总结起来：只要是有固定格式的变量，在 SQL 语句执行前，应该严格按照固定格式去检查，确保变量是我们预想的格式，这样很大程度上可以避免 SQL 注入攻击。

比如，我们前面接受 username 参数例子中，我们的产品设计应该是在用户注册的一开始，就有一个用户名的命名规则，比如 5-20 个字符，只能由大小写字母、数字以及一些安全的符号组成，不包含特殊字符。此时我们应该有一个check\_username 的函数来进行统一的检查。不过，仍然有很多例外情况并不能应用到这一准则，比如文章发布系统，评论系统等必须要允许用户提交任意字符串的场景，这就需要采用过滤等其他方案了。（使用正则表达式进行格式验证！）

3.所有的 SQL 语句都封装在存储过程中。

所有的 SQL 语句都封装在存储过程中，这样不但可以避免 SQL 注入，还能提高一些性能。 涉及连接的问题

**一．内连接和外连接的区别？**

为了说明问题，定义如下 2 个表。

[TEST1@orcl#16-12 月-11] SQL>select \* from t1;

ID NAME

- --------- --------------------

1 aaa

2 bbb

[TEST1@orcl#16-12 月-11] SQL>select \* from t2;

ID AGE

---------- ----------

1 20

3 30

内连接（inner join）：只显示符合连接条件的记录。

[TEST1@orcl#16-12 月-11] SQL>select \* from t1 inner join t2 on t1.id=t2.id;

ID NAME ID AGE

- - ----------- -------------------- ---------- --------------------------------

1 aaa 1 20

外连接分左外连接、右外连接、全外连接三种。

1）左外连接（LEFT JOIN 或 LEFT OUTER JOIN ）

即以左表为基准，到右表找匹配的数据，找不到匹配的用 NULL 补齐。

显示左表的全部记录及右表符合连接条件的记录。

[TEST1@orcl#16-12 月-11] SQL>select \* from t1 left join t2 on t1.id=t2.id;

ID NAME ID AGE

---------- -------------------- ---------- ----------

1 aaa 1 20

2 bbb NULL NULL

2）右外连接（RIGHT JOIN 或 RIGHT OUTER JOIN ）

[TEST1@orcl#16-12 月-11] SQL>select \* from t1 right join t2 on t1.id=t2.id;

ID NAME ID AGE

---------- -------------------- ---------- ----------

1 aaa 1 20

NULL NULL 3 30

即以右表为基准，到左表找匹配的数据，找不到匹配的用 NULL 补齐。

显示右表的全部记录及左表符合连接条件的记录。

3）全外连接（FULL JOIN 或 FULL OUTER JOIN）

除了显示符合连接条件的记录外，在 2 个表中的其他记录也显示出来。

**二．inner join 和 left join 的性能比较。**

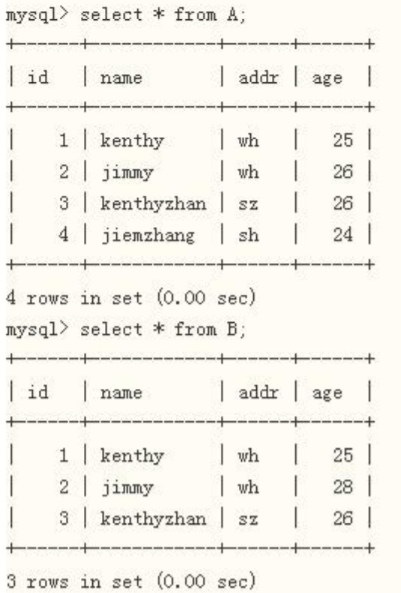
从理论上来分析，确实是 inner join 的性能要好，因为是选出 2 个表都有的记录，而 left join 会出来左边表的所有记录、满足 on 条件的右边表的记录。

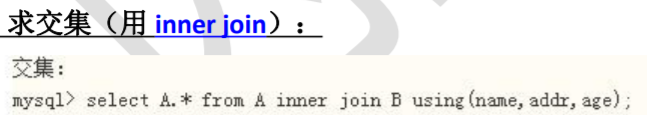
1.在解析阶段，左连接是内连接的下一阶段，内连接结束后，把存在于左输入而未存在于右输入的集，加回总的结果集，因此如果少了这一步效率应该要高些。

2.在编译的优化阶段，如果左连接的结果集和内连接一样时，左连接查询会转换成内连接查询，即编译优化器认为内连接要比左连接高效。

**三．联合查询的索引使用。**

在 where 子句中要加筛选条件，才可以都用上索引。 四．数据库中两个表求交集、并集、差集。





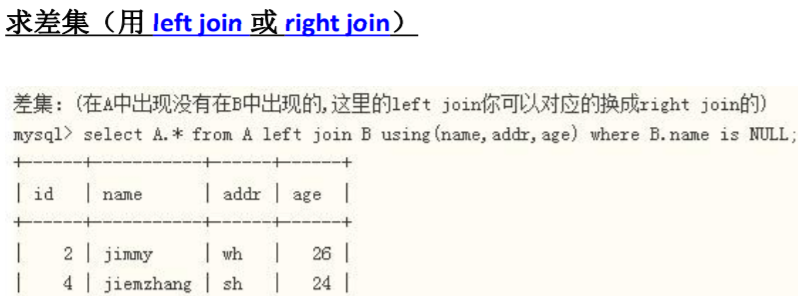
using(column\_list)：其作用是为了方便书写联结的多对应关系，大部分情况下 USING 语

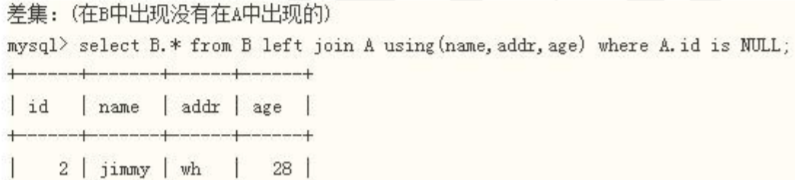
句可以用 ON 语句来代替，如下面例子：

a LEFT JOIN b USING (c1,c2,c3)，其作用相当于下面语句

a LEFT JOIN b ON a.c1=b.c1 AND a.c2=b.c2 AND a.c3=b.c3

只是用 ON 来代替会书写比较麻烦而已。





求并集（用 union）

1. **存储过程的概念以及优缺点是什么。**

存储过程：就是一些编译好了的 sql 语句，这些 SQL 语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。

优点：

1.存储过程因为 SQL 语句已经预编译过了，因此运行的速度比较快。

2.存储过程在服务器端运行，减少客户端的压力。

3.允许模块化程序设计，就是说只需要创建一次过程，以后在程序中就可以调用该过程任意次，类似方法的复用。

4.减少网络流量，客户端调用存储过程只需要传存储过程名和相关参数即可，与传输 SQL 语句相比自然数据量少了很多。

5. 增强了使用的安全性，充分利用系统管理员可以对执行的某一个存储过程进行权限限制，从而能够实现对某些数据访问的限制，避免非授权用户对数据的访问，保证数据的安全。程序员直接调用存储过程，根本不知道表结构是什么，有什么字段，没有直接暴露表名以及字段名给程序员。

缺点：

调试麻烦（至少没有像开发程序那样容易），可移植性不灵活（因为存储过程是依赖于具体的数据库）。