Sql知识

1. **基础知识**

aa

1. **增删改查**

插入数据：

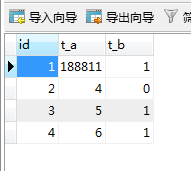
表结构：id(自增长)，name,age

insert into user(name,age) values('abc',11);

数据迁移

insert into tab1 select \* from tab2 where …

数据存在则更新，不存在则插入（插入数据）



这里的id是主键，如果主键是自增长的话，在插入的时候没有设置id的值，那么下面两种方法的操作都不会成功，都会增加一条数据，而不会去更新

方法1：使用replace

REPLACE INTO customer(id,t\_a,t\_b) VALUES(1,188811,1)

方法2：

INSERT INTO customer(id,t\_a,t\_b) VALUES(1,188811,1) ON DUPLICATE KEY UPDATE t\_b=188811

查询数据：

select \* from tab;

模糊查询，like ‘%field%’

当某条件有值时用获取的值，没有值的时候用默认值的查询（数据库注入操作）,当status有值时使用传来的值，如果没有则使用默认的0值

select \* from tab where status = ? or ? = 0

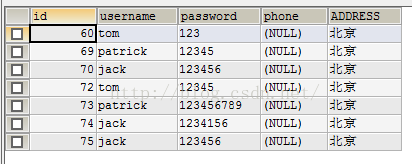
更新数据：

update user set name=’abc’ where id=123;

**删除重复数据：**

删除重复语句并保留一条数据的SQL

表数据结构：user表



SQL语句：

DELETE FROM `user` WHERE id NOT IN(SELECT \* FROM(SELECT id FROM `user` GROUP BY username)AS b)；

1. **根据某个字段判断数据是否存在重复**

SELECT \* FROM MC\_REGION GROUP BY REGION\_NAME HAVING COUNT(1)>1;

以上句子的意思是查询表MC\_REGION中根据字段REGION\_NAME分组，是否存在同REGION\_NAME的数据。也就是判断数据是否存在重复的。

1. **数据库所在服务器磁盘满了**

**问题描述**：如果数据库客户端能连上数据库，但是不能读取数据库数据，每次读取的时候都会报错，而且连接其他数据库的时候不会报错。

原因：1.可能是数据库所在磁盘满了，这个时候读取数据库数据就会报错。

解决方法：删除磁盘的部分数据，为数据库读取数据提供必要的空间。

1. **sql查询结果将两字段用某种规则拼接起来**

**描述**：查询数据库的时候，我们想将得到的两个字段值按某种方式拼接成一个字段。

**方法**：concat(id,'-',siteid)，id是第一个字段，中间的字符是按某种方式拼接（这个参数是可选的），第三个参数是第二个拼接的字段。

1. **查询数据库中有多少表**

SELECT tb.TABLE\_NAME as tabName FROM information\_schema.TABLES as tb WHERE table\_schema = 'boar-message' and TABLE\_NAME LIKE 'tbl\_queue1\_d%';

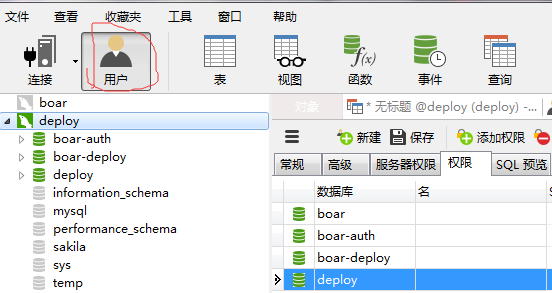
#筛选出某个数据库的某些表名满足条件的数据，并返回表名

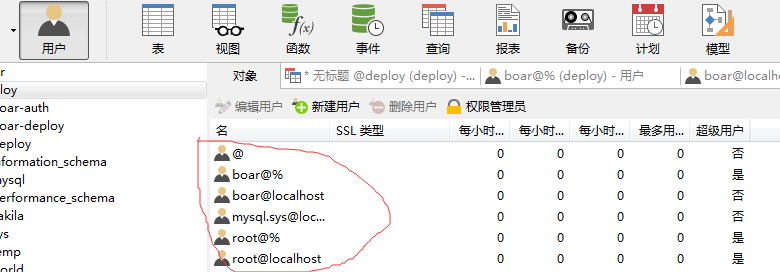
#boar-message数据库名

#TABLE\_NAME 表名

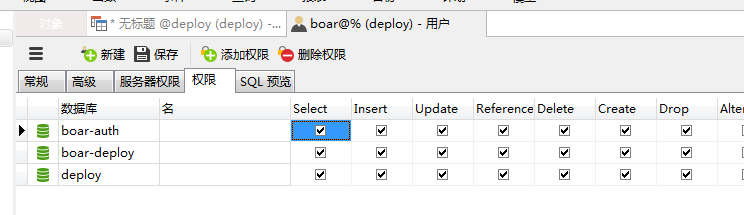
1. **navicat新增用户和配置权限**

给某个用户配置相关权限

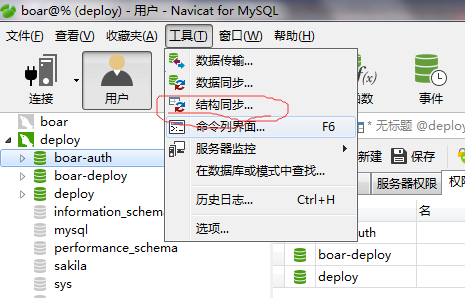


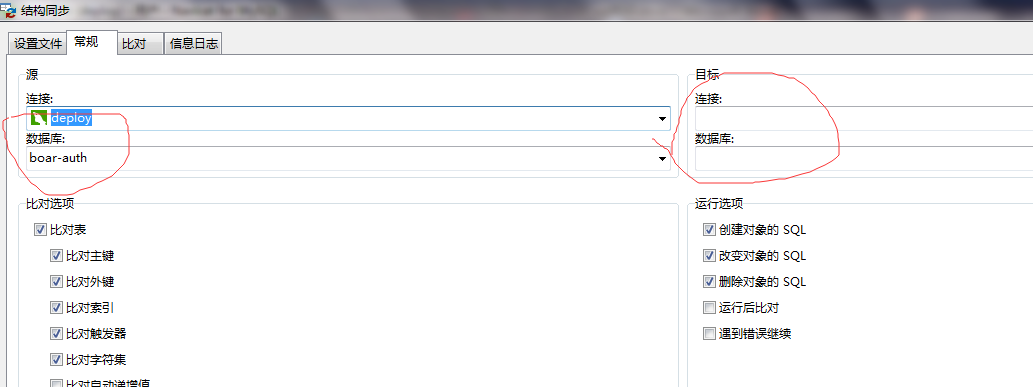


选中用户，点编辑用户



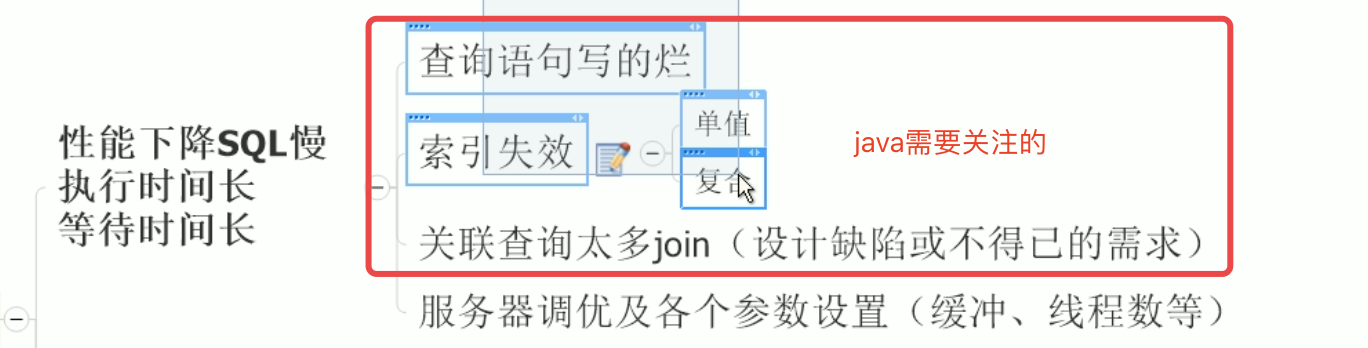
1. **navicat比对两个库表的结构不同**





注：还可以数据同步，操作参考结构同步

1. **索引**



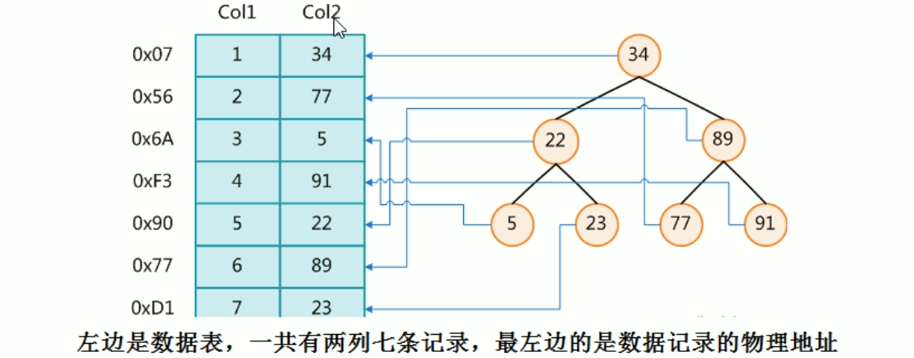
索引概念：

官方定义：索引index是帮助sql高效获取数据的数据结构，可以得到索引的本质—索引是数据结构（排好序的的快速查找数据结构）。索引的目的在于提高查询效率，可以类比字典，如果要查“ghong”这个单词，我们肯定需要定位到g字母，然后再往下找到h字母，再找到剩下的ong。

索引一般都是用B-tree索引。

索引主要对查找和排序有影响。

数据库一般在存储数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据（也就是数据的存储地址—指针），这样就可以在这些数据结构上实现高级查找算法，这种数据结构，就是索引。下图就是一种可能的索引方式示例：



为了加快上图Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在一定的复杂度内获取到相应数据，从而快速的检索出符合条件的记录。

如果一张表频繁的增删改，该表建的索引可能会失效，因此dba可能会随时去重建索引，要不然查询可能就很慢，频繁的删改的字段不适合建索引（因为每次修改都会修改索引）。

一般来说索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储在磁盘上。

实际上索引也是一张表，该表保存了主键索引字段，并指向实体表的记录，所以索引列也是要占用空间的。

优点：提高数据检索的效率，降低数据库的IO成本，通过索引列对数据进行排序，降低数据排序成本，降低CPU的消耗

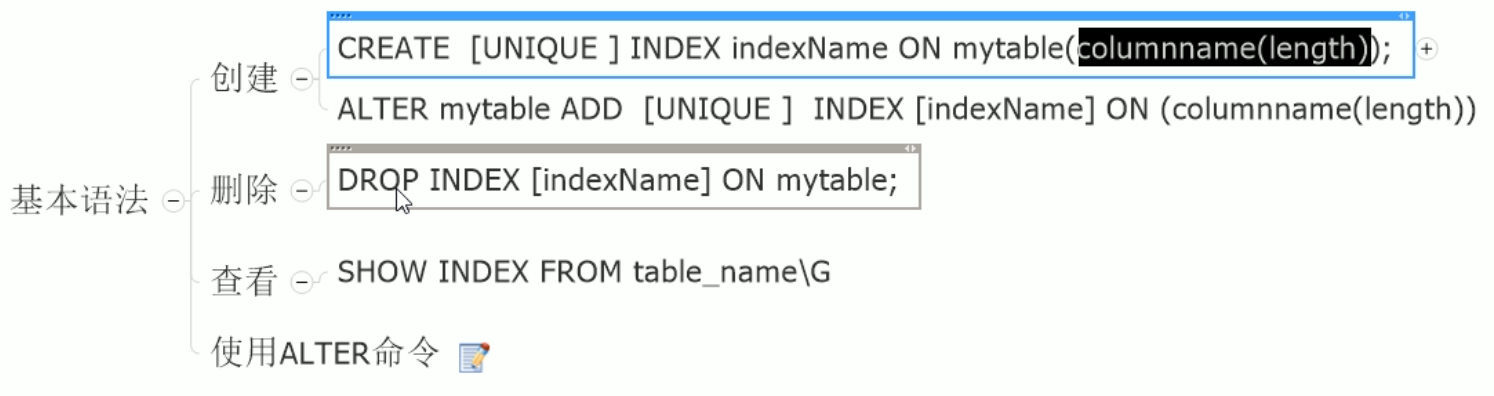
缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行insert，update，delete，因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件，每次更新添加了索引列的字段，都会调整因为更新所带来的键值变化后的索引信息。

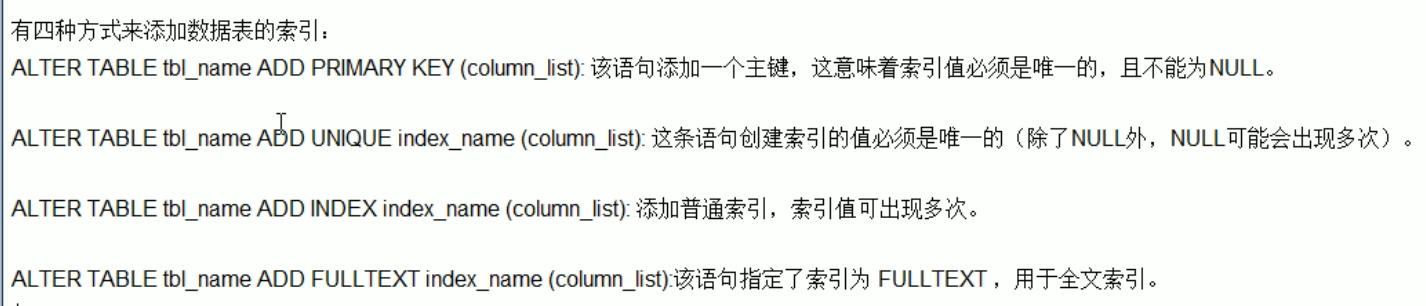
索引分类：

单值索引：一个索引只包含单个列，一个表可以有多个单值索引

唯一索引：索引列的值必须唯一，但允许有空值

复合索引：一个索引包含多个列

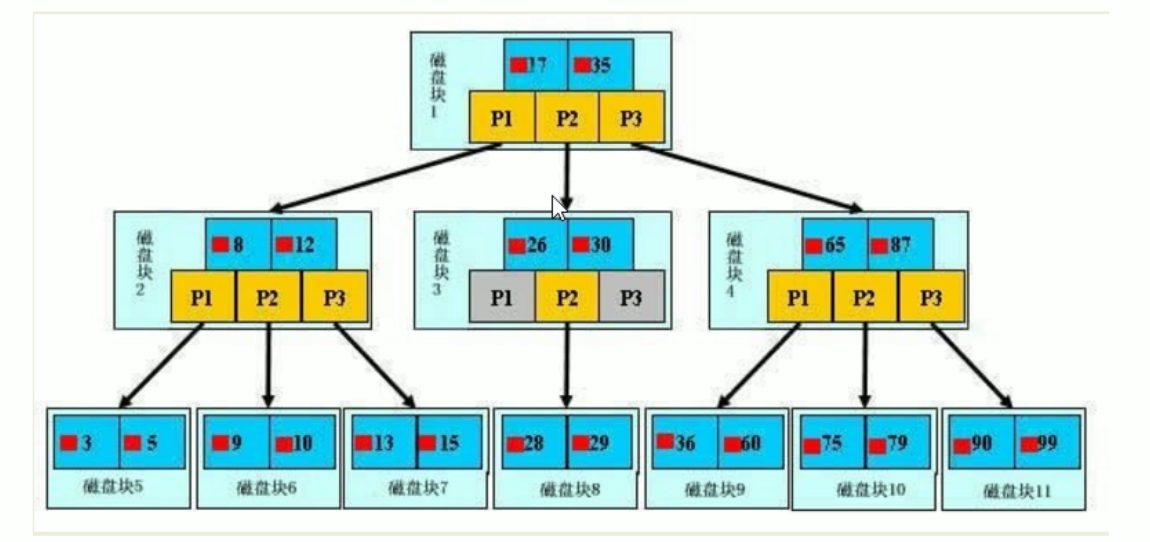


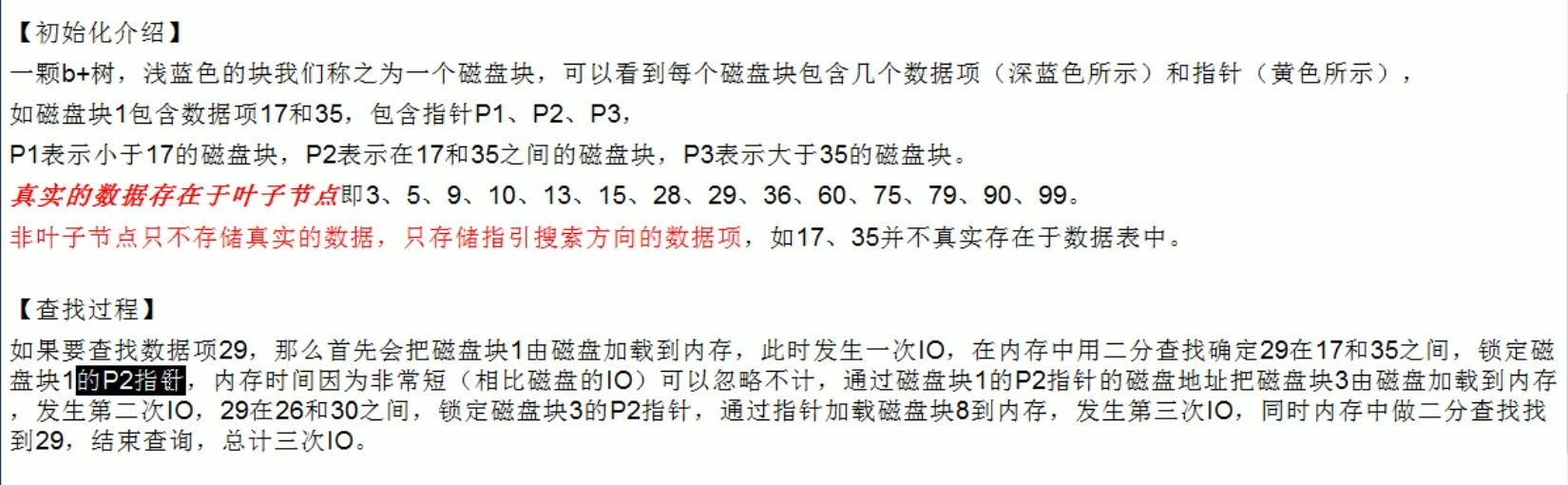


红色是着重要了解的

索引结构：BTree索引；Hash索引；full-text全文索引；R-Tree索引

b-tree结构：检索原理





哪些情况适合建索引？

1. 主键自动创建索引
2. 频繁作为查询条件的字段应该创建索引
3. 查询中与其他表关联的字段，外键关系建立索引
4. 频繁更新的字段不适合创建索引—因为每次更新不单单是更新记录还会更新索引
5. where条件里用不到的字段不创建索引
6. 单键/组合索引的选择问题？（高并发下倾向创建组合索引）
7. 查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度
8. 查询中统计或者分组的字段

哪些情况不要建索引？

1. 表记录比较少，万条以下
2. 经常增删改的字段不要建索引
3. 如果某个列包含很多重复的值，也就是选择性比较低；索引的选择性是指索引列中不同值的数目与表中记录数的比。如果一个表中有2000条记录，表索引列有1980个不同的值，那么这个索引的选择性就是1980/2000=0.99。一个索引的选择性越接近于1，这个索引的效率就越高。

**单表建索引：**

如果字段查询时加上范围，则该字段最好不建索引，比如：

select \* from user where age > 20 and name = ‘gh’order by phone;

索引建在name和phone上，age不建索引

**双表连接建索引：**

左连接:索引建在右表；右连接：索引建在左表

**索引优化：**

尽可能减少join语句中的nestloop的循环总次数，永远用小结果集驱动大的结果集

优先优化nestloop的内层循环

保证join语句中被驱动表上join条件字段已经被索引。

**索引失效：应该避免这种情况**

索引失效情况：

（1）全值匹配：也就是索引有多少个字段，select的时候就有相同字段，并且where顺序是一样的。

（2）最佳左前缀法则：比如user表的复合索引是由（id,name,age）三个字段建立的索引。我们在select where的时候，只有name或者age会发现没有用到索引，这个时候一定要加上第一个字段id作为查询条件。如果单独只有id字段也可以用到索引。如果select where的时候id=? and age=？没有中间项name，这个时候索引也是失效的。

（3）不在索引列上做任何操作（计算、函数、自动or手动类型转换），会导致索引失效而转向全表扫描

（4）存储引擎不能使用索引中范围条件右边的列

比如select \* from user where name=’gh’and age>25 and phone=’187…’

索引是复合索引，包含（name,age,phone）三个字段

上面的sql中age使用了范围大于25，因此后面的phone字段的索引就失效了。

（5）尽量使用覆盖索引（只访问索引的查询：索引列和查询列一致），减少select \*

（6）MySQL在使用不等于（!=或者<>）的时候无法使用索引会导致全表扫描

（7）is null或者is not null也无法使用索引

（8）like以通配符开头（%abc）MySQL索引失效会变成全表扫描；‘%abc%’，‘%abc’都会全表扫描，‘abc%’则会用到索引

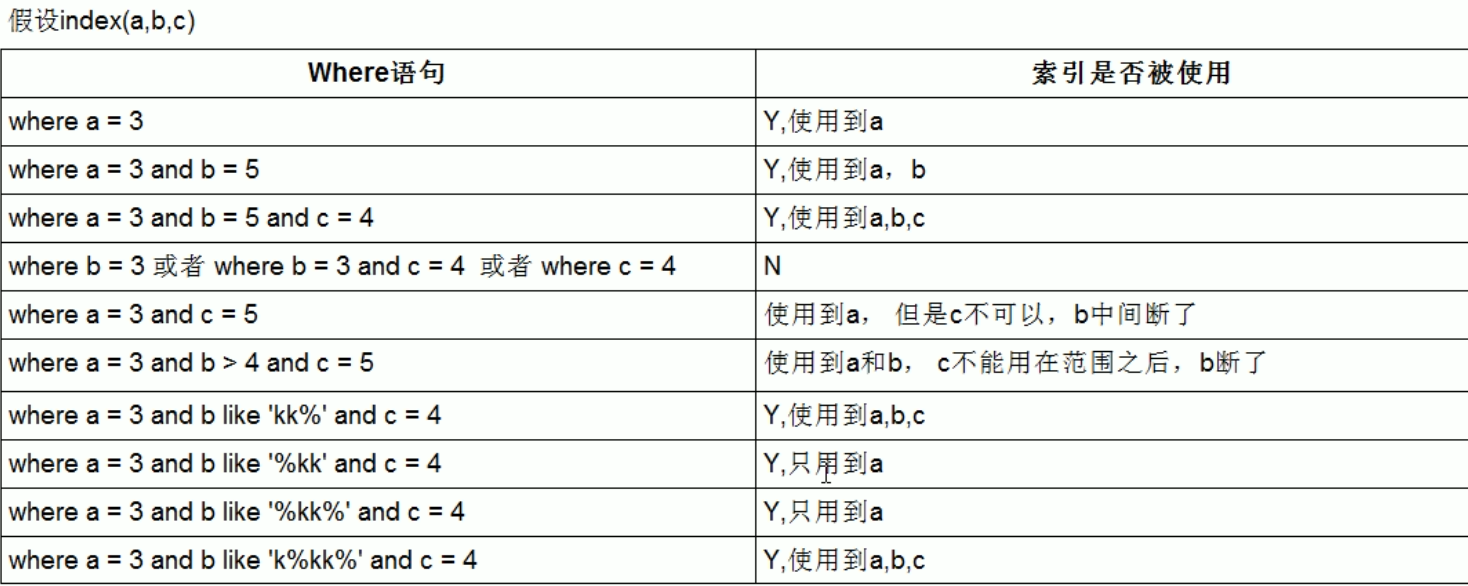
如何避免‘%abc%’全表扫描？

有个user表（id,name,age，phone）字段，新建age，name复合索引，然后select age,name from user where name like ‘%abc%’此时就会用到索引，但是select \* from user where name like ‘%abc%’索引会失效

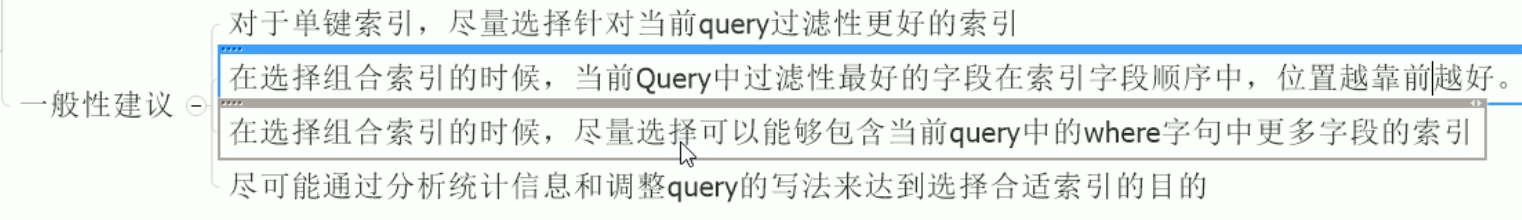
（9）字符串不加单引号，索引失效。

比如select \* from where name=2000;name是varchar类型

（10）少用or，用它来连接时索引会失效



group by的字段，一定要跟索引的字段顺序一样.group by基本上都需要进行排序，会有临时表产生。



**FULLTEXT全文索引：**

MySQL从3.23.23版开始支持全文索引和全文检索。在MySQL中，全文索引的索引类型为FULLTEXT。全文索引可以在VARCHAR或者TEXT类型的列上创建。它可以通过CREATE TABLE命令创建，也可以通过ALTER TABLE或CREATE INDEX命令创建。对于大规模的数据集，通过ALTER TABLE（或者CREATE INDEX）命令创建全文索引要比把记录插入带有全文索引的空表更快。

全文索引是采用分词技术。

全文索引和普通索引的区别：



与一般的索引立即更新不同，全文索引一般是定期维护索引的，所以对于频繁更新的数据不合适，需要做全文索引的对象一般都是论文网页之类

表的存储引擎是MyISAM，默认存储引擎InnoDB不支持全文索引（新版本MYSQL5.6的InnoDB支持全文索引）

全文索引支持的字段类型：char、varchar和text

添加：ALTER TABLE table\_name ADD FULLTEXT ( column);

删除：DROP INDEX index\_name ON table\_name;

使用全文索引：

SELECT \* FROM table\_name WHERE MATCH(index\_name) AGAINST(‘搜索值’);

默认MySQL不支持中文全文检索，怎么办？大致方法有下面几个：

A. 扩展MySQL，添加中文全文检索支持，难度较大

B. 为中文内容表提供一个对应的英文索引表（即将FULLTEXT索引列按照一定的规则转化成英文索引表中的每一条记录，比如全部进行base64编码，内容表和英文索引表的id相同），检索时先将检索词也用相同规则转换成英文，然后再使用。如果还要支持按拼音全文检索，那么还需要在索引表中增加对应的拼音内容（就需要中文转拼音算法了）。当然如果还需要支持中英文交互搜索，比如搜索William时也需要返回威廉，反之亦然，那么还需要将威廉对应的英文翻译也存到索引表中去。

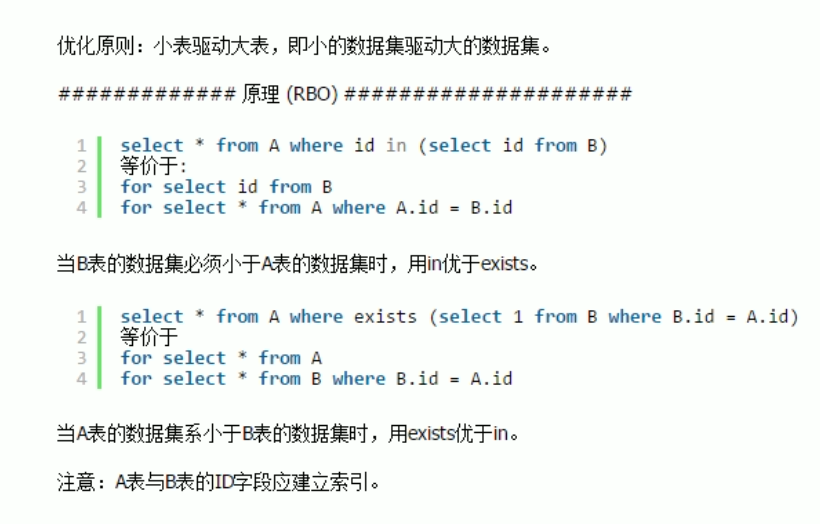
C. 从MySQL 5.7开始，MySQL内置了ngram全文检索插件，用来支持中文分词，并且对MyISAM和InnoDB引擎有效。

备注1：目前，使用MySQL自带的全文索引时，如果查询字符串的长度过短将无法得到期望的搜索结果。MySQL全文索引所能找到的词的默认最小长度为4个字符。另外，如果查询的字符串包含停止词，那么该停止词将会被忽略。

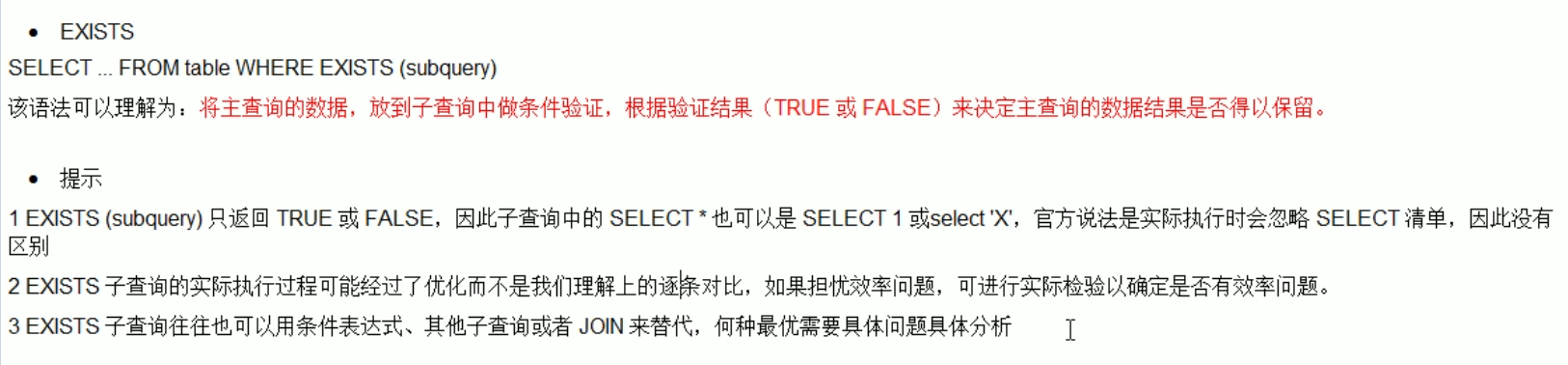
备注2：如果可能，请尽量先创建表并插入所有数据后再创建全文索引，而不要在创建表时就直接创建全文索引，因为前者比后者的全文索引效率要高。

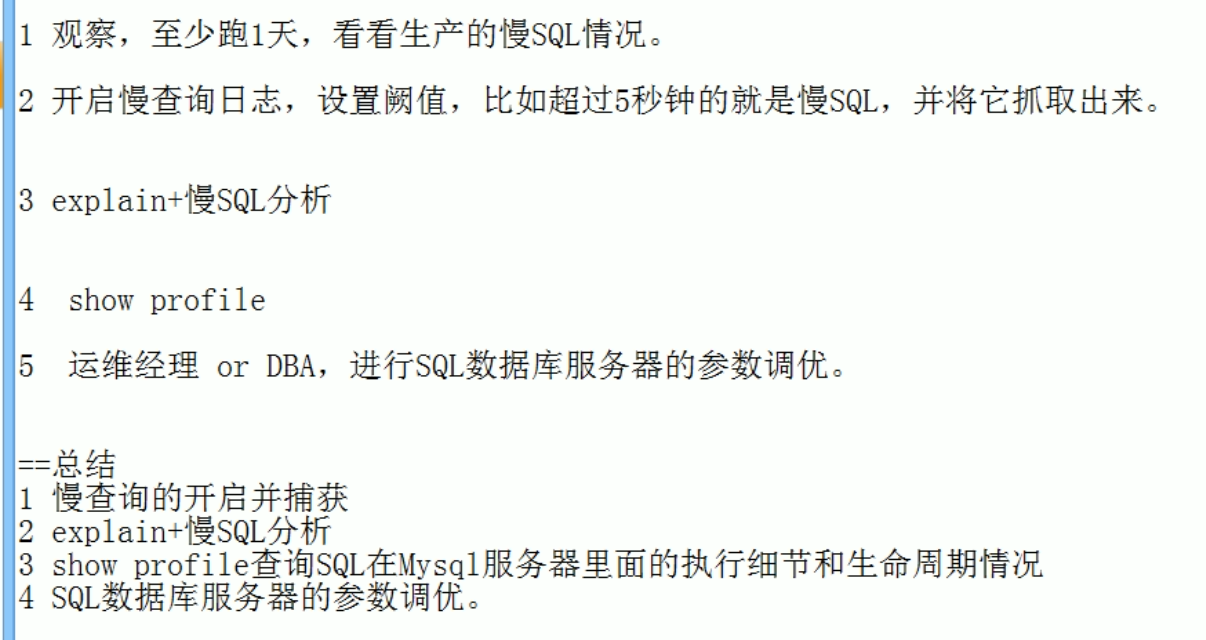
1. **sql调优**

**小表驱动大表**



exists是in写法的一种变种。





**order by子句：**

尽量使用index方式排序，避免使用filesort方式排序, order by的字段，一定要跟索引的字段个数以及顺序一样。

Sql支持两种方式排序：filesort和index，index效率高

Index:是指MySQL扫描索引本身完成排序，filesort方式效率较低。

比如索引index(age,name)

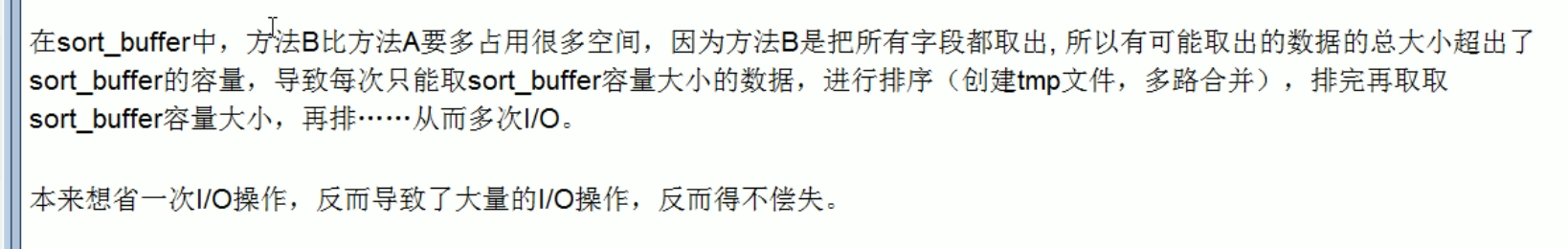
注：order by age asc,name desc这个时候索引失效

filesort排序又分为双路排序和单路排序。

双路排序：mysql4.1之前是使用双路排序，字面意思就是两次扫描磁盘，最终得到数据，读取行指针和orderby列，对他们进行排序，然后扫描已经排序好的列表，按照列表中的值重新从列表中读取对应的数据输出。取一批数据，要对磁盘进行两次扫描，I/O是很耗时的。因此MySQL4.1后出现改进版的单路排序

单路排序：从磁盘读取查询需要的所有列，按照orderby列在buffer对它们进行排序，然后扫描排序后的列表进行输出，它的效率更快一些，避免了第二次读取数据，并且把随机io变成了顺序io，但是它会使用更多的空间，因为把所有的数据都缓存在内存中了

单路排序存在的问题：



解决方法：增大sort\_buffer\_size参数设置并且增大max\_length\_for\_sort\_data参数的设置



**group by语句：**

与order by基本相同，group by实质是先排序后进行分组，遵照索引建的最佳左前缀，当无法使用索引列，增大max\_length\_for\_sort\_data参数的设置+增大sort\_buffer\_size参数的设置，where高于having，能写where限定的条件就不要去having限定了

1. **函数和存储过程**
2. **函数定义**

DELIMITER //

CREATE FUNCTION IF EXIST deleteById(uid SMALLINT UNSIGNED)

RETURNS VARCHAR(**20**)

BEGIN

DELETE FROM son WHERE id = uid;

RETURN (SELECT COUNT(id) FROM son);

END//

DELIMITER // 意思是修改默认的结束符";"为"//",以后的SQL语句都要以"//"作为结尾

特别说明:

UDF中,REURN语句也包含在BEGIN...END中

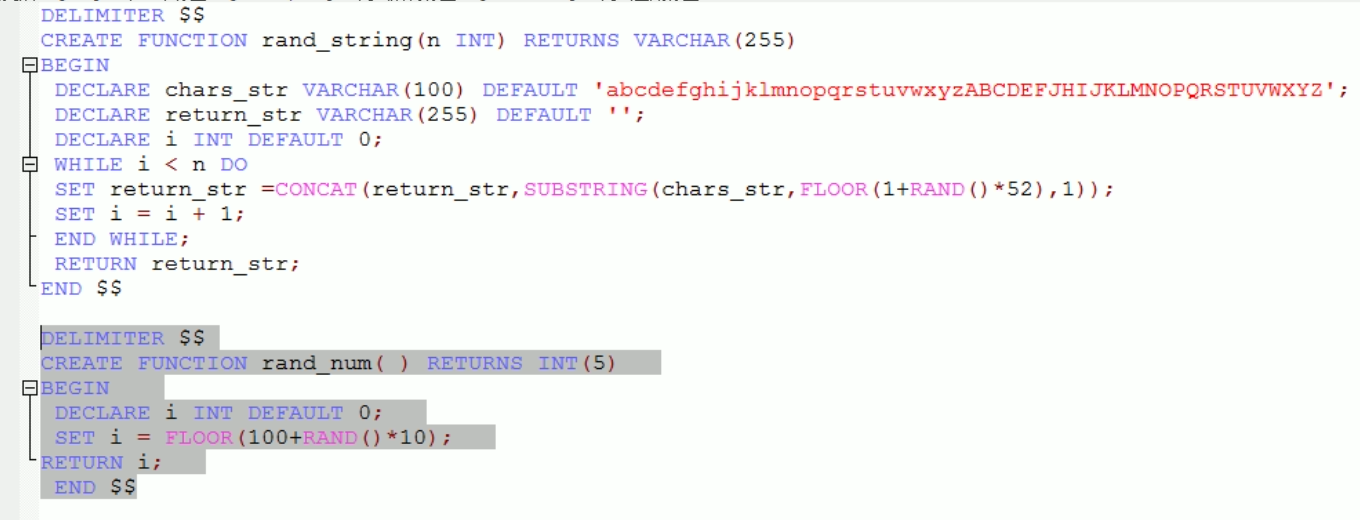
自定义函数中定义局部变量语法:

DECLARE var\_name[,varname]...date\_type [DEFAULT VALUE];

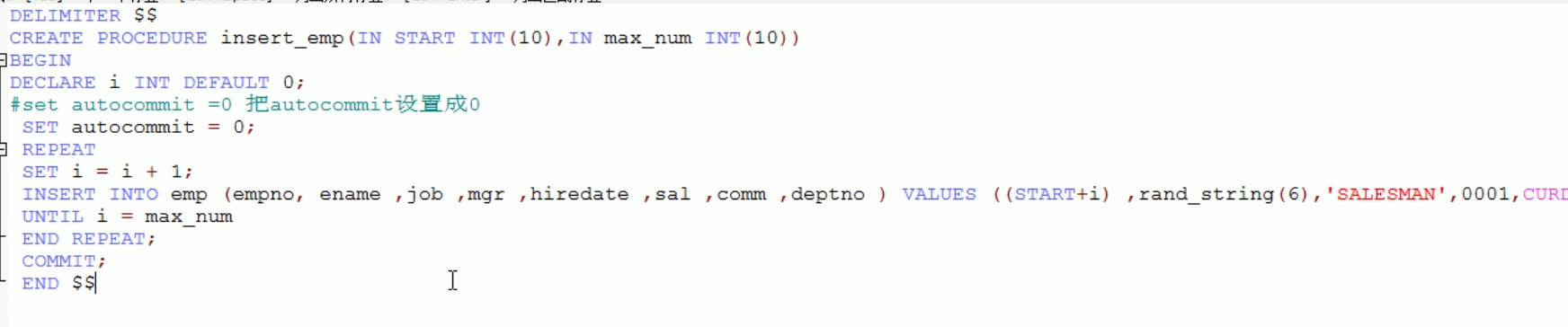
简单来说就是:

DECLARE 变量1[,变量2,... ]变量类型 [DEFAULT 默认值]

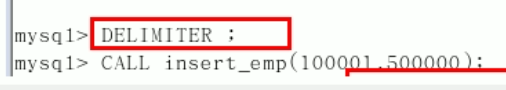
这些变量的作用范围是在BEGIN...END程序中,而且定义局部变量语句必须在BEGIN...END的第一行定义



1. **存储过程**



调用存储过程：



存储过程：procedure就是一段可执行程序，这个程序运行在数据中。

定义一个procedure:

DELIMITER //修改结束的标记

CREATE PROCEDURE proc1()

BEGIN ――{

SELECT \* FROM users;

END // ―― }

DELIMITER ;

CALL proc1();

定义一个**接收参数**的:

DELIMITER $$

CREATE PROCEDURE proc2(IN \_id VARCHAR(32),IN \_nm VARCHAR(30))

BEGIN

INSERT INTO users(id,NAME) VALUES(\_id,\_nm);

END $$

CALL proc2('U003','Rose');

定义一个**返回类型**的参数：

DELIMITER $$

CREATE PROCEDURE proc3(IN \_id VARCHAR(32),IN \_nm VARCHAR(30),OUT \_size INT)

BEGIN

INSERT INTO users(id,NAME) VALUES(\_id,\_nm);

SELECT COUNT(1) INTO \_size FROM users;

IF(\_size=10) THEN

INSERT INTO users VALUES('U0000','ddd','ddddd');

END IF;

END $$

CALL proc3('U002882','Rose',@wj);

SELECT @wj;

**Java调用存储过程**

|  |
| --- |
| @Test  **public** **void** proc1() **throws** Exception{  //dbutils不提供调用存储过程的能力  Connection con = DataSourceUtils.getDatasSource().getConnection();  //获取调用过程的对象  CallableStatement cs =  con.prepareCall("{call proc1()}");  //执行  **boolean** boo = cs.execute();//如果返回true,指最后一句执行的是select语句  **if**(boo){  ResultSet rs = cs.getResultSet();  **while**(rs.next()){  System.***err***.println(rs.getString("name"));  }  }  con.close();  } |

1. **数据库雪崩**

在数据库的使用过程中，一些微妙的操作，在特殊的场景中就可能导致雪崩效应。

出现雪崩的情况大致如下：

（1）当数据库中存在未提交事务，并且未提交事务已经持有了某个表的哪怕是最小的锁时，如果此时对这个对象发起DDL(比如创建数据库或者表)操作，这个DDL操作将会堵塞接下来的其他对该对象的任意操作请求，包括读请求。因为PG的锁等待判断是整个等待队列的冲突判断，这种判断的目的是防止某些锁请求一直被打断，从而这样设计的。

这种情况很容易引起雪崩。（暂时不理解）

（2）业务系统异常，导致数据库请求暴增，从而引起雪崩。这种属于业务系统设计或BUG导致。

（3）主备切换后，热数据不在缓存中，导致请求变慢，如果在高峰期，应用会疯狂递交请求导致雪崩。

（4）执行计划出现异常，导致快SQL变成慢SQL，如果此类请求是高并发的，会直接引起雪崩。

（5）一般是由于某个缓存节点失效，导致其它节点的缓存命中率下降，缓存中缺失的数据直接去数据库查询，短时间内造成数据库服务器崩溃。

（6）由于缓存周期性失效，比如设置每隔6个小时失效一次，那么每6个小时将会有一个请求峰值，严重的话，也会导致数据库崩溃。

如何规避数据雪崩？

（1）语句超时。

使用这种方法，可以有效的避免雪崩，通常雪崩是由业务高峰期的高并发小事务引起的，对这类QUERY加语句超时，可以很好的防止雪崩。

待缓存预热，或者执行计划恢复后，就能恢复正常。

（2）缓存预加载

PostgreSQL提供了缓存快照和缓存预加载的功能，用户在切换主备角色前，可以对缓存进行预热，防止冷数据过多带来的雪崩效应。

（3）缓存有效期随机设置3-9小时之间的一个随机值。

（4）控制缓存在闲时过期（比如夜里）。

1. **on语句和where语句条件**

SELECT order\_id, status, employee\_id, last\_name FROM orders LEFT JOIN employees ON employee\_id = salesman\_id AND order\_id = 58;

比较如上语句和如下语句的区别

SELECT order\_id, status, employee\_id, last\_name FROM orders LEFT JOIN employees ON employee\_id = salesman\_id WHERE order\_id = 58;

order\_id如果加在where后，则是对已经执行左连接的数据做筛选，满足order\_id=58的数据才会被输出。如果order\_id加在on条件后，没有出现在where后，则表示在左连接的时候进行筛选，order\_id属于orders表，此处就是对左连接条件的时候进行筛选，如果左连接双方表都存在order\_id=58的数据则两边都输出数据，如果不存在order\_id=58的数据，则只输出左表数据，相当于是作为连接条件而存在。

1. **in和exist对比**

in:

in在查询的时候，首先查询子查询的表，然后将内表和外表做一个笛卡尔积，然后按照条件进行筛选。所以相对内表比较小的时候，in的速度较快。

如上sql语句先执行子查询

exists:

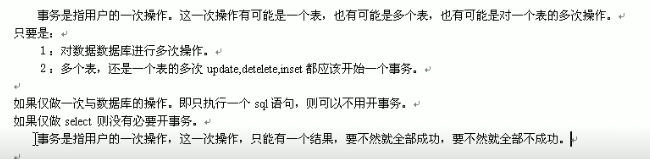
遍历循环外表，然后看外表中的记录有没有和内表的数据一样的。匹配上就将结果放入结果集中，不会继续进行匹配。

in 和 exists的区别: 如果子查询得出的结果集记录较少，主查询中的表较大且又有索引时应该用in, 反之如果外层的主查询记录较少，子查询中的表大，又有索引时使用exists。其实我们区分in和exists主要是造成了驱动顺序的改变(这是性能变化的关键)，如果是exists，那么以外层表为驱动表，先被访问，如果是IN，那么先执行子查询，所以我们会以驱动表的快速返回为目标，那么就会考虑到索引及结果集的关系了 ，另外IN时不对NULL进行处理。

in 是把外表和内表作hash 笛卡尔积连接，而exists是对外表作loop循环，每次loop循环再对内表进行查询。一直以来认为exists比in效率高的说法是不准确的。

如果查询语句使用了not in 那么内外表都进行全表扫描，没有用到索引；而not extsts 的子查询依然能用到表上的索引。所以无论那个表大，用not exists都比not in要快。

1. **事务**
2. **概念**



事务处理：保证所有事务都作为一个工作单元来执行，即使出现了故障，都不能改变这种执行方式。当在一个事务中执行多个操作时，要么所有的事务都被提交(commit)，要么整个事务回滚(rollback)到最初状态

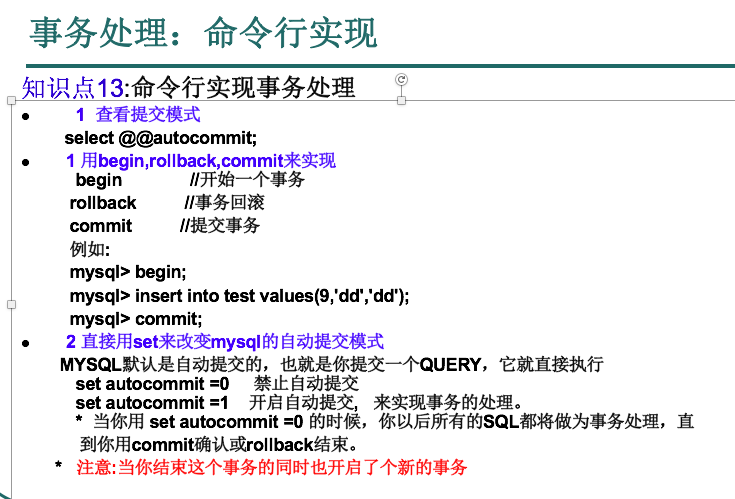
在JDBC中，事务默认是自动提交的，每次执行一个 SQL 语句时，如果执行成功，就会向数据库自动提交，而不能回滚

为了让多个 SQL 语句作为一个事务执行：

* 调用 Connection 对象的 setAutoCommit(false); 以取消自动提交事务
* 在所有的 SQL 语句都成功执行后，调用 commit(); 方法提交事务
* 在出现异常时，调用 rollback(); 方法回滚事务

可以通过Connection的getAutoCommit()方法来获得当前事务的提交方式

1. **事务命令**



1. **事务隔离级别**

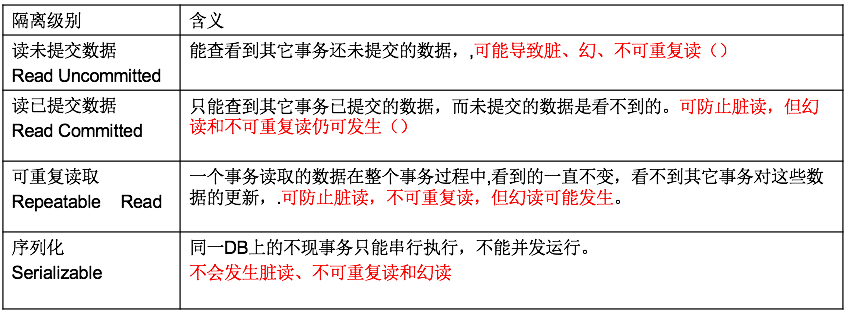
事务隔离级别（transaction isolation levels）：隔离级别就是对事务并发控制的四个等级。分为

1）、串行化(SERIALIZABLE)

2）、可重复读(REPEATABLE READ)

3）、读已提交(READ COMMITED)

4）、读未提交(READ UNCOMMITED)



1. **一般数据库支持的隔离级别**

1、mysql默认的隔离级别为Repeatable\_Read

2、sqlserver 默认的隔离级别为Read Commited

3、oracle数据库支持READ COMMITTED和SERIALIZABLE两种事务隔离性级别，不支持READ UNCOMMITTED和REPEATABLE READ这两种隔离性级别,Oracle数据库默认使用的事务隔离性级别却是READ COMMITTED.

1. **设置数据库隔离级别**



对于同时运行的多个事务,当这些事务访问数据库相同的数据时,如果没有采取必要的隔离机制，就会产生并发问题.

脏读:一个事务读取了另一个事务改写但还未提交的数据,如果这些数据被回滚，则读到的数据是无效的。

不可重复读：在同一事务中，多次读取同一数据返回的结果有所不同。换句话说就是，后续读取可以读到另一事务已提交的更新数据。

可重复读:在同一事务中多次读取数据时，能够保证所读数据一样，也就是，后续读取不能读到另一事务已提交的更新数据。

幻读：一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发现有些原来没有的记录。（在当前事务过程中，在两次查询之间你做了更新，就可能产生）。例如第一个事务对一个表中的数据进行了修改，比如这种修改涉及到表中的“全部数据行”。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入“一行新数据”。那么，以后就会发生操作第一个事务的用户发现表中还存在没有修改的数据行，就好象发生了幻觉一样。

注：事务隔离级别从上而下越来越严格，级别越高，越影响系统的可用性，因此不是级别越高越好。