数据库事务隔离级别：

脏读：事务A读取了事务B还未提交的数据

不可重复读：事务A读取一条记录后，事务B修改当前记录，事务A再去读，两次读取的记录结果就不一样(一个事务范围内多次查询却返回了不同的数据值，这是由于在查询间隔，被另一个事务修改并提交了)

幻读：幻读和不可重复读的意思差不到，但是还是有本质的区别

不可重复读和幻读的区别：两者在意思都差不多，但是有本质区别的

避免不可重复读锁行就可以了，避免幻读所表就可以

不可重复读重点在于select，重点在一行记录

幻读重点在insert/update/delete，重点在整个表

不可重复读是事务A是select一行数据后，事务B对该行修改了，事务A再select(当前事务内)，发现两次不一样(当行)

幻读是事务A在读取整张表或表的一段连续部分时(between and)，事务B向表中做了修改(事务A的范围)，导致事务A两次读取结果不同(多行)

数据库的隔离级别：

读未提交(Read uncommitted)：最低级别，任何事情都无法保证

读已提交(Read committed)：可避免脏读

可重复读(Repeatable read)：可避免脏读，不可重复读

串性化(Serializable )：可避免脏读，不可重复读，幻读，内存消耗大，很少用(锁表)

Oracle，SQLServer默认事务隔离级别是RC

MySQL的默认事务隔离级别是RR，但是MySQL采用InnOB引擎，可以避免幻读(MVCC)

事务的四个特性：

A：原子性(atomicity)，事务要么全部成功，要么全部失败

C：一致性(consistency)，张三转账给李四100元，张三账户减100，那么李四就账户就必须要增100元，不存在张三减100，李四任然不变的中间状态(就是区别正确性)

I：隔离性(islation)，多个事务同时执行时互不影响，相互是隔离的

D：持久化(durability)，事务一旦提交(commit)，就不能在恢复了(rollBack)

数据库碎片：

当删除数据库中的某条或多条记录后，记录所处的存储空间任然存在(只是没有数据)，这段空间就叫数据库碎片

Mysql可以使用optimize table tableName命令对表进行优化，清理碎片，在运行命令时，会锁表

MySQL引擎：MyISAM/ InnoDB常用

MEMORY： 通过在内存中创建临时表来存储数据，每个表对应一个磁盘文件，该文件只存储表的结构，其数据存储在内存中，所以该引擎有极高的插入，更新和查询效率，通过Hash索引，由于数据在内存中所以很容易丢失，所以该引擎生命周期很短

MyISAM：在读取数据方面速度快，且不占用大量内存和存储资源，但不支持事务，也不支持行级锁和外键，当insert时，会锁全表，效率低，但是此所以存储了表的行数，所以select count(\*)的时候不用扫描全表，如果读操作(select)大于写操作且不需要事务时，MySAM是很好的选择(小型应用可以)，B+树索引

InnoDB：该引擎为MySQL提供了事务支持(默认RR)，系统修复和多版本并发控制(MVCC)的行级锁，支持自增长列，支持行级锁和外键约束(MySQL默认引擎)，MySQL在运行时InnoDB会在内存中建立缓冲池，用于缓存数据和索引，该引擎没有保存表的行数，所以select count(\*) form table时需要扫描全表，该引擎具有行锁，所以在高并发下，使用InnoDB的效率较高，B+树索引

数据库常见索引：B+Tree/Hash

数据库的索引：

在执行select \* from person where name=”Jack”的时候，会去扫描整个表，找到所有满足的记录，全表扫描降低了性能，所以需要索引。索引是一种数据结构，select \* from person where id=”120”，在根据id查找的时候，我们就可以为id创建索引，如id是由小到大排序，在查找id的时候就可以根据二分法查找，减少了查询时间，提高了效率，而在创建表的时候需要声明主键，其实就是在创建索引(主键：确保记录唯一/索引)

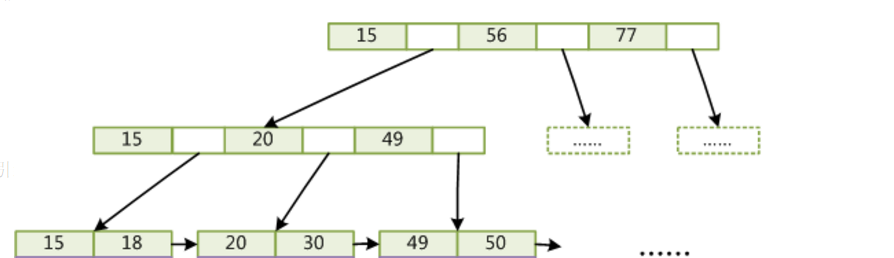
原则上可以为表的每一列都创建索引，但是索引的创建和维护都需要大量的时间，和数据量成正比，并且所以还需要占用物理空间，除了数据空间外，每个索引还要占用物理空间，虽然索引可以提高查询(select)的效率，但是在insert/update/delete时，不仅要更新数据表，还有更新索引，所以索引不能创建过多

数据库中常见的索引方式是B+Tree和Hash索引

Hash索引采用的是hash算法，排序在hash数组上，所以hash所以可以一次定位效率高，但是hash索引比较的是hash值，所以hash索引只适用于指定查询，不能用于一定范围内查询

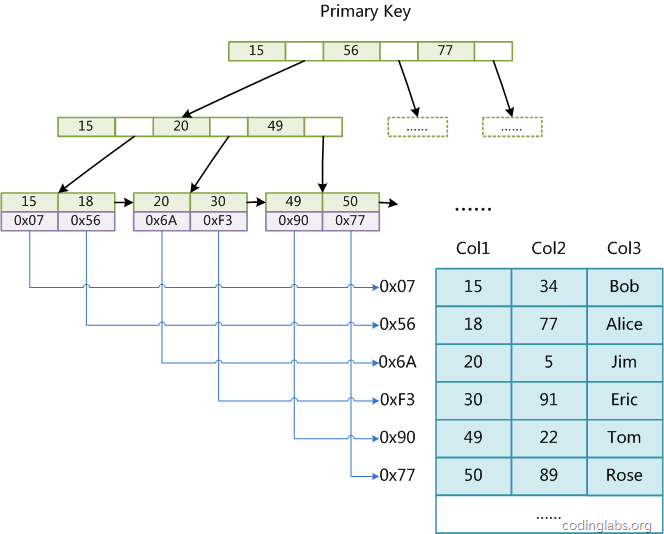
B++Tree的实现：<https://www.cnblogs.com/wade-luffy/p/6292784.html> ，B+Tree是基于平衡二叉树实现的，但是如果采用平衡二叉树作为索引，当数据很大时，输的深度会很大，每次IO请求会增多，所以B+Tree在每个节点增加了分支(多路搜索树)，并且B+Tree的每个节点的大小是**页**的倍数，页是计算机存储器的逻辑地址，每次IO请求时读取的都是页(4K)的倍数

B+Tree中，所有的叶子节点才是数据域，而非根节点只是起查询判断的作用，而且叶子节点都是双向链表



MyISAM和InnoDB都采用的是B+Tree索引，但是两者存在差距的

MyISAM的B+Tree的叶子节点的数据域中保存的是索引和记录对应的地址，在查询的时候，先根据索引查到记录的地址，再通过地址找到数据，这种方式也叫非聚集索引



MyISAM的辅助索引和非聚集索引类似，保存的都是地址

InnoDB的B+Tree的叶子节点的数据域保存的是索引和数据，找到索引后就可以直接取出数据，这也叫聚集索引，因为InnoDB的数据文件本身按主键聚集，所以InnoDB的表必须有主键(因为辅助索引要通过主键查询，所以聚集索引只能存在一个)



InnoDB的辅助索引中的data域保存的是主键，先在辅助索引中找到对应的主键，然后再到聚集索引中查询，因为辅助索引也会包含主键列，所以在定义主键的时候要合适，不要太大

MyISAM的索引文件和数据文件是分离的，索引文件保存的是地址，而InnoDB数据文件本身就是一个索引结构，数据保存在叶子节点的data域，InnoDB保存的记录时连续的，而MyISAM保存的是无序的(地址)，一般情况下，建议采用自增的主键作为InnoDB的聚集索引，自增已经排好序，所以速度快

**InnoDB索引**和**MyISAM索引**的区别：

一是主索引的区别，InnoDB的数据文件本身就是索引文件。而MyISAM的索引和数据是分开的。

二是辅助索引的区别：InnoDB的辅助索引data域存储相应记录主键的值而不是地址。而MyISAM的辅助索引和主索引没有多大区别。

唯一索引(UNIQUE)/主键索引(PRIMARY)/聚集索引/普通索引(INDEX)

主键索引：ALTER TABLE `表名` ADD PRIMARY KEY ( `列名称` )

唯一索引：ALTER TABLE `表名` ADD UNIQUE ( `列名称`)

普通索引：ALTER TABLE `表名` ADD INDEX index\_name ( `列名称` )

唯一索引就是确保改列不会出现同一属性值

主键索引确保主键(一列或多列)所在的列不会出现相同的属性值

聚集索引/非聚集索引：聚集索引的访问速度更快，不必查询两次

索引不仅占用存储空间，而且插入和修改数据时花费的时间较多

删除记录会导致索引的数据结构不完整，所以一般都是假删除(删除后索引为空)

主键自动创建唯一索引

进行增删改表的表或字段不要创建索引

索引列最好是NOT NULL，如果为空，会对索引造成影响

MyISAM/ InnoDB的区别：事务，锁，聚集索引

InnoDB支持事务，行级锁(mvcc)，MyISAM不支持，表锁，即MyISAM同一个表上的读锁和写锁是互斥的(写优先)

InnoDB是聚集索引，查询速度快，MyISAM是非聚集索引

MyISAM适用于insert，update操作，InnoDB适用于select操作

乐观锁/悲观锁：

乐观锁是指当前用户在对记录进行操作的时候，其他用户不会操作，各个操作能在不加锁的基础上进行，乐观的认为自己在修改数据的时候别人不会修改，乐观锁的实现：为表添加一个version列(版本)，在更新数据时，先select version form table where id =#{id}，读取到version，然后进行更新操作，update table set \*\*\*,version=version+1 where id=#{id} and version=#{version}，如果在select和update之间无其他用户更新此表，那个操作将成功，否则更新失败，但是，MySQL的默认是RR隔离级别，是不会存在乐观锁的，一个事务内可重复读，乐观锁通常用于读操作，当资源争用不要严重时(多个用户不会同时访问同一个资源，线程对资源的占用时间短，当A竞争资源时，发现资源以被占用，所以挂起，但是B很快就是否资源了，释放资源后，A还再等时间片的，所以增加了A的时间，采用乐观锁，循环执行，直到成功，可以减少时间)，采用乐观锁(读操作适用乐观锁)

悲观锁是指用户之间在同一段时间会对同一数据进行操作，彼此影响，所以在操作的时候加上独占锁，即select xxx for update

数据库隔离级别是在锁的基础上实现的

表锁/行锁：

表锁：开销小，加锁快，不会出现死锁，粒度大，发生锁冲突的可能性大，并发低

行锁：开销大，加锁慢，会出现死锁，粒度小，发生锁冲突的可能性小，并发高

MyISAM只支持表锁，读操作(select)时，会阻塞其他用户的写请求，但不会阻塞读请求，写操作时，会阻塞其他用户的所有操作

MyISAM在执行select语句时，会自动在涉及的表上加读锁，在update，delete，insert时加写锁，也可以手动添加锁lock table tableName1 read(添加写锁)，但是手动添加锁后，在释放锁之前(unlock tables)只能操作tableName1，不能操作其他表

InnoDB实现了一下两者类型的行锁：<https://www.cnblogs.com/boblogsbo/p/5602122.html>

共享锁(S)：又叫读锁(select)，多个事务对于同一数据共享一把锁，可以访问数据，但是不能修改数据，用SQL语句select xxx lock in share mode实现

排他锁(X)：又叫写锁(update，delete，insert)，排他锁指的是一个事务在一行数据加上排他锁后，其他事务不能再在其上加其他的锁(而不是锁住一行数据后，其他事务就不能对数据进行读写操作)，可以用SQL语句select xxx for update实现。

注意：update，delete，insert会自动添加排他锁，select语句则不认不会添加其他任何锁，添加排他锁select \*\*\* for update，加共享锁select \*\*\* lock in share mode，加过锁的数据行在其他事务是不能修改数据的，但是可以直接通过select \*\*\* from \*\*\*查询数据，因为普通查询没有任何锁机制

锁是加载索引上的，所以两个事务在update/delete的时候因为选择了不同索引造成死锁

以下几种情况：

事务A执行select \* from user where id=1 for update时，会对user表中的id为1的行上排他锁，当其他事务带锁(排他锁/共享锁)来操作此行时，只有等待，等事务A释放排他锁后，其他事务才能执行，但是可以执行select \* from user where id=1会读取出记录，因为单纯的select是不带锁的

事务A执行select \* from user where id=1 for update但还未提交，事务B执行select \*\*for update/select \*\* lock in share mode/update/insert/delete带锁的操作时，都会等待事务A释放锁，释放锁后，事务B才执行，如果事务B执行select \* from user where id=1时会直接返回结果(不带锁)

事务A执行select \* from user where id=1 lock in share mode但还未提交，事务B执行select \* from user where id=1直接输出结果，执行update/insert/delete操作等事务A释放锁(共享锁，只读不写)，执行select \*\*\* lock in share mode会报错

表锁(意向锁)：

意向共享锁(IS)：事务打算对数据行添加共享锁时，需要在行锁在的表添加意向共享锁。

意向排他锁(IX)：事务打算对数据行添加排他锁时，需要在行锁在的表添加意向排他锁。

如果事务A只是使用select \*\*\*，事务B可以对数据进行写操作，可以使用for update/lock in share mode来加锁

当前事务拥有锁的判断：

事务A执行如下SQL语句：当执行语句1时，当前事务的锁为共享锁，当执行语句2时，当前事务的锁为排他锁

BEGIN;

SELECT \* FROM USER WHERE id=1 lock in share mode;//语句1

UPDATE USER SET height=333 WHERE id=1;//语句2

COMMIT;

InnoDB的行锁是通过给索引加锁的，只有通过索引来获取行的时候才是行锁，否则是表锁 所以在使用where语句的查询条件时，使用索引作为查询条件

事务A执行select \* from user where name=”jack” for update但还未提交，事务B执行select \* from user where id(索引列)=1 for update，事务B的执行要等事务A执行完才能执行，事务A拥有意向排他锁 ，事务B拥有排他锁和意向排他锁，意向排他锁和意向拍他锁都是排他锁，所以事务B等待，如果事务A是意向共享锁，事务B是共享锁则事务B不用等待，

事务A拥有的锁(表锁/行锁)，拥有行锁就拥有对应的表锁，当事务B去执行事务A对应的表时首先用事务A的表锁和事务B的表锁去判断(事务一定有表锁)，如果互斥(共享/排他)，就等待，如果不互斥，就进行行锁的判断

共享锁和共享锁兼容，共享锁和排他锁互斥，排他锁和共享锁互斥，排他锁和排他锁互斥，意向锁也是这样判断

拥有了(S X IS IX)后，就可以解决不可重复读的问题了，但是任然不能解决幻读(MVCC)

间隙锁

一个表有1到101行记录，当事务A执行select \* from user where id>100 for updae;如果只是给100和101行记录加排他锁，如果事务B执行insert语句时，就会在表中添加102行记录(排他锁锁的是100和101行记录)，事务A再执行相同语句时，就会发现结果不同，会出现幻读。而间隙锁的意思是在查找的条件范围内(count(\*) > < between and)加锁，这个锁就叫间隙锁，但是这样会操作严重的锁等待，所以在查询的时候最好使用相等条件来获取，注意，如果在select \*\*\*for update在查询一个表中不存在的行时，也会对此行加间隙锁，其它实现想在此行insert只有等释放锁后

数据库的隔离级别越高，产生锁冲突的几率越高，占用的资源也越多

事务和行锁是选InnoDB的理由，但是在某些情况下可以使用表锁

a 事务需要更新表的大部分数据或全部数据，表又比较大

b 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

死锁：

在InnoDB的是一个事务中，锁是逐步获得的，这样就可能会导致死锁

事务A执行

select \* from table1 where id=1 for update(排他锁)

select \* from table2 where id=1 for update；

事务B执行

select \* from table2 where id=1 for update，

select \* from table1 where id=1 for update；

当事务A执行语句1的时候，table1的id=1的行锁住，当事务B执行语句1的时候，table2的id=1的行锁住，当事务A执行语句2的时候，发现有锁，等待，事务B执行语句2时，发现有锁，等待，这样就造成了死锁

发生死锁后，InnoDB一般会自动检测到，并释放一个事务，使另外一个事务获得锁，但是在涉及外部锁，表锁的情况下InnoDB并不会自动检查死锁，可以为事务设置一个等待超时innodb\_lock\_wait\_timeout，当事务在一定时间内未执行完，就回滚并释放锁，死锁造成的原因是应用程序设计的问题，大多数死锁都是可以避免的

避免死锁：

两个事务访问两个表的顺序不同，发生死锁的几率非常高，如果以相同的顺序访问，就可以避免死锁(执行顺序)

数据库事务隔离级别的选择(使用较低的隔离级别)

设计索引，尽量使用索引访问数据库，使加锁更精确

选择合理的事务大小(一个事务内执行SQL语句的条数)

一次性请求足够级别的锁，比如事务A要修改数据，直接申请排他锁，不要先共享再排他(一个事务内有多条SQL)

尽量用相等条件访问数据，避免间隙锁对并发插入的影响

不要请求超过实际的锁，比如，在查询的时候就申请共享锁，不要申请排他锁

特定事务可以使用表锁

<https://blog.csdn.net/xifeijian/article/details/20313977>

MVCC：

redo log：它是保存执行SQL语句的log文件，当执行SQL语句时，redo log(可以理解为SQL语句中的数据)写入到log buffer中，当commit的生活，log buffer中的数据会被刷新到磁盘

undo log：在执行SQL语句之前，将当前的数据copy到undo buffer中(备份原数据)，如果事务回滚，则将undo中的数据刷新到磁盘上

为了实现事务的隔离级别，采用锁机制，但是创建锁和销毁锁会都消耗性能，而MVCC没有采取锁的而是采用时间戳的机制实现事务的隔离级别，优化了性能

InnoDB为每行隐藏了3个字段DB\_TRX\_ID，DB\_ROLL\_PTR，DB\_ROW\_ID

DB\_TRX\_ID：当事务执行时，自增

DB\_ROLL\_PTR：回滚指针，指向undo log

DB\_ROW\_ID：当数据更新时，为当前的DB\_TRX\_ID，只有写操作时才更新

事务1插入3条记录

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Name | DB\_TRX\_ID | DB\_ROW\_ID |
| 1 | A1 | 1 |  |
| 2 | A2 | 1 | 1 |
| 3 | A3 | 1 | 1 |

事务2更新id=1的记录数次

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Name | DB\_TRX\_ID | DB\_ROW\_ID |
| 1 | A1 | 8 | 8 |
| 2 | A2 | 1 | 1 |
| 3 | A3 | 1 | 1 |

事务3执行select操作，事务获取到DB\_TRX\_ID为8(DB\_TRX\_ID最大的值)

Begin；

Select \* from test；

Select \* from test；

事务4执行insert操作操作, 事务获取到DB\_TRX\_ID为9

|  |  |  |  |
| --- | --- | --- | --- |
| Id | Name | DB\_TRX\_ID | DB\_ROW\_ID |
| 1 | A1 | 8 | 8 |
| 2 | A2 | 1 | 1 |
| 3 | A3 | 1 | 1 |
| 4 | A4 | 9 | 9 |

事务3在执行第一个select语句的时候，查询结果为1,2,3，在执行第二条select语句时，事务4执行了insert语句，事务3执行第二条SQL语句，此时事务3的DB\_TRX\_ID为8，只会查询到比自己DB\_TRX\_ID小或等于的记录，则获取的记录任然是1,2,3，这里实现了在读的时候，写入数据了(当然，select未加锁)，读写可以同时进行，并且避免了幻读

MVCC实现了不加锁的情况下，实现了RR，并且读写可以同时进行，避免了幻读(insert/update/delete统称写操作)

MVCC用于解决读写之间的冲突，这也是为啥select语句本身不带锁

<https://blog.csdn.net/whoamiyang/article/details/51901888>

MVCC只解决了部分的幻读问题，事务A执行两次select，事务B如果修改了select中的一行记录，事务第二次执行select读出的数据是更新的，幻读其实看重的是获取记录的条数

SQL：

启动MySQL

net start mysql

关闭MySQL

net stop mysql

插入数据

向指定列插入值

insert into tableName (字段) values (值);

向表中所有列插入值

insert into tableName values ();

从源表中筛序符合条件的记录，插入到指定表中

insert into tableName1 (字段) select 字段 from tableName2 where ..

批量插入：insert into tableName (字段) values (值),(),()...

删除数据

delete from tableName where ..

更新语句

update tableName set 字段=值 where ..

批量更新：

update test set name="newName" where id in (1,2,3,4);

update test

set name =case id

when 1 then "newName1"

when 2 then "newName2"

when 3 then "newName3"

end

where id in (1,2,3)

如果id为1，则name更新为newName1

replace into test (id,name) values (1,'newName1'),(2,'newName2');(性能好)

查询语句

select 字段 from tableName where ..

select \* from tableName where id>10;#(>,<,!=...)

select 字段 from tableName where id>10 OR name="jack"#(OR,AND)

select 字段 from tableName where (..OR..)AND(..OR..)

select 字段+字段-字段 from tableName;#还可以做指定运算

select distinct 字段 from tableName;#查询每个唯一的记录输出(可能检索出的记录会重复),去重查询

select 字段 from tableName where ..order by id desc、asc;#对查询结果进行排序，asc升序

select \* from tableName limit 0,5;分页查询

select YEAR(birth) from tableName;#YEAR()函数提取年，MONTH(),DAYOFMONTH()

select count(\*) from tableName;#查询总记录数

select max(price) ffrom tableName where ..;max,min,sum,avg

常用函数：

lower(str):将str转化为小写(upper)

concat(str1,str2...)：将字符串连接起来

length(str)：求字段长度

substr(str,start,length)：截取指定str长度

where 判断条件：

>,<,<=,>=,=,!=

between..and..

in(100,200):查询条件在某个集合内，not in

模糊查询

like/not like:

like'%zhang\_' % 表示零个或任意多个字符，\_表示一个字符(转义字符ESCAPE)

select \* from user where name like ‘%zheng\_’ ESCAPE ‘\_’;

正则表达式

REGEXP：SELECT \* FROM test WHERE name REGEXP '[A-Z][0-9](正则表达式)';

is null：查询条件不为空

and,or,not

in/not in：

select \* from user where name in (select name from user where id in/not in (1,2,3))

is null/is not null：select \* from user where name is null/is not null;

exists/not exists：存在量词

union(并)/intersect(交)/except(差)

聚合查询：

select 列 聚合函数 from tableName where .. group by 列 having ..;

group by语句用于结合聚合函数，根据一个或多个列对结果集进行分组

select name,count(\*) from user group by name;

使用group by将name相同的列聚合在一起，然后根据聚合函数对聚合的结果操作

上就是将相同name聚合在一起，然后获取聚合的条数

SELECT NAME ,MAX(score) FROM test GROUP BY NAME;

获取name的最大score

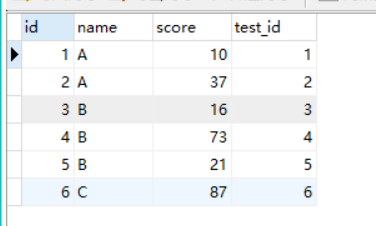
聚合函数：sum(),avg(),count(),max(),min()

select name,count(\*) from user group by name having count(\*)>2;

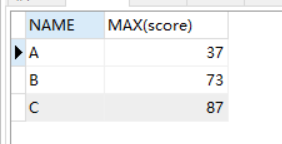
having：对聚合后的结果进行筛选

where是在聚合之前，having是在聚合之后

test表



SELECT NAME ,MAX(score) FROM test GROUP BY NAME;



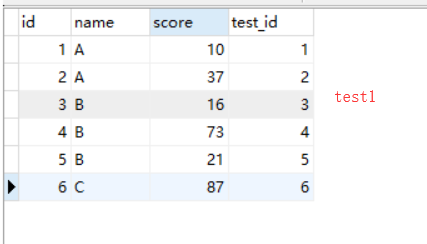
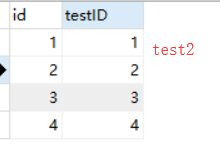
连接查询：

内连接/外连接：内连接查询出两张表相互匹配的记录

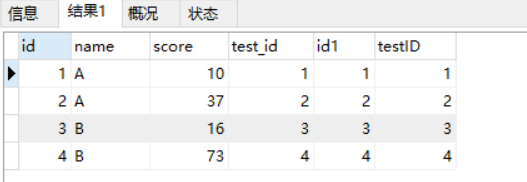
内连接：

select 字段 from tableName1 inner join tableName2 on tableName1.字段名=tableName2.字段名

等同于 select 字段 from tableName1,tableName2 where tableName1.字段名=tableName2.字段名

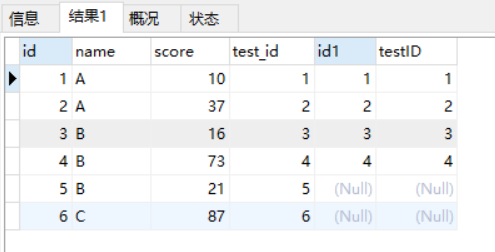
SELECT \* FROM test INNER JOIN test2 on test.test\_id=test2.testID;



外连接：

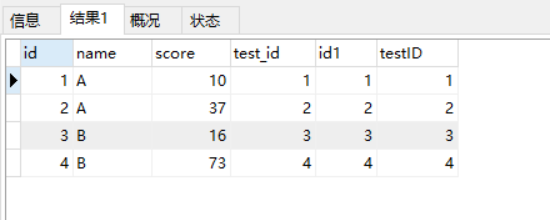
左外连接：左表(test)的test\_id有1,2,3,4,5,6则根据此到右表(test2)中查询，无论有没有记录，都返回

SELECT \* FROM test LEFT JOIN test2 on test.test\_id=test2.testID



右外连接：右表(test2)的testID有1,2,3,4则根据此到左表查询，可以修改表的顺序来进行左/右外连接查询

SELECT \* FROM test RIGHT JOIN test2 on test.test\_id=test2.testID



视图和存储过程：

视图就是个虚表，将多个表通过SQL语句整合在一个，而存储过程类似一个函数，可以多查询的结果经行处理，一个是图(结果集)，一个是过程(函数，有具体的执行过程)

Explain：

模拟Mysql优化器是如何执行SQL查询语句的，从而知道Mysql是如何处理你的SQL语句的。分析你的查询语句或是表结构的性能瓶颈。帮助写出更优化的查询语句

<http://www.cnblogs.com/yycc/p/7338894.html>

<http://mp.weixin.qq.com/s/213sPk-0RtWwwX6YotN6eQ>

SQL优化：

如果索引的字段可以为NULL，那么数据库在查询的时候不会走索引，而是扫描全表，而为什么索引不能为NULL，如果索引为空了，B+Tree还如何判断，所以要设置一个默认值

在设计表的时候在，不要有NULL，如果为空，设置一个默认的值

避免使用!=,>,< 会放弃使用索引

避免使用OR，这使用union，会导致引擎放弃使用索引而进行全表扫描(OR AND 使用union/intersect/except代替)

select \* from t where  num=10 or num=20--->select id from t where num=10 union all select id from t where num=20

避免使用in,not in能用between..and..就不要用in 会导致引擎放弃使用索引而进行全表扫描

like不要应用在索引上，like ‘%aa%’不会使用索引，like ‘aa%’才会使用索引

排序尽量用升序

IN用EXISTS代替，NOT IN用NOT EXISTS

select name from student where Sid in (select Tid from teacher where Tid<2);

select name from student where exists (select Tid from teacher where Tid<2 and Tid=student.Sid);

in一般使用在多个表之间使用

不要在where 语句中使用函数/表达式，这样会导致放弃使用索引：select \*\*\* where YAER(bir)<2007;

避免使用子查询

表的字段越短越好

事务的回滚：

当提交的事务出错时，可以通过rollBack会滚动操作，也可以回滚到事务的中间点位置：roollback to 中间点名(save point)

MySQL查询过程：

1，客户端向MySQL服务器发送一条查询请求

2，服务器首先检查查询缓存，如果命中缓存，则立刻返回存储在缓存中的结果。否则进入下一阶段

在缓存内部有一个缓存SQL语句的Hash表，对应的缓存结果，要求SQL语句必须完全一样，连空格都不能多一个，更不能在SQL语句中使用函数，当在写操作时，缓存将会关闭，当缓存对应的表更新后，缓存丢失

3，服务器进行SQL解析、预处理、再由优化器生成对应的执行计划

4，MySQL根据执行计划，调用存储引擎的API来执行查询

5，将结果返回给客户端，同时缓存查询结果

<https://mp.weixin.qq.com/s/UIVDpgBafCKO87Td1W4pXg>

数据库事务在commit的时候，如果断电了会怎样，数据库在执行SQL语句时，会先记录下这个事务的redo操作日志，然后才真正开始操作数据库，在操作之前会将日志写入到磁盘上(外存)，那么断电后，即时没有commit，重启数据库后，数据库会去读取日志文件进行undo或redo回滚，保证了数据的一致性

数据库是分为数据文件和日志文件的，日志文件记录的是SQL语句，数据库在执行SQL语句时，先将要执行SQL的相关数据写入到日志中，当commit后，会将日志文件(外设)中数据写入到数据文件(外设)中，当写入成功后，才会将日志文件移出，所以无论是哪里出错，都会保证数据库的一致性