

数据库

Part-----数据库专题

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 * from news;`

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

根据一行或者多行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。

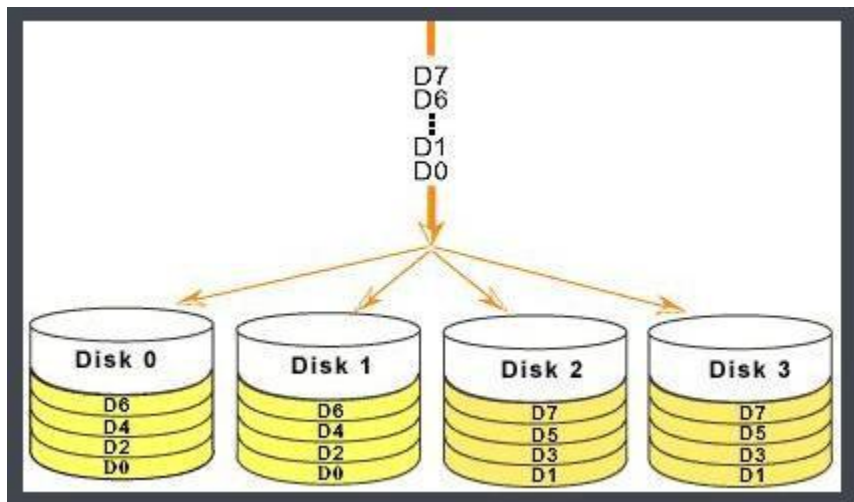
4.3 磁盘 I/O 的优化

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阵列。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？

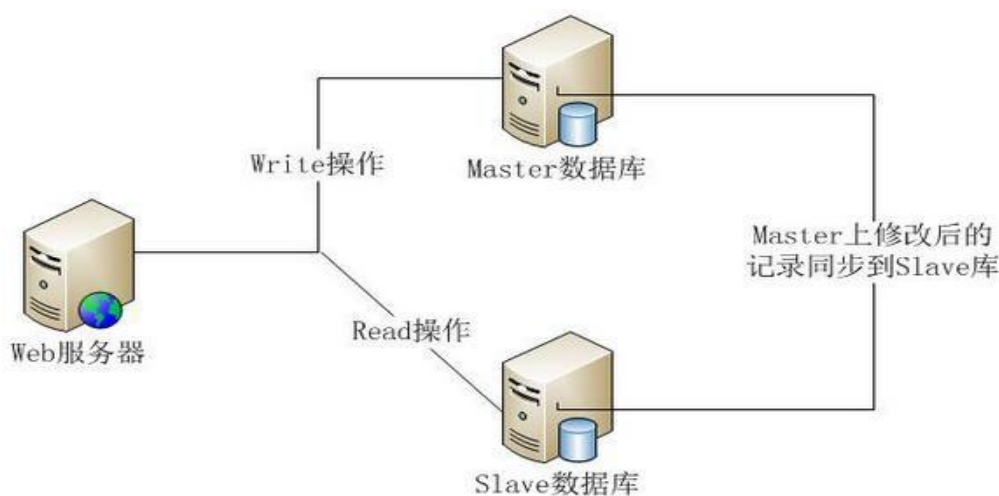
1.使用优化查询的方法（见上面）

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。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】



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

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：

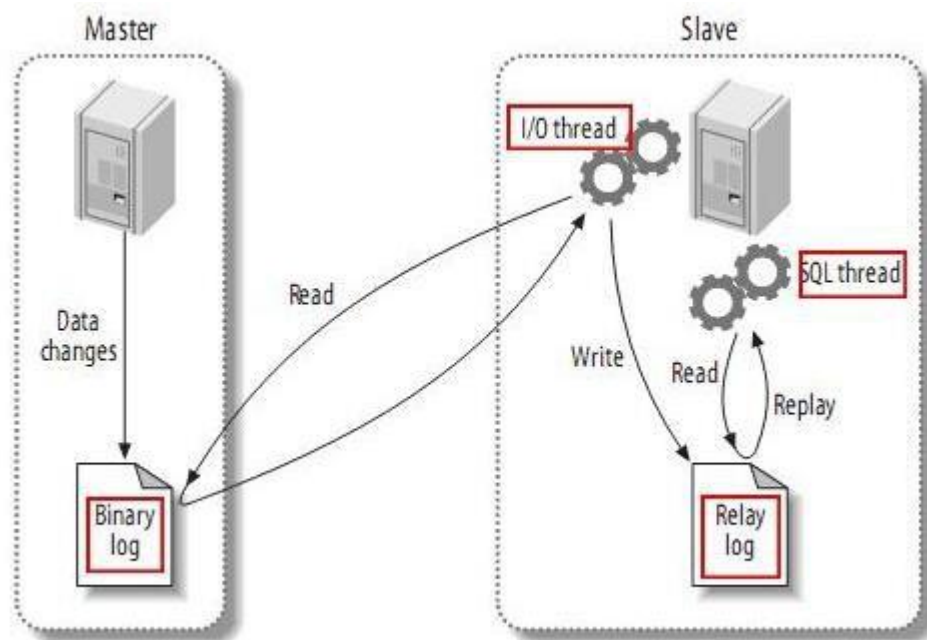
(1)master 将**数据改变**记录到**二进制日志(binary log)**中,也即是配置文件 log-bin 指定的文件(这些记录叫做二进制日志事件，binary log events)

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(Relay Log)。

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

(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 preparedStatement,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@orc#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。

```
mysql> select * from A;
```

id	name	addr	age
1	kenthy	wh	25
2	jimmy	wh	26
3	kenthyzhan	sz	26
4	jiemzhang	sh	24

```
4 rows in set (0.00 sec)
```

```
mysql> select * from B;
```

id	name	addr	age
1	kenthy	wh	25
2	jimmy	wh	28
3	kenthyzhan	sz	26

```
3 rows in set (0.00 sec)
```

求交集（用 `inner join`）：

交集：

```
mysql> select A.* from A inner join B using(name, addr, age);
```

id	name	addr	age
1	kenthy	wh	25
3	kenthyzhan	sz	26

```
2 rows in set (0.00 sec)
```

`using(column_list)`：其作用是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。

求差集（用 [left join](#) 或 [right join](#)）

差集：（在A中出现没有在B中出现的,这里的left join你可以对应的换成right join的）
mysql> select A.* from A left join B using(name,addr,age) where B.name is NULL;

id	name	addr	age
2	jinmy	wh	26
4	jiemzhang	sh	24

2 rows in set (0.00 sec)

差集：（在B中出现没有在A中出现的）

mysql> select B.* from B left join A using(name,addr,age) where A.id is NULL;

id	name	addr	age
2	jinmy	wh	28

1 row in set (0.00 sec)

求并集（用 [union](#)）

涉及存储过程的问题

一、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）。

涉及范式的问题

一. 数据库的三级范式？

1NF: 字段不可再分，原子性。

2NF：满足第二范式（2NF）必须先满足第一范式（1NF）。

一个表只能说明一个事物。非主键属性必须**完全依赖**于主键属性。

3NF: 满足第三范式（3NF）必须先满足第二范式（2NF）。每列都**与主键**有直接关系 **不存在传递依赖**。任何非主属性不依赖于其它非主属性。

不符合第一范式的例子(关系数据库中 create 不出这样的表)：

表: 字段 1, 字段 2(字段 2.1, 字段 2.2), 字段 3

不符合第二范式的例子:

表: 学号, 姓名, 年龄, 课程名称, 成绩, 学分;

这个表明显说明了两个事物: 学生信息, 课程信息。

不符合第三范式的例子:

学号, 姓名, 年龄, 所在学院, 学院地点, 学院联系电话, 主键为"

学号 " ;

存在依赖传递: (学号) → (所在学院) → (学院地点, 学院电话)

涉及事务的问题

1. 数据库事务正确执行的四个基本要素（事务的 4 个属性）.

事务是由一组 SQL 语句组成的逻辑处理单元。

ACID: 原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)。

原子性: 一个事务(transaction)中的所有操作, 要么全部完成, 要么全部不完成, 不会结束在中间某个环节。事务在执行过程中发生错误, 会被 **回滚** (Rollback) 到事务开始前的状态, 就像这个事务从来没有执行过一样。

一致性: 在事务开始和完成时, 数据库中的数据都保持一致的状态, 数据的**完整性约束**没有被破坏。(事务的执行使得数据库**从一种正确状态转换成另一种正确状态**)。具体来说就是, 比如表与表之间存在外键约束关系, 那么你对数据库进行的修改操作就必须要满足约束条件, 即如果你修改了一张表中的数据, 那你还需要修改与之存在外键约束关系的其他表中对应的数据, 以达到一致性。

隔离性: 一个事务的执行不能被其他事务干扰。为了防止事务操作间的混淆, 必须串行化或序列化请求, 使得在同一时间仅有一个请求用于同一数据。(**在事务正确提交之前**, 不允许把该事务**对数据的任何改变**提供给任何其他事务)。(事务处理过程中的中间状态对外部是不可见的)。隔离性通过**锁**就可以实现。

持久性: 一个事务一旦提交, 它对数据库中数据的改变就应该是永久性的, 并不会被回滚。

2.并发事务带来的问题。

1. 更新丢失。

两个事务 T1 和 T2 读入同一数据并修改, T2 提交的结果覆盖了 T1 提交的结果, 导致 T1 的修改被丢失。

2. 脏读。

事务 T1 修改某一数据 并将其写回磁盘 事务 T2 读取同一数据后 T1 由于某种原因被撤销, 这时 T1 已修改过的数据恢复原值, T2 读到的数据就与数据库中的数据不一致, 则 T2 读到的数据就为“脏”数据, 即不正确的数据。

例如:

张三的工资为 5000, 事务 A 中把他的工资改为 8000, 但事务 A 尚未提交。与此同时, 事务 B 正在读取张三的工资, 读取到张三的工资为 8000。随后, 事务 A 发生异常, 而回滚了事务。张三的工资又回滚为 5000。最后, 事务 B 读取到的张三工资为 8000 的数据即为脏数据, 事务 B 做了一次脏读。

3. 不可重复读。

是指在一个事务内, 多次读同一数据。在这个事务还没有结束时, 另外一个事务也访问该同一数据。那么, 在第一个事务中的两次读数据之间, 由于第二个事务的修改, 那么第一个事务两次读到的数据可能是不一样的。这样就发生了在一个事务内两次相同的查询读到的数据是不一样的, 因此称为是不可重复读。

例如：

在事务 A 中，读取到张三的工资为 5000，操作没有完成，事务还没提交。与此同时，事务 B 把张三的工资改为 8000，并提交了事务。随后，在事务 A 中，再次读取张三的工资，此时工资变为 8000。在一个事务中前后两次读取的结果并不致，导致了不可重复读。

4. 幻读。

例如

目前工资为 5000 的员工有 10 人，事务 A 读取所有工资为 5000 的人数为 10 人。此时，事务 B 插入一条工资也为 5000 的记录。这是，事务 A 再次读取工资为 5000 的员工，记录为 11 人。此时产生了幻读。

不可重复读的重点是修改：

同样的条件，你读取过的数据，再次读取出来发现值不一样了。

幻读的重点在于新增或者删除：

同样的条件，第 1 次和第 2 次读出来的记录数不一样。

避免不一致性的方法和技术就是并发控制。最常用的技术是封锁技术。

3. 并发控制的方式（如何解决并发问题）。

加锁，如乐观锁和悲观锁。

4. 数据库事务的隔离级别介绍、举例说明。

数据库事务的隔离性：数据库系统必须具有隔离并发运行各个事务的能力，使它们不会相互影响，避免各种并发问题。

数据库提供了 4 种隔离级别（由低到高）：

这 4 个级别可逐个解决脏读，不可重复读和幻读这几个问题。

1. 读未提交数据

允许事务读取未被其他事务提交的变更，可能有脏读，不可重复读和幻读的问题。

比如：某时刻会话 a 修改了一个数据，但还未提交，此时会话 b 读取了该数据，这是，会话 a 回滚了事务，这就导致数据出现了不一致状态，这就是脏读。

2. 读已提交数据

允许事务读取已经被其他事务提交的变更，可以避免脏读，可能有不可重复读和幻读的问题。

例如：某时刻会话 a 的一个事务里查询一个数据，得到的数据是 1，这时会话 b 修改了该数据的值为 2，并提交了，此时会话 a 的事务又要读取该数据，这时的数据是 2，这样就出现了同一个事务内，读的结果不一样，这就是不可重复读。

不可重复读，是指在数据库访问中，一个**事务**范围内两个相同的查询却返回了不同数据。

3. 可重复读（Mysql 的默认隔离级别）

确保事务可以多次从一个字段中读取相同的值，在这个事务持续期间，禁止其他事务对这个字段进行更新，可以避免脏读和不可重复读，可能会有幻读。

4. 可串行化

所有事务都一个接一个地串行执行。可以避免脏读，不可重复读，幻读。

✓：可能出现 ×：不会出现

 快速回顾

	脏读	不可重复读	幻读
Read uncommitted	✓	✓	✓
Read committed	×	✓	✓
Repeatable read	×	×	✓
Serializable	×	×	×

5. MySQL 事务控制语句

BEGIN 或 START TRANSACTION: 显式地开启一个事务；

COMMIT: 提交事务，并使已对数据库进行的所有修改称为永久性的；

ROLLBACK: 回滚会结束用户的事务，并撤销正在进行的所有未提交的修改；

6. 数据库怎么保证数据的一致性。

事务，悲观锁，乐观锁。

涉及锁的问题

一．如何并发访问数据库。

加锁。

二．说下数据库的锁机制，数据库中都有哪些锁。

锁是一种并发控制技术，锁是用来在多个用户同时访问同一个数据的时候保护数据的。

1. 有 2 种基本的锁类型：

共享 (S)锁：多个事务可封锁一个共享页；任何事务都不能修改该页；通常是该页被读取完毕，S 锁立即被释放。在执行 `select` 语句的时候需要给操作对象（表或者一些记录）加上共享锁，但加锁之前需要检查是否有排他锁，如果没有，则可以加共享锁（一个对象上可以加 n 个共享锁），否则不行。共享锁通常在执行完 `select` 语句之后被释放，当然也有可能是在事务结束（包括正常结束和异常结束）的时候被释放，主要取决于数据库所设置的事务隔离级别。

排它 (X)锁：仅允许一个事务封锁此页；其他任何事务必须等到 X 锁被释放才能对该页进行访问；X 锁一直到事务结束才能被释放。执行 `insert`、`update`、`delete` 语句的时候需要给操作的对象加排他锁，在加排他锁之前必须确认该对象上没有其他任何锁，一旦加上排他锁之后，就不能再给这个对象加其他任何锁。排他锁的释放通常是在事务结束的时候（当然也有例外，就是在数据库事务隔离级别被设置成 `Read Uncommitted`（读未提交数据）的时候，这种情况下排他锁会在执行完更新操作之后就释放，而不是在事务结束的时候）。

2. 按锁的粒度

3. 按锁的机制

既然使用了锁，就有出现死锁的可能。

产生死锁的四个必要条件：

- (1) 互斥条件：一个资源每次只能被一个进程使用。
- (2) 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- (3) 不可剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。
- (4) 环路等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

只要系统发生了死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

1. 预防死锁

预防死锁的发生只需破坏死锁产生的四个必要条件之一即可。

1) 破坏互斥条件

如果允许系统资源都能共享使用，则系统不会进入死锁状态。但有些资源根本不能同时访问，如打印机等临界资源只能互斥使用。所以，破坏互斥条件而预防死锁的方法不太可行，而且在有的场合应该保护这种互斥性。

2) 破坏不剥夺条件

当一个已保持了某些不可剥夺资源的进程，请求新的资源而得不到满足时，它必须释放已经保持的所有资源，待以后需要时再重新申请。这意味着，一个进程已占有的资源会被暂时释放，或者说是被剥夺了，或从而破坏了不可剥夺条件。

该策略实现起来比较复杂，释放已获得的资源可能造成前一阶段工作的失效，反复地申请和释放资源会增加系统开销，降低系统吞吐量。这种方法常用于状态易于保存和恢复的资源，如 CPU 的寄存器及内存资源，一般不能用于打印机之类的资源。

3) 破坏请求和保持条件

采用预先静态分配方法，即进程在运行前一次申请完它所需要的全部资源，在它的资源未满足前，不把它投入运行。一旦投入运行后，这些资源就一直归它所有，也不再提出其他资源请求，这样就可以保证系统不会发生死锁。

这种方式实现简单，但缺点也显而易见，系统资源被严重浪费，其中有些资源可能仅在运行初期或运行快结束时才使用，甚至根本不使用。而且还会导致“饥饿”现象，当由于个别资源长期被其他进程占用时，将致使等待该资源的进程迟迟不能开始运行。

4) 破坏环路等待条件

为了破坏循环等待条件，可采用顺序资源分配法。首先给系统中的资源编号，规定每个进程，必须按编号递增的顺序请求资源，同类资源一次申请完。也就是说，只要进程提出申请分配资源 R_i ，则该进程在以后的资源申请中，只能申请编号大于 R_i 的资源。

这种方法存在的问题是，编号必须相对稳定，这就限制了新类型设备的增加；尽管在为资源编号时已考虑到大多数作业实际使用这些资源的顺序，但也经常会发生作业使用资源的顺序与系统规定顺序不同的情况，造成资源的浪费；此外，这种按规定次序申请资源的方法，也必然会给用户的编程带来麻烦。

2. 避免死锁

银行家算法。

3. 检测死锁

死锁定理。

4. 解除死锁

3.1 从死锁进程处剥夺资源

3.2 终止部分或全部进程

三. mysql 锁的粒度（即锁的级别）。

MySQL 各存储引擎使用了三种类型（级别）的锁定机制：行级锁定，页级锁定和表级锁定。

1.表级锁，直接锁定整张表，在你锁定期间，其它进程无法对该表进行写操作。如果你是写锁，则其它进程则读也不允许。特点：开销小，加锁快；不会出现死锁；锁定粒度最大，发生锁冲突的概率最高，并发度最低。

MyISAM 存储引擎采用的是表级锁。

有 2 种模式：表共享读锁和表独占写锁。加读锁的命令：`lock table 表名 read`；去掉锁的命令：`unlock tables`。

支持并发插入：支持查询和插入操作并发进行（在表尾并发插入）。

锁调度机制：写锁优先。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。

2.行级锁，仅对指定的记录进行加锁，这样其它进程还是可以对同一个表中的其它记

录进行操作。特点：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。

InnoDB 存储引擎既支持行级锁，也支持表级锁，但默认情况下是采用行级锁。

3.页级锁，一次锁定相邻的一组记录。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。

最常用的处理多用户并发访问的方法是加锁。当一个用户锁住数据库中的某个对象时，其他用户就不能再访问该对象。加锁对并发访问的影响体现在锁的粒度上。比如，(表锁)放在一个表上的锁限制对整个表的并发访问；（页锁）放在数据页上的锁限制了对整个数据页的访问；（行锁）放在行上的锁只限制对该行的并发访问。

四．乐观锁和悲观锁的概念，实现方式和适用场景。

锁有两种机制：悲观锁和乐观锁。

悲观锁，锁如其名，他对世界是悲观的，他认为别人访问正在改变的数据的概率是很高的，所以从数据开始更改时就将数据锁住，直到更改完成才释放。

一个典型的倚赖数据库的悲观锁调用：

```
select * from account where name="Erica" for update
```

这条 sql 语句锁定了 **account** 表中所有符合检索条件（**name="Erica"**）的记录。本次事务提交之前（事务提交时会释放事务过程中的锁），外界无法修改这些记录。该语句用来锁定特定的行（如果有 where 子句，就是满足 where 条件的那些行）。当这些行被锁定后，其他会话可以选择这些行，但不能更改或删除这些行 直到该语句的事务被 **commit** 语句或 **rollback** 语句结束为止。需要注意的是，**selectfor update** 要放到 **mysql** 的事务中，即 **begin** 和 **commit** 中，否则不起作用。

为了更好的理解 **select..... for update** 的锁表的过程，本人将要以 **mysql** 为例，进行相应的讲解。

1、要测试锁定的状况，可以利用 **MySQL** 的 **Command Mode**，开二个视窗来做测试。

表的基本结构如下：

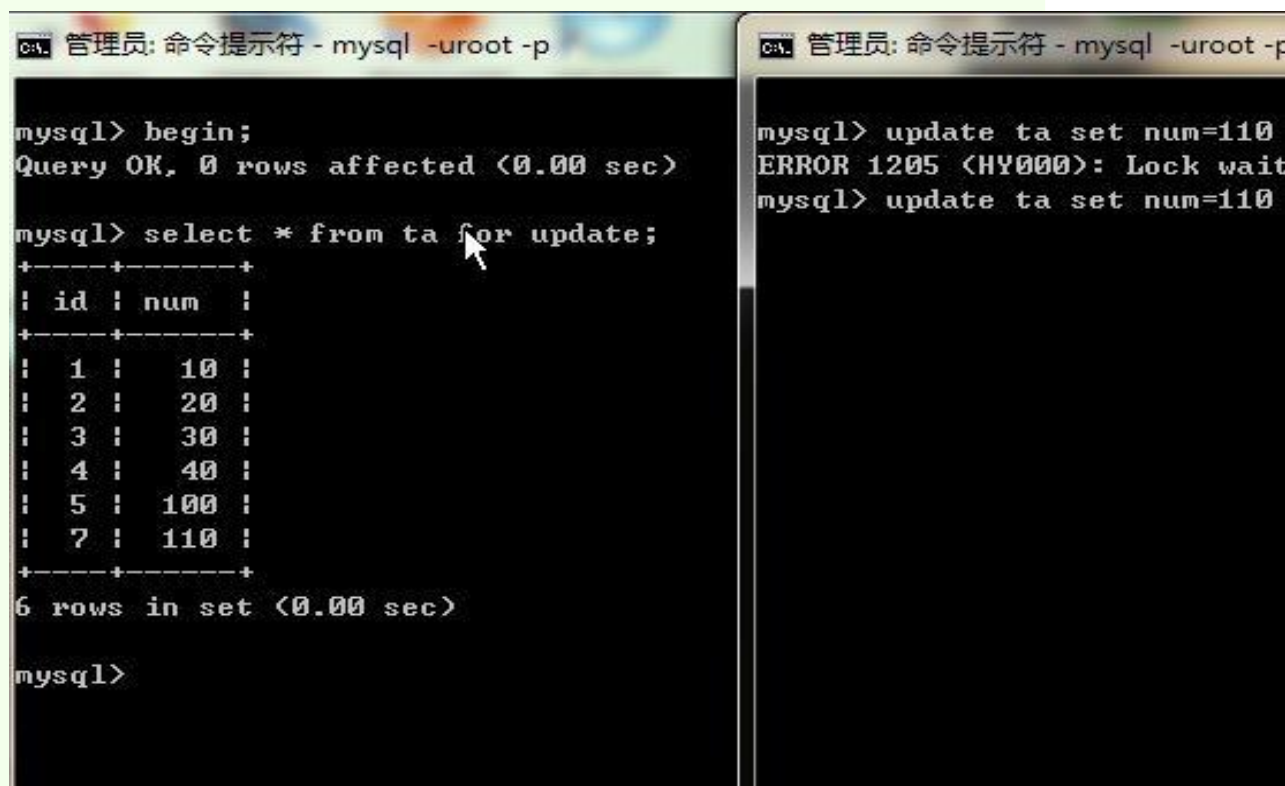

```
mysql> desc ta;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)| NO   | PRI | NULL    |       |
| num   | int(11)| YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.02 sec)
```

表中内容如下:

```
mysql> select * from ta;
+-----+-----+
| id | num |
+-----+-----+
| 1  | 10  |
| 2  | 20  |
| 3  | 30  |
| 4  | 40  |
| 5  | 100 |
| 7  | 100 |
+-----+-----+
6 rows in set (0.00 sec)
```

开启两个测试窗口，在其中一个窗口执行 `select * from ta for update;`

然后在另外一个窗口执行 `update` 操作如下图:



```

CA. 管理员: 命令提示符 - mysql -uroot -p

mysql> begin;
Query OK, 0 rows affected (0.00 sec)

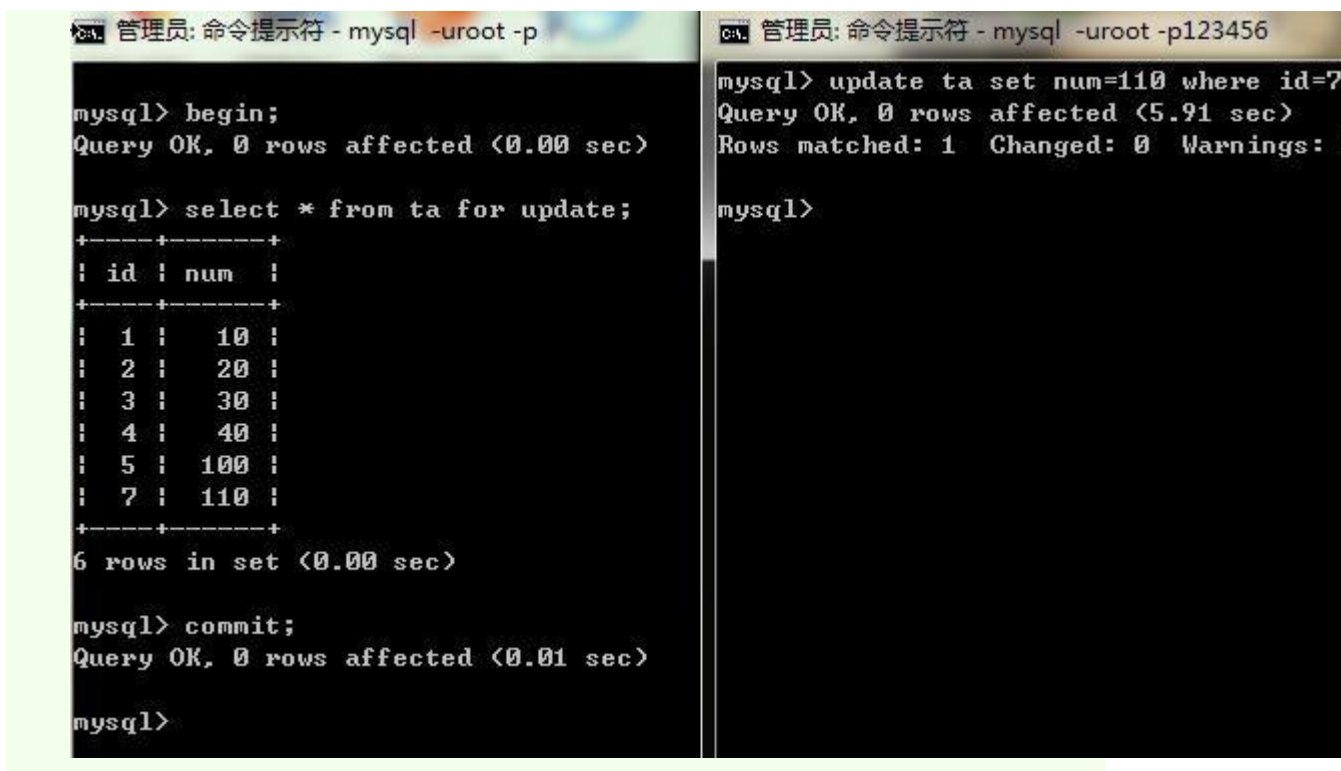
mysql> select * from ta for update;
+-----+-----+
| id | num |
+-----+-----+
| 1  | 10  |
| 2  | 20  |
| 3  | 30  |
| 4  | 40  |
| 5  | 100 |
| 7  | 110 |
+-----+-----+
6 rows in set (0.00 sec)

mysql>

CA. 管理员: 命令提示符 - mysql -uroot -p

mysql> update ta set num=110;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
mysql> update ta set num=110;
```

等到一个窗口 commit 后的图片如下：



```
mysql> begin;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from ta for update;
+----+-----+
| id | num |
+----+-----+
| 1  | 10  |
| 2  | 20  |
| 3  | 30  |
| 4  | 40  |
| 5  | 100 |
| 7  | 110 |
+----+-----+
6 rows in set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql>

mysql> update ta set num=110 where id=7;
Query OK, 0 rows affected (5.91 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql>
```

悲观锁可能会造成加锁的时间很长，并发性不好，特别是长事务，影响系统的整体性能。

悲观锁的实现方式：

悲观锁，也是基于数据库的锁机制实现。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先

上锁。

乐观锁，他对世界比较乐观，认为别人访问正在改变的数据的概率是很低的，所以直到修改完成准备提交所做的修改到数据库的时候才会将数据锁住，当你读取以及改变该对象时并不加锁，完成更改后释放。乐观锁不能解决脏读的问题。

乐观锁加锁的时间要比悲观锁短，大大提升了大并发量下的系统整体性能表现。

乐观锁的实现方式：

1. 大多是基于数据版本 (Version) 记录机制实现, 需要为每一行数据增加一个版本标识 (也就是每一行数据多一个字段 version), 每次更新数据都要更新对应的版本号+1。

工作原理: 读出数据时, 将此版本号一同读出, 之后更新时, 对此版本号加一。此时, 将提交数据的版本信息与数据库表对应记录的当前版本信息进行比较, 如果提交的数据版本号大于数据库表当前版本号, 则予以更新, 否则认为是过期数据, 不得不重新读取该对象并作出更改。

假设数据库中帐户信息表中有一个 version 字段, 当前值为 1; 而当前帐户余额字段 (balance) 为 \$100。

- 1 操作员 A 此时将其读出 (version=1), 并从其帐户余额中扣除 \$5 (\$100-\$50)。
- 2 在操作员 A 操作的过程中, 操作员 B 也读入此用户信息 (version=1), 并从其帐户余额中扣除 \$20 (\$100-\$20)。
- 3 操作员 A 完成了修改工作, 将数据版本号加一 (version=2), 连同帐户扣除后余额 (balance=\$50), 提交至数据库更新, 此时由于提交数据版本大于数据库记录当前版本, 数据被更新, 数据库记录 version 更新为 2。
- 4 操作员 B 完成了操作, 也将版本号加一 (version=2) 试图向数据库提交数据 (balance=\$80), 但此时比对数据库记录版本时发现, 操作员 B 提交的数据版本号为 2, 数据库记录当前版本也为 2, 不满足“提交版本必须大于记录当前版本才能执行更新”的乐观锁策略, 因此, 操作员 B 的提交被驳回。这样, 就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员 A 的操作结果的可能。

从上面的例子可以看出, 乐观锁机制避免了长事务中的数据库加锁开销 (操作员 A 和操作员 B 操作过程中, 都没有对数据库数据加锁), 大大提升了大并发量下的系统整体性能表现。

2. 使用时间戳来实现

同样是在需要乐观锁控制的 table 中增加一个字段, 名称无所谓, 字段类型使用时间戳 (timestamp), 和上面的 version 类似, 也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比, 如果一致则 OK, 否则就是版本冲突。

悲观锁和乐观锁的适用场景:

如果并发量不大, 可以使用悲观锁解决并发问题; 但如果系统的并发量非

常大的话,悲观锁定会带来非常大的性能问题,所以我们就要选择乐观锁定的方法。现在大部分应用都应该是乐观锁的。

涉及命令的问题

一 . truncate 与 delete 的区别是什么？

TRUNCATE TABLE: 删除内容、不删除定义、释放空间。

DELETE TABLE:删除内容、不删除定义，不释放空间。

DROP TABLE: 删除内容和定义，释放空间。

1.truncate table 表名，只能删除表中全部数据。

delete from 表名 where.....，可以删除表中全部数据，也可以删除部分数据。

2.delete from 记录是一条条删的，所删除的每行记录都会进日志，而 truncate 一次性删掉整个页，因此日志里面只记录页释放。

3.Truncate 删除后，不能回滚。Delete 可以回滚。

4.Truncate 的执行速度比 Delete 快。

5.Delete 执行后，删除的数据占用的存储空间还在，还可以恢复数据。

Truncate 删除的数据占用的存储空间不在，不可以恢复数据。

二 . sql 的授权语句和收回权限语句？

grant 权限 on 数据库对象 to 用户

grant insert on scott.Employees to user1,user2;

grant delete on scott.Employees to user1,user2;

```
grant select on scott.Employees to user1,user2;  
grant update on scott.Employees to user1,user2;
```

revoke 权限 on 数据库对象 from 用户

三 . 怎么新加一行记录、怎么添加一个列字段 , 修改列 ?

插入一行数据:

```
insert into stu(stuName,stuAge,stuSex) values('张三','20','男')
```

增加列:

1. **alter table** tableName **add** (column) columnName **varchar(30)** (一定要有数据类型!!!)

删除列:

1. **alter table** tableName **drop** (column) columnName

四. **select Count (*)** 和 **Select Count(数字)** 以及 **Select Count(column)**区别。

count(*) 跟 count(1) 的结果一样, 返回记录的总行数, 都对 NULL 的统计, 而count(column) 是不包括NULL 的统计。

```
mysql> select * from ab;
+-----+
| id | name |
+-----+
| 1 | 2 |
| 2 | NULL |
+-----+
2 rows in set (0.00 sec)

mysql> select count(*) from ab;
+-----+
| count(*) |
+-----+
| 2 |
+-----+
1 row in set (0.00 sec)

mysql> select count(name) from ab;
+-----+
| count(name) |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)

mysql> select count(0) from ab;
+-----+
| count(0) |
+-----+
| 2 |
+-----+
1 row in set (0.00 sec)
```

五. EXISTS 关键字的使用方法？

EXISTS 关键字表示存在。使用 EXISTS 关键字时，内层查询语句不返回查询的记录，而是返回一个真假值。

如果内层查询语句查询到符合条件的记录，就返回一个真值（true），否则，将返回一个假值（false）：

当返回的值为 true 时，外层查询语句将进行查询。

当返回的值为 false 时，外层查询语句将不进行查询或者查询不出任何记录。

实例 1

如果 department 表中存在 d_id 取值为 1003 的记录，则查询 employee 表的记录。SELECT 语句的代码如下：

```
SELECT * FROM employee

        WHERE EXISTS

                (SELECT d_name FROM department WHERE d_id=1003);
```

因为 department 表中存在 d_id 值为 1003 的记录，内层查询语句返回一个 true，外层查询语句接收 true 后，开始查询 employee 表中的记录。因为没有设置查询 employee 表的查询条件，所以查询出了 employee 表的所有记录。

实例 2

EXISTS 关键字可以与其它查询条件一起使用。条件表达式与 EXISTS 关键字之间用 AND 或者 OR 进行连接。

如果 department 表中存在 d_id 取值为 1003 的记录，则查询 employee 表中 age 大于 24 岁的记录。SELECT 语句的代码如下：

```
SELECT * FROM employee

        WHERE age>24 AND EXISTS

                (SELECT d name FROM department WHERE d id=1003);
```

因为，当内层查询语句从 department 表中查询到记录，返回一个 true，外层查询语句开始进行查询，根据查询条件，从 employee 表中查询出 age 大于 24 岁的两条记录。

实例 3

NOT EXISTS 与 EXISTS 正好相反。使用 NOT EXISTS 关键字时，当返回的值是 true 时，外层查询语句不进行查询或者查询不出任何记录；当返回值是 false 时，外层查询语句将进行查询。

如果 department 表中不存在 d_id 字段取值为 1003 的记录，则查询 employee 表的记录。SELECT 语句的代码如下：

```
SELECT * FROM employee

        WHERE NOT EXISTS

                (SELECT d_name FROM department WHERE d_id=1003);
```

六.有一个学生表，有三个字段：**name**、**course**、**score**，
每一个学生都有三门课程，比如数学、语文、英语，写 **sql** 语句，查找出三门课程的成绩都大于 **80** 的学生。

采用逆向思维。求三门成绩都大于 80 的人，也可以是使先查出有成绩小于 80 的人，再除去这些人就是三门成绩都大于 80 的人了。

方法 1:

```
SELECT DISTINCT A.name  
FROM Student A  
WHERE A.name not in(  
SELECT Distinct S.name FROM Student S WHERE S.score <80  
)
```

方法 2:

```
SELECT S.name  
FROM Student S  
GROUP BY S.name  
Having MIN(S.score)>=80
```

七. 判断表的字段值是否为空

- 1、查询字段值为空的语法：where <字段名> **is null**
- 2、查询字段值不为空的语法：where <字段名> **is not null 或者 where NOT** (<字段名> IS NULL)

涉及索引的问题

一. 索引的优缺点。

16.2、索引优缺点

建立索引的优点

- 1.大大加快数据的检索速度;
- 2.创建唯一性索引, 保证数据库表中每一行数据的唯一性;
- 3.加速表与表之间的连接;
- 4.在使用分组和排序子句进行数据检索时, 可以显著减少查询中分组和排序的时间。

索引的缺点

- 1.索引需要占物理空间。
- 2.当对表中的数据进行增加、删除和修改的时候, 索引也要动态的维护, 降低了数据的维护速度。

性别字段为什么不适合加索引？从B+树原理解释。

尽量选择区分度高的字段作为索引,区分度的公式是 $\text{count}(\text{distinct col})/\text{count}(*)$, 表示字段不重复的比例, 比例越大我们扫描的记录数越少, 唯一键的区分度是 1, 而一些状态、性别字段可能在大数据面前区分度就是 0。在性别字段上增加索引, 并不能明显加快检索速度。

二 . 索引有哪些？

索引的作用是“排列好次序，使得查询时可以快速找到”。

唯一索引/非唯一索引、主键索引（主索引）、聚集索引/非聚集索引、组合索引。

1.唯一索引是在表上一个或者多个字段组合建立的索引，这个或者这些字段的值组合起来在表中不可以重复。

如下表中，为‘学号’建索引：

学号	姓名

01	张三
02	李四

2.非唯一索引是在表上一个或者多个字段组合建立的索引，这个或者这些字段的值组合起来**在表中可以重复，不要求唯一**。

如下表中，为 Score 建索引，可不唯一：

Score Name	

98	张三
98	李四
96	王五

3.主键索引（主索引）是唯一索引的特定类型。**表中创建主键时自动创建的索引**。
一个表只能建立一个主索引。

4.聚集索引（聚簇索引），表中记录的物理顺序与键值的索引顺序相同。一个表只能有一个聚集索引。

扩展：聚集索引和非聚集索引的区别？分别在什么情况下使用？

聚集索引和非聚集索引的**根本区别**是表中记录的物理顺序和索引的排列顺序是否一致。

聚集索引的表中记录的物理顺序与索引的**排列顺序一致**。

优点是**查询速度快**，因为一旦具有第一个索引值的记录被找到，具有连续索引值的记录也一定物理的紧跟其后。

缺点是对表进行修改速度较慢，这是为了保持表中的记录的物理顺序与索引的顺序一致，而把记录**插入到数据页的相应位置，必须在数据页中进行数据重**

排，降低了执行速度。在插入新记录时数据文件为了维持 B+Tree 的特性而频繁的分裂调整，十分低效。

建议使用聚集索引的场合为：

A.某列包含了小数目的不同值。

B.排序和范围查找。

非聚集索引的记录物理顺序和索引的顺序不一致。

其他方面的区别：

1.聚集索引和非聚集索引都采用了 B+树的结构，但非聚集索引的叶子层并不与实际的数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针的方式。聚集索引的叶节点就是数据节点，而非聚集索引的叶节点仍然是索引节点。

2.非聚集索引添加记录时，不会引起数据顺序的重组。

看上去聚簇索引的效率明显要低于非聚簇索引，因为每次使用辅助索引检索都要经过 两次 B+树查找，这不是多此一举吗？聚簇索引的优势在哪？

1.由于行数据和叶子节点存储在一起，这样主键和行数据是一起被载入内存的，找到叶子节点就可以立刻将行数据返回了，如果按照主键 Id 来组织数据，获得数据更快。

2.辅助索引使用主键作为"指针"，而不是使用地址值作为指针的好处是，减少了当出现行移动或者数据页分裂时,辅助索引的维护工作，InnoDB 在移动行时无须更新辅助索引中的这个"指针"。也就是说行的位置会随着数据库里数据的修改而发生变化，使用聚簇索引就可以保证不管这个主键 B+树的节点如何变化，辅助索引树都不受影响。

建议使用非聚集索引的场合为：

- a.此列包含了大数目的不同值；
- b.频繁更新的列

5.组合索引（联合索引）

基于多个字段而创建的索引就称为组合索引。

创建索引

```
create index idx1 on table1(col1,col2,col3)
```

查询

```
select * from table1 where col1= A and col2= B and col3 = C
```

组合索引查询的各种场景：

组合索引 Index (A, B, C)

下面条件可以用上该组合索引查询：

A>5
A=5 AND B>6
A=5 AND B=6 AND C=7
A=5 AND B=6 AND C IN (2, 3)

下面条件将不能用上组合索引查询：

B>5	——查询条件不包含组合索引首列字段
B=6 AND C=7	——理由同上

下面条件将能用上部分组合索引查询（重要！！！！）：

A>5 AND B=2	——当范围查询使用第一列，查询条件仅仅能使用 <u>第一列</u>
A=5 AND B>6 AND C=2	——范围查询使用第二列，查询条件仅仅能使用 <u>前列</u>
A=5 AND B IN (2, 3) AND C=2	——理由同上

组合索引排序的各种场景：

兹有组合索引 Index(A,B)。

下面条件可以用上组合索引排序：

ORDER BY A——首列排序

A=5 ORDER BY B——第一列过滤后第二列排序

ORDER BY A DESC, B DESC——注意，此时两列以相同顺序排序

A>5 ORDER BY A——数据检索和排序都在第一列

下面条件不能用上组合索引排序：

ORDER BY B ——排序在索引的第二列

A>5 ORDER BY B ——范围查询在第一列，排序在第二列

A IN(1,2) ORDER BY B ——理由同上

ORDER BY A ASC, B DESC ——注意，此时两列以不同顺序排序

```
alter table users add index lname_fname_age(lname,fname,age);
```

创建了 lname_fname_age 多列索引,相当于创建了(lname)单列索引, (lname,fname)联合索引以及(lname,fname,age)联合索引。

举例说明：上面给出一个多列索引(username,password,last_login)，当三列在 where 中出现的顺序如 (username,password,last_login)、(username,password)、(username) 才能用到索引，如下面几个顺序 (password,last_login)、(password)、(last_login)--- 这三者不从 username 开始，(username,last_login)---断层，少了 password，都无法利用到索引。因为 B+tree 多列索引保存的顺序是按照索引创建的顺序，检索索引时按照此顺序检索。

三 . 数据库索引的原理（实现）。

目前大部分数据库系统及文件系统都采用B-Tree(**B 树**)或其变种 B+Tree (**B+树**) 作为索引结构。 **B+Tree** 是数据库系统实现索引的首选数据结构。

在 MySQL 中，索引属于存储引擎级别的概念，**不同存储引擎对索引的实现方式是不同的**，本文主要讨论MyISAM 和InnoDB 两个存储引擎的**索引实现方式**。

MyISAM 索引实现

MyISAM 引擎使用B+Tree 作为索引结构，**叶节点的data 域存放的是数据记录的地址**。下图是MyISAM 索引的原理图：

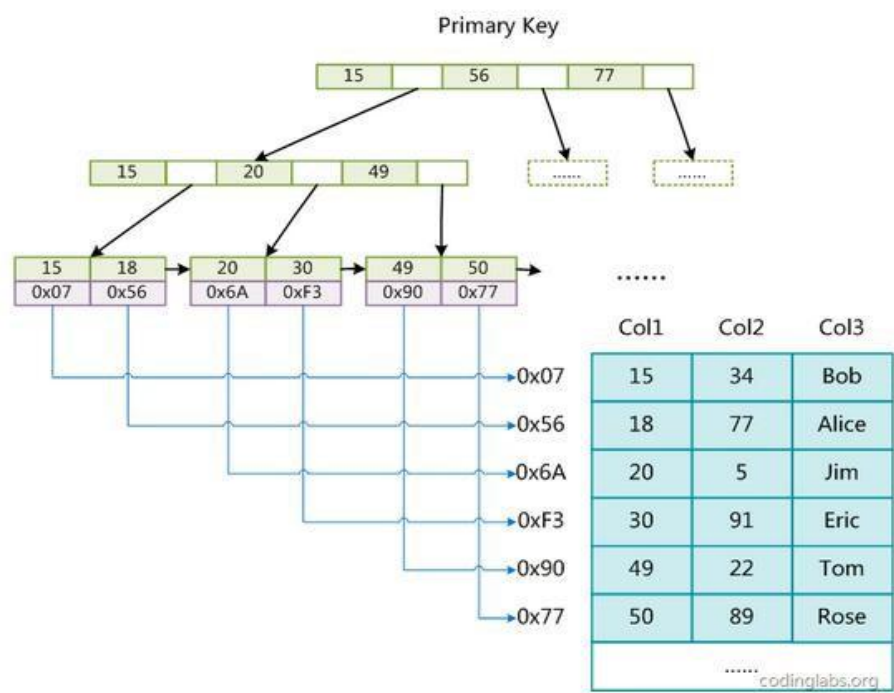


图8

主键索引

这里设表一共有三列，假设我们以 Col1 为主键，则图 8 是一个MyISAM 表的**主索引 (Primary key)** 示意。可以看出 MyISAM 的**索引文件**仅仅保存数据记录的地址。

在 MyISAM 中，主索引和辅助索引 (Secondary key) 在结构上没有任何区别，只是主索引要求key 是唯一的，而辅助索引的key 可以重复。如果我们在Col2 上建立一个辅助索引，则此索引的结构如下图所示：

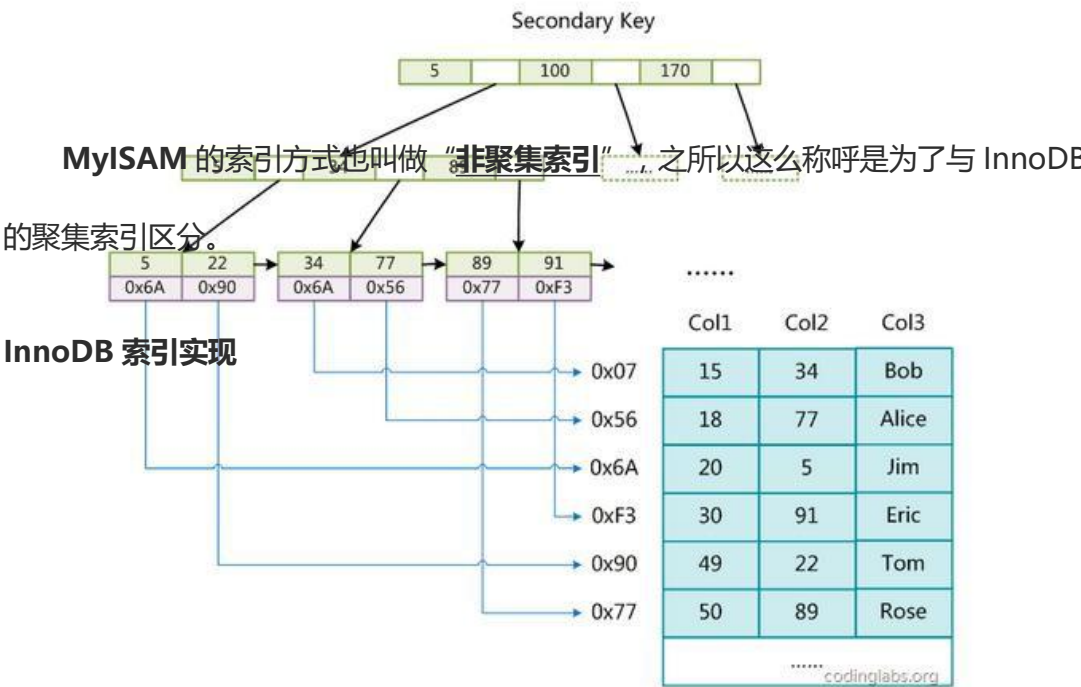


图9

辅助索引

同样也是一颗B+Tree，data 域保存数据记录的地址。因此，MyISAM 中索引检索的算法为首先按照 B+Tree 搜索算法搜索索引，如果指定的Key 存在，则取出其data 域的值，然后以 data 域的值为地址，读取相应数据记录。

虽然InnoDB 也使用B+Tree 作为索引结构，但具体实现方式却与MyISAM 截然不同。

1. 第一个重大区别是InnoDB 的数据文件本身就是索引文件。从上文知道，MyISAM 索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB 中，表数据文件本身就是按 B+Tree 组织的一个索引结构，这棵树的叶节点 data 域保存了完整的数据记录。这个索引的key 是数据表的主键，因此InnoDB 表数据文件本身就是主索引。

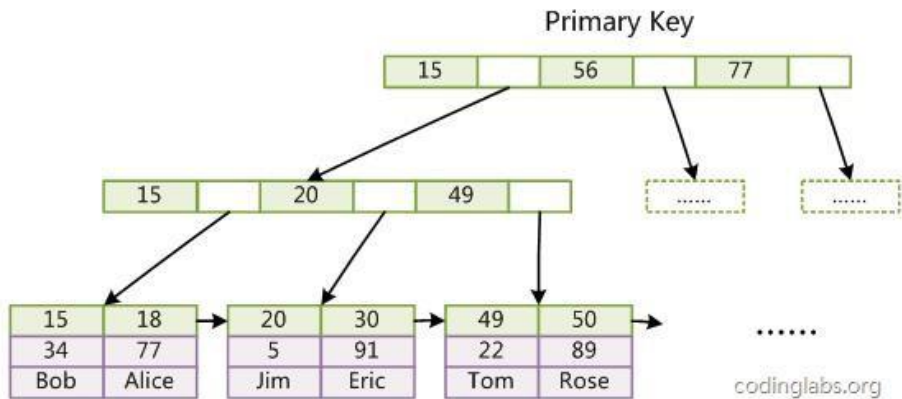


图10

主索引

上图是InnoDB 主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做**聚集索引**。因为InnoDB 的数据文件本身要按主键聚集，

1 .InnoDB 要求表必须有主键（ MyISAM 可以没有 ），如果没有显式指定，则 MySQL 系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则 MySQL 自动为InnoDB 表生成一个隐含字段作为主键，类型为长整形。同时，请尽

量 2.在InnoDB 上采用自增字段做表的主键。因为InnoDB 数据文件本身是一棵 B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持 B+Tree 的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页。如下图所示：

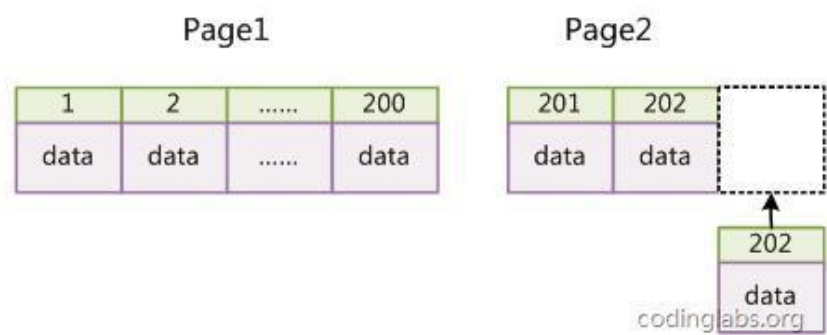


图13

这样就会形成一个紧凑的索引结构，近似顺序填满。由于每次插入时也不需要移动已有数据，因此效率很高，也不会增加很多开销在维护索引上。

2.第二个与MyISAM 索引的不同是InnoDB 的辅助索引data 域存储相应记录主键的值而不是地址。换句话说，InnoDB 的所有辅助索引都引用主键作为 data 域。

例如，图 11 为定义在Col3 上的一个辅助索引：

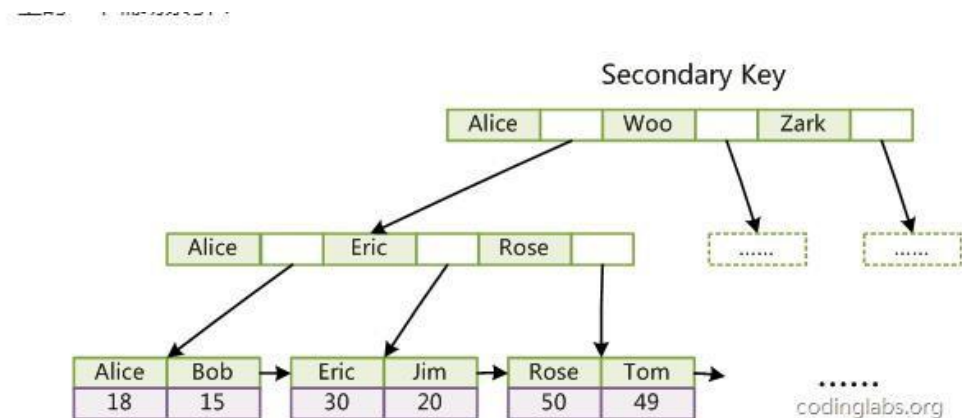


图11

辅助索引

聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：

首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录。

引申：为什么不建议使用过长的字段为主键？

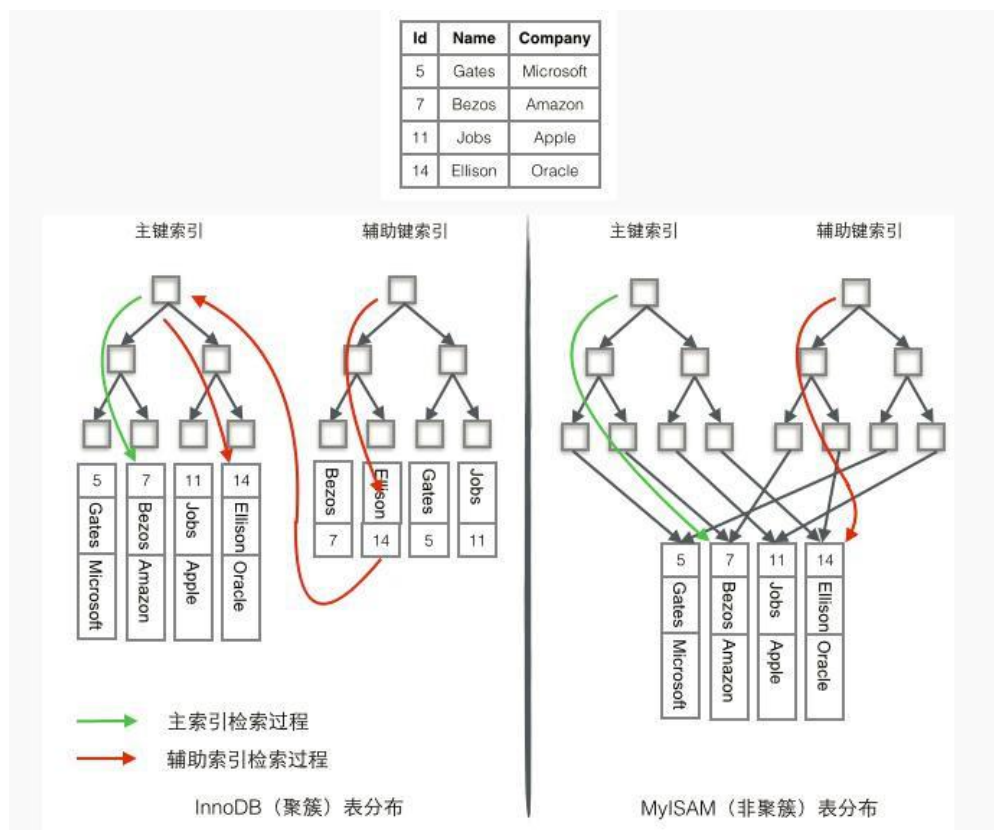
因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。

扩展阅读 1：

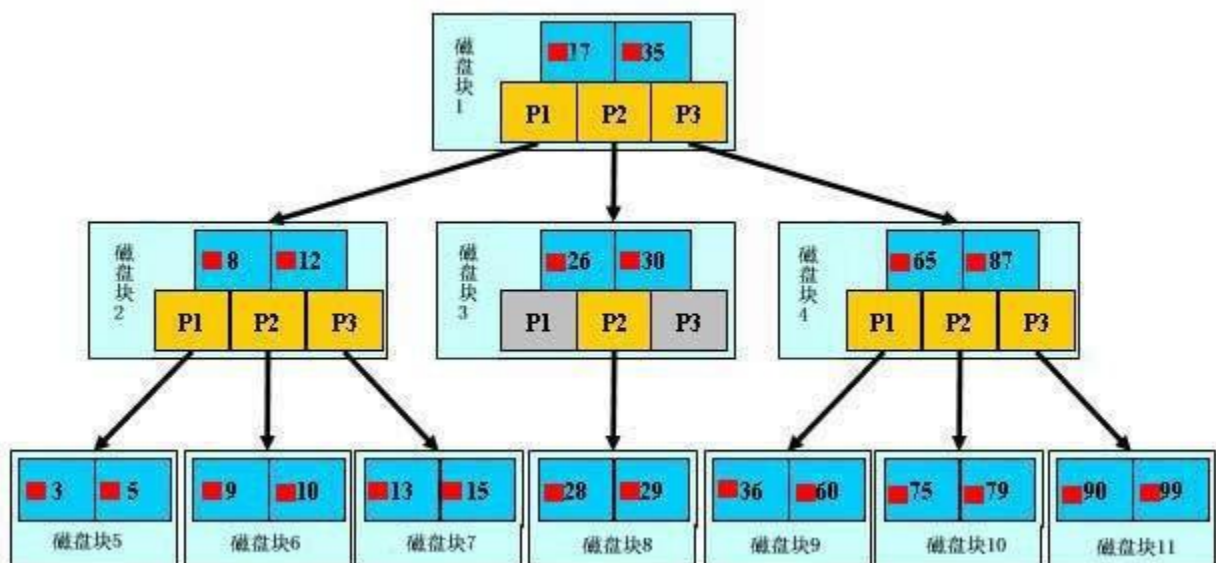
InnoDB 使用的是聚簇索引，将主键组织到一棵 B+ 树中，而行数据就储存在叶子节点上，若使用 "where id = 14" 这样的条件查找主键，则按照 B+ 树的检索算法即可查找到对应的叶节点，之后获得行数据。若对 Name 列进行条件搜索，则需要两个步骤：第一步在辅助索引 B+ 树中检索 Name，到达其叶子节点获取对应的主键。第二步使用主键在主索引 B+ 树中再执行一次 B+ 树检索操作，最终到达叶子节点即可获取整行数据。

MyISM 使用的是非聚簇索引，非聚簇索引的两棵 B+ 树看上去没什么不同，节点的结构完全一致只是存储的内容不同而已，主键索引 B+ 树的节点存储了主键，辅助键索引 B+ 树存储了辅助键。表数据存储在独立的地方，这两颗 B+ 树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

为了更形象说明这两种索引的区别，我们假想一个表如下图存储了 4 行数据。其中 Id 作为主索引，Name 作为辅助索引。图示清晰的显示了聚簇索引和非聚簇索引的差异。



扩展阅读 2：



b 树的查找过程：

如图所示，如果要查找数据项 29，那么首先会把磁盘块 1 由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定 29 在 17 和 35 之间，锁定磁盘块 1 的 P2 指针，内存时间因为非常短（相比磁盘的 IO）可以忽略不计，通过磁盘块 1 的 P2 指针的磁盘地址把磁盘块 3 由磁盘加载到内存，发生第二次IO，29 在 26 和 30 之间，锁定磁盘块 3 的 P2 指针，通过指针加载磁盘块 8 到内存，发生第三次IO，同时内存中做二分查找找到 29，结束查询，总计三次IO。真实的情况是，3 层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

注意：当b+树的数据项是复合的数据结构（建立的是复合索引），比如 (name,age,sex)的时候，b+树是按照从左到右的顺序来建立搜索树的，比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较 name 来确定下一步的搜索方向，如果name 相同再依次比较 age 和sex，最后得到检索的数据；但当(20,F)这样的没有 name 的数据来的时候，b+树就不知道下一步该查哪个节点，因为建立搜索树的时候 name 就是第一个比较因子，必须要先根据 name 来搜索才能知道下一步去哪里查询。比如当(张三,F)这样的数据来检索时，b+树可以用name 来指定搜索方向，但下一个字段age 的缺失，所以只能把名字等于张三的数据都找到，然后再匹配性别是 F 的数据了，这个是非常重要的性质，即索引的最左匹配特性**。注意：**B+tree 多列索引保存的顺序是按照索引创建的顺序，检索索引时按照此顺序检索。****

最左前缀原则中 where 字句有 or 出现还是会遍历全表。

四 Mysql 的 B+树索引的优点？为什么不用二叉树？B-树和 B+树为什么比红黑树更合适？

数据库文件很大，需要存储到磁盘上，索引的结构组织要尽量减少查找过程中磁盘 I/O 的存取次数。

1. 高度原因

B+ 树中的每个结点可以包含大量的关键字，这样树的深度降低了，所以任何关键字的查找必须走一条从根结点到叶子结点的路，所有关键字查询的路径长度相同，导致每一个数据的查询效率相当，这就意味着查找一个元素只要很少结点从外存磁盘中读入内存，很快访问到要查找的数据，减少了磁盘 I/O 的存取次数。

2. 磁盘预读原理和局部性原理

将一个节点的大小设为等于一个页，这样每个节点只需要一次 I/O 就可以完全载入。

五 . 建索引的几大原则。

1 最左前缀匹配原则，mysql 会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，范围查询会导致组合索引半生效。

比如 a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，c 可以用到索引，d 是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d 的顺

序可以任意调整。where 范围查询要放在最后（这不绝对，但可以利用一部分索引）。

特别注意：and 之间的部分可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)

索引可以任意顺序，mysql 的查询优化器会帮你优化成索引可以识别的形式。

where 字句有 or 出现还是会遍历全表。

2 尽量选择**区分度**高的字段作为索引,某字段的区分度的公式是 $\text{count}(\text{distinct}$

$\text{col})/\text{count}(*)$ ，表示字段不重复的比例，**比例越大，我们扫描的记录数越少**，查找匹配的时候可以过滤更多的行，唯一索引的区分度是 1，而一些状态、**性别字段**可能在大数据面前区分度就是 0。

4.不在索引列做运算或者使用函数。

5.尽量**扩展索引，不要新建索引**。比如表中已经有 a 的索引，现在要加(a,b)的索引，那么只需要**修改原来的索引**即可。

6. **Where 子句**中经常使用的字段应该创建索引，

分组字段或者**排序字段**应该创建索引，

两个表的**连接字段**应该创建索引。

7.like 模糊查询中，右模糊查询（321%）会使用索引，而%321 和%321%会放弃索引而使用全局扫描。

涉及存储引擎的问题

一. mysql 中 MyIsam 与 InnoDB 的区别，至少 5 点。

Mysql 数据库中，最常用的两种引擎是 innodb 和 myisam。InnoDB 是 Mysql 的默认存储引擎。

[1.事务处理上方面](#)

MyISAM 强调的是**性能**，查询的**速度**比 InnoDB 类型更快，但是**不提供事务支持**。

InnoDB **提供事务支持事务**。

[2.外键](#)

MyISAM **不支持外键**，InnoDB **支持外键**。

[3.锁](#)

MyISAM **只支持表级锁**，InnoDB **支持行级锁和表级锁**，默认是行级锁，行锁大幅度提高了多用户并发操作的性能。innodb 比较适合于**插入和更新操作**比较多的情况，而 myisam 则适合用于**频繁查询**的情况。另外，InnoDB 表的行锁也不是绝对的，如果在执行一个 SQL 语句时，MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表，例如 update table set num=1 where name like "%aaa%" 。

[4.全文索引](#)

MyISAM **支持全文索引**，InnoDB **不支持全文索引**。innodb 从 mysql5.6 版本开始提供对全文索引的支持。

[5.表主键](#)

MyISAM：允许没有主键的表存在。

InnoDB：如果没有设定主键，就会自动生成一个 6 字节的主键(用户不可见)。

6. 表的具体行数

MyISAM：select count(*) from table,MyISAM 只要简单的读出保存好的行数。因为 MyISAM 内置了一个计数器，count(*)时它直接从计数器中读。

InnoDB 不保存表的具体行数，也就是说，执行select count(*) from table 时，InnoDB 要扫描一遍整个表来计算有多少行。

二. 一张表,里面有 ID 自增主键,当 insert 了 17 条记录之后,删除了第 15,16,17 条记录,再把 Mysql 重启,再 insert 一条记录,这条记录的ID 是 18 还是 15 ?

如果表的类型是MyISAM，那么是 18。

因为 MyISAM 表会把自增主键的最大 ID 记录到数据文件里，重启 MySQL 自增主键的最大ID 也不会丢失。

如果表的类型是InnoDB，那么是 15。

InnoDB 表只是把自增主键的最大 ID 记录到内存中，所以重启数据库会导致最大ID 丢失。

其他问题

一. 如果在数据库上进行了误操作该怎么处理。

1.利用数据备份。

2. 利用 binlog 文件。

二. 数据库在进行水平分表之后, sql 分页查询该怎么进行? 分表之后想让一个 id 多个表是自增的, 效率实现。数据库中的分页查询语句怎么写?

1. 用 union all 合并几个分表的结果集, 之后进行分页查询。

2. 如假定共 3 个分表, 记录数分别为 90, 120, 80, 总记录数为 290

设分页是每页显示 40 条, 则

第 1 页 表一的 1 到 40

第 2 页 表一的 41 到 80

第 3 页 表一的 81 到 90 + 表二的 1 到 30

第 4 页 表二的 31 到 70

第 5 页 表二的 71 到 110

第 6 页 表二的 111 到 120 + 表三的 1 到 30

.....

3. 用 sphinx (斯芬克司) 先建索引, 然后分页查询。

当我们对 MySQL 进行分表操作后, 将不能依赖 MySQL 的自动增量来产生唯一 ID 了, 因为数据已经分散到多个表中。

使用队列服务, 如 redis、memcacheq 等, 将一定量的 ID 预分配在一个队列里, 每次插入操作, 先从队列中获取一个 ID, 若插入失败的话, 将该 ID 再次添加到队列中, 同时监控队列数量, 当小于阈值时, 自动向队列中添加元素。

三. 关系型数据库和非关系型数据库的区别。

非关系型数据库的优势:

1. 性能

NOSQL 是基于键值对的, 可以想象成表中的主键和值的对应关系, 而且不需要经过 SQL 层的解析, 所以性能非常高。

2. 可扩展性

同样也是因为基于键值对, 数据之间没有耦合性, 所以非常容易水平扩展。

关系型数据库的优势:

1. 复杂查询

可以用 SQL 语句方便的在一个表以及多个表之间做非常复杂的数据查询。

2. 事务支持

使得对于安全性能很高的数据访问要求得以实现。

四. 数据库连接池的原理？连接池使用什么数据结构实现？实现连接池？

一.早期我们怎么进行数据库操作

1.原理：一般来说，java 应用程序访问数据库的过程是：

- ①加载数据库驱动程序；
- ②通过jdbc 建立数据库连接；
- ③访问数据库，执行sql 语句；
- ④断开数据库连接。

2.代码

// 查询所有用户

```
1.  Public void FindAllUsers(){
2.      //1、装载 sqlserver 驱动对象
3.      DriverManager.registerDriver(new SQLServerDriver());
4.      //2、通过 JDBC 建立数据库连接
5.      Connection con =DriverManager.getConnection("jdbc:sqlserver://192.168.
        2.6:1433;DatabaseName=customer", "sa", "123");
6.      //3、创建状态
7.      Statement state =con.createStatement();
8.      //4、查询数据库并返回结果
9.      ResultSet result =state.executeQuery("select * from users");

10.     //5、输出查询结果
11.     while(result.next()){
12.         System.out.println(result.getString("email"));
13.     }
14.     //6、断开数据库连接
15.     result.close();
16.     state.close();
17.     con.close();
18. }
```


3. 分析

程序开发过程中，存在很多问题：

首先，每一次 web 请求都要建立一次数据库连接。**建立连接是一个费时的活动**，每次都得花费 0.05s ~ 1s 的时间，而且**系统还要分配内存资源**。这个时间对于一次或几次数据库操作，或许感觉不出系统有多大的开销。可是对于现在的 web 应用，尤其是大型电子商务网站，同时有几百人甚至几千人在线是很正常的事。在这种情况下，**频繁的进行数据库连接操作势必占用很多的系统资源，网站的响应速度必定下降，严重的甚至会造成服务器的崩溃**。不是危言耸听，这就是制约某些电子商务网站发展的技术瓶颈问题。

其次，**对于每一次数据库连接，使用完后都得断开**。否则，如果程序出现异常而未能关闭，**将会导致数据库系统中的内存泄漏**，最终将不得不重启数据库。还有，这种开发不能控制被创建的连接对象数，系统资源会被毫无顾及的分配出去，如连接过多，也可能导致内存泄漏，服务器崩溃。

通过上面的分析，我们可以看出来，“**数据库连接**”**是一种稀缺的资源**，为了保障网站的正常使用，应该对其进行妥善管理。其实我们查询完数据库后，如果不关闭连接，而是暂时存放起来，当别人使用时，把这个连接给他们使用。就避免了一次建立数据库连接和断开的操作时间消耗。原理如下：

二. 技术演进出来的数据库连接池

由上面的分析可以看出，问题的根源就在于对数据库连接资源的低效管理。我们知道，**对于共享资源**，有一个著名的设计模式：**资源池设计模式**。该模式正是为了解决

资源的频繁分配、释放所造成的问题。为解决上述问题，可以采用数据库连接池技术。

数据库连接池的基本思想就是为数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。我们可以通过设定连接池最大连接数来防止系统无尽地与数据库连接。

更为重要的是我们可以通过连接池的管理机制监视数据库的连接的数量、使用情况，为系统开发、测试及性能调整提供依据。

我们自己尝试开发一个连接池，来为上面的查询业务提供数据库连接服务：

- ① 编写class 实现DataSource 接口
- ② 在class 的构造器一次性创建 10 个连接，将连接保存LinkedList 中
- ③ 实现getConnection，从 LinkedList 中返回一个连接
- ④ 提供将连接放回连接池中的方法

1、连接池代码

```
1. public class MyDataSource implements DataSource {
2.     //因为 LinkedList 是用链表实现的,对于增删实现起来比较容易
3.     LinkedList<Connection> dataSources = new LinkedList<Connection>();
4.
5.     //初始化连接数量
6.     public MyDataSource() {
7.         //问题：每次 new MyDataSource 都会建立 10 个链接，可使用单例设计模式解决此类问题
8.
9.         for(int i = 0; i < 10; i++) {
10.            try {
11.                //1、装载sqlserver 驱动对象
12.                DriverManager.registerDriver(new SQLServerDriver());
13.            };
```

```

14.         "jdbc:sqlserver://192.168.2.6:1433;DatabaseNam
    e=customer", "sa", "123");
15.         //3、将连接加入连接池中
16.         dataSources.add(con);
17.     } catch (Exception e) {
18.         e.printStackTrace();
19.     }
20.     }
21. }
22.
23. @Override
24. public Connection getConnection() throws SQLException {
25.     //取出连接池中一个连接
26.     final Connection conn = dataSources.removeFirst(); // 删除第
    一个连接返回
27.     return conn;
28. }
29.
30. //将连接放回连接池
31. public void releaseConnection(Connection conn) {
32.     dataSources.add(conn);
33. }
34. }

```

2、使用连接池重构我们的用户查询函数

```

1. //查询所有用户
2. Public void FindAllUsers(){
3.     //1、使用连接池建立数据库连接
4.     MyDataSource dataSource = new MyDataSource();
5.     Connection conn =dataSource.getConnection();
6.     //2、创建状态
7.     Statement state =con.createStatement();
8.     //3、查询数据库并返回结果
9.     ResultSet result =state.executeQuery("select * from users");

10.    //4、输出查询结果
11.    while(result.next()){
12.        System.out.println(result.getString("email"));
13.    }
14.    //5、断开数据库连接

```

```
15.         result.close();
16.         state.close();
17.         //6、归还数据库连接给连接池
18.         dataSource.releaseConnection(conn);
19. }
```

连接池的工作原理：

连接池的核心思想是连接的复用，通过建立一个数据库连接池以及一套连接使用、分配和管理策略，使得该连接池中的连接可以得到高效，安全的复用，避免了数据库连接频繁建立和关闭的开销。

连接池的**工作原理**主要由三部分组成，分别为连接池的**建立**，连接池中连接的**使用管理**，连接池的**关闭**。

第一、连接池的建立。一般在系统初始化时，连接池会根据系统配置建立，并在池中建立几个连接对象，以便使用时能从连接池中获取。**java** 中提供了很多容器类，可以方便的构建连接池，例如 **Vector**（线程安全类），**linkedlist** 等。

第二、连接池的管理。连接池管理策略是连接池机制的核心，连接池内连接的分配和释放对系统的性能有很大的影响。其策略是：

当客户请求数据库连接时，首先查看连接池中是否有空闲连接，如果存在空闲连接，则将连接分配给客户使用并作相应处理（即标记该连接为正在使用，引用计数加 1）；如果没有空闲连接，则查看当前所开的连接数是否已经达到最大连接数，如果没有达到**最大连接数**，如果没达到就重新创建一个连接给请求的客户；如果达到，就按设定的**最大等待时间**进行等待，如果超出最大等待时间，则抛出异常给客户。

当客户释放数据库连接时，先判断该连接的引用次数是否超过了规定值，如果超过了就从连接池中删除该连接，并判断当前连接池内总的连接数是否小于**最小连接数**，若小于就将连接池充满；如果没超过就将该连接标记为开放状态，可供再次复用。

第三、连接池的关闭。当应用程序退出时，关闭连接池中所有的链接，释放连接池相关资源，该过程正好与创建相反。

连接池的主要优点：

- 1) 减少连接的创建时间。连接池中的连接是已准备好的，可以重复使用的，获取后可以直接访问数据库，因此减少了连接创建的次数和时间。
- 2) 更快的系统响应速度。数据库连接池在初始化过程中，往往已经创建了若干数据库连接置于池中备用。此时连接的初始化工作均已完成。对于业务请求处理而言，直接利用现有可用连接，避免了数据库连接初始化和释放过程的时间开销，从而缩减了系统整体响应时间。
- 3) 统一的连接管理。如果不使用连接池，每次访问数据库都需要创建一个连接，这样系统的稳定性受系统的连接需求影响很大，很容易产生资源浪费和高负载异常。连接池能够使性能最大化，将资源利用控制在一定的水平之下。连接池能控制池中的链接数量，增强了系统在大量用户应用时的稳定性。

c3p0 数据库连接池。

五. 列级约束与表级约束的区别？

(1) **列级约束**：只能应用于一列上。

表级约束：可以应用于一列上，也可以应用在一个表中的多个列上。

(**即**：如果你创建的约束涉及到该表的多个属性列，则必须创建的是表级约束（必须定义在表级上）；否则既可以定义在列级上也可以定义在表级上，此时只是 SQL 语句格式不同而已）

(2) **列级约束**：包含在列定义中，直接跟在该列的其它定义之后，用空格分隔；不必指定列名。

表级约束：与列定义相互独立，不包含在列定义中；与定义用‘,’分隔；必须指出要约束的列的名称。

(**注**：因为在创建列级约束时，只需将创建列约束的语句添加到该字段（列）的定义子句后面；而在创建表级约束时，需要将创建表级约束的语句添加到各个字段（列）定义语句的后面，因为并不是每个定义的字段都要创建约束，所以必须指明需要创建的约束的列名。) **举例**

说明：Create Table project

```
(  
    项目编号 int  
    Constraint pk_pno primary key,  
    项目名称 char(20),
```

项目负责人 char(20),
Constraint un_pname_pm unique (项目名称, 项目负责人)
)

分析：“项目编号”字段设置为主键，主键约束名为 pk_pno，此主键约束为列主键约束“项目名称”和“项目负责人”的组合字段设置唯一性约束，此约束为表级约束）

六. 关系五种基本运算

① 并：

R,S 具有相同的模式（元素相同，结构相同），记为 $R \cup S$, 返回由 R 或者 S 元组构成的集合组成

② 差：

R,S 具有相同的模式（元素相同，结构相同），记为 $R - S$ ，由属于 R 但不属于 S 的元组组成

③ 广义笛卡尔积：

$R \times S$ 由 n 目和 m 目的关系 R,S 组成一个 (n+m) 列的元组集合，若 R 有 K_1 个元组，S 有 K_2 个元组，则 $R \times S$ 有 $K_1 \times K_2$ 个元组。

④ 投影 (π)：

从关系的垂直方向开始运算，选择关系中的若干列组成新的列。

⑤ 选择 (σ)：

从关系的水平方向进行运算，选择满足给定条件的元组组成新的关系。

Part-----数据库专题（补 1）

1.数据库索引失效的几种情况。

- 1.对于组合索引，不是使用的第一部分，则不会使用索引。
- 2.or 语句前后没有同时使用索引。要想使用 or，又想让索引生效，只能将 or 条件中的每个列都加上索引。
- 3.如果列类型是字符串，那一定要在条件中使用引号引起来，否则不会使用索引。
- 4.如果 mysql 估计使用全表描述比使用索引快，则不使用索引。
- 5.在索引列上做运算或者使用函数。
- 6.以“%”开头的 LIKE 查询，模糊匹配。

2.一个表只有一列 name，有重复的 name，求出前十个 name 数最大的 name。

```
select distinct name,count(name) a
from user
group by name
order by a desc
limit 10
```

扩展：limit 的用法。

一.如果给定两个参数，第一个参数指定第一个返回记录行的偏移量，第二个参数指定返回记录行的最大数目。初始记录行的偏移量是 0(而不是 1)。

```
SELECT * FROM table LIMIT 5,10; //检索记录行 6-15
```

//为了检索从某一个偏移量到记录集的结束所有的记录行，可以指定第二个参数为 -1:

```
SELECT * FROM table LIMIT 95,-1; // 检索记录行 96-last.
```

二.如果只给定一个参数，它表示返回最大的记录行数目。

```
SELECT * FROM table LIMIT 5; //检索前 5 个记录行；LIMIT n 等价于 LIMIT 0,n。
```

3.服务器与服务器之间传输文件夹下的文件，一个文件夹下有 10 个文件，另一个文件夹下有 100 个文件，两个文件夹大小相等，问，哪个传输更快？

10 个文件更快。

1) 建立连接数更少，建立连接的开销比传输文件的开销大。

2) 文件写入磁盘，要计算文件的起始位置，文件数目少的话，这个开销就小了。

4. 数据库表里有 100 万条数据，想要删除 80 万条数据，但是因为锁的原因，删除很慢，现在想要快速删除怎么办？

如果需要保留的数据不多，需要删除的数据很多，那么可以考虑把需要保留的数据复制到临时表，然后删除所有数据，最后复制回去。

具体做法是：先把要保留的数据用 `insert into ... select * from ... where ...` 移到另外的表中，**truncate table** 旧表，然后再 `insert into ... select * from ...`，这个不存在锁，比 `delete` 效率高。