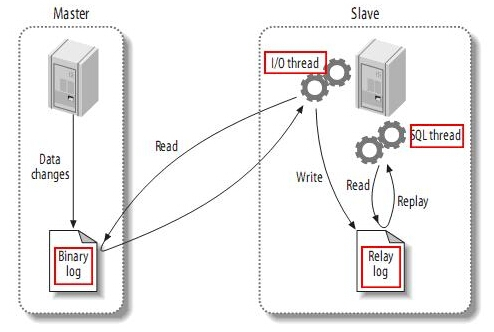
mysql主(称master)从(称slave)复制的原理：

      (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)

      (2).slave将master的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,将改变反映它自己的数据(数据重演)

附简要原理图：

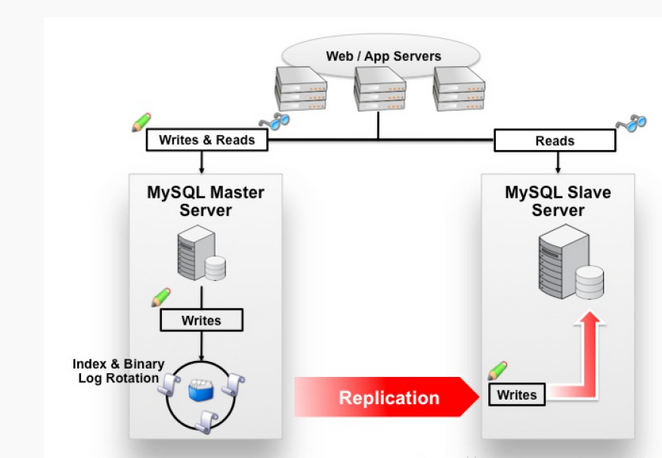


**第一步是在主服务器上记录二进制日志。在每个更新数据的事务完成之前，主服务器都会将数据更改记录到二进制日志中。即使事务在执行期间是交错的，mysql也会串行地将事务写入到二进制日志中。在把事件写入二进制日志之后，主服务器告诉存储引擎提交事务。**

**第二步是从服务器把主服务器的二进制日志拷贝到自己的硬盘上，进入所谓的“中继日志”中。首先，它启动一个工作线程，叫I/O线程，这个I/O线程开启一个普通的客户端连接，然后启动一个特殊的二进制日志转储进程（它没有相应的SQL命令）。这个转储进程从主服务器的二进制日志中读取数据。它不会对事件进行轮询。如果3跟上了主服务器，就会进入休眠状态并等待有新的事件发生时主服务器发出的信号。I/O线程把数据写入从服务器的中继日志中。**

**第三步SQL线程读取中继日志，并且重放其中的事件，然后更新从服务器的数据。由于这个线程能跟上I/O线程，中继日志通常在操作系统的缓存中，所以中继日志的开销很低。SQL线程执行事件也可以被写入从服务器自己的二进制日志中，它对于有些场景很实用。**

半同步复制架构图如下所示：



**半同步复制的概念：**

  1，当Slave主机连接到Master时，能够查看其是否处于半同步复制的机制。

  2，当Master上开启半同步复制的功能时，至少应该有一个Slave开启其功能。此时，一个线程在Master上提交事务将受到阻塞，直到得知一个已开启半同步复制功能的Slave已收到此事务的所有事件，或等待超时。

  3，当一个事务的事件都已写入其relay-log中且已刷新到磁盘上，Slave才会告知已收到。在 Master 实例上，有一个专门的线程(ack\_receiver)接收备库的响应消息，并以通知机制告知主库备库已经接收的日志，可以继续执行。

  4，如果等待超时，也就是Master没被告知已收到，此时Master会自动转换为异步复制的机制。当至少一个半同步的Slave赶上了，Master与其Slave自动转换为半同步复制的机制。

  5，半同步复制的功能要在Master，Slave都开启，半同步复制才会起作用；否则，只开启一边，它依然为异步复制。

  6，半同步特性的出现，就是为了保证在任何时刻主备数据一致的问题。相对于异步复制，半同步复制要求执行的每一个事务，都要求至少有一个备库成功接收后，才返回给用户。

## 一，为什么要使用半同步复制？

MySQL复制默认是异步复制，Mysql Master Server将自己的Binary Log通过复制线程传输出去以后，Mysql Master Sever就自动返回数据给客户端，但并不知道Slave是否或何时已经接收且已处理，因此存在一定的概率备库与主库的数据是不对等的。在异步复制的机制的情况下，如果Master宕机，事务在Master上已提交，但很可能这些事务没有传到任何的Slave上。假设有Master->Salve故障转移的机制，此时Slave也可能会丢失事务。有些情况下需要保持主备库的强一致性，此时启用MySQL的半同步复制特性则是非常完美的。semi\_sync\_replication是google为mysql开发的一个基于半同步的补丁，从mysql5.5之后，mysql为了保证主从库数据一致性，引进了semi-sync功能。

在半同步复制的架构下，当master在将自己binlog发给slave上的时候，要确保slave已经接受到了这个二进制日志以后，才会返回数据给客户端。对比两种架构：异步复制对于用户来说，可以确保得到快速的响应结构，但是不能确保二进制日志确实到达了slave上；半同步复制对于客户的请求响应稍微慢点，但是他可以保证二进制日志的完整性。

## 二，半同步复制的实现

配置主节点：

1. mysql> install plugin rpl\_semi\_sync\_master soname 'semisync\_master.so'; #安装插件
2. Query OK, 0 rows affected (0.07 sec)
4. mysql> show global variables like '%semi%';
5. +------------------------------------+--------------+
6. | Variable\_name | Value |
7. +------------------------------------+--------------+
8. | rpl\_semi\_sync\_master\_enabled | OFF |
9. | rpl\_semi\_sync\_master\_timeout | 10000 |
10. | rpl\_semi\_sync\_master\_trace\_level | 32 |
11. | rpl\_semi\_sync\_master\_wait\_no\_slave | ON |
12. | rpl\_semi\_sync\_master\_wait\_point | AFTER\_COMMIT |
13. +------------------------------------+--------------+
14. 5 rows in set (0.00 sec)
16. mysql> set global rpl\_semi\_sync\_master\_enabled=on; #启用插件
17. Query OK, 0 rows affected (0.02 sec)
19. mysql> set global rpl\_semi\_sync\_master\_timeout=2000; #设置超时时间
20. Query OK, 0 rows affected (0.00 sec)
22. mysql> show global variables like '%semi%';
23. +------------------------------------+--------------+
24. | Variable\_name | Value |
25. +------------------------------------+--------------+
26. | rpl\_semi\_sync\_master\_enabled | ON |
27. | rpl\_semi\_sync\_master\_timeout | 2000 |
28. | rpl\_semi\_sync\_master\_trace\_level | 32 |
29. | rpl\_semi\_sync\_master\_wait\_no\_slave | ON |
30. | rpl\_semi\_sync\_master\_wait\_point | AFTER\_COMMIT |
31. +------------------------------------+--------------+
32. 5 rows in set (0.00 sec)

rpl\_semi\_sync\_master\_enabled是控制Master是否开启半同步，开启或不开启,将其设置为ON或OFF(1or0).

rpl\_semi\_sync\_master\_timeout是控制Master等待多长时间被告知Slave已收到，也就是所谓的超时时间。

rpl\_semi\_sync\_slave\_enabled是控制Slave是否开启半同步，开启或不开启，将其设置为ON或OFF(1or0)。

监控半同步复制的状态变量（几个常用的）：

Rpl\_semi\_sync\_master\_clients：查看有多少个开启半同步复制的插件的Slave

Rpl\_semi\_sync\_master\_status：查看在Master上半同步复制是否正在运行，其值为ON时，说明Master已启用半同步且已被告知有Slave收到；其值为OFF时，说明Master没启用半同步或是没被告知，由于timeout等原因。

Rpl\_semi\_sync\_master\_no\_tx：查看有多少事务没有用半同步复制的机制进行复制。

Rpl\_semi\_sync\_master\_yes\_tx：查看有多少事务是通过半同步复制机制成功复制。

Rpl\_semi\_sync\_slave\_status：查看Slave上半同步复制是否正常运行，其值为ON时，说明Slave正通过半同步复制且Slave I/O正在运行；为OFF时，反之。

使用相同步骤配置从节点，完成后需要重启io\_thread，不重启当执行时会超时，超时后则自动降为异步：

1. MariaDB [mydb]> install plugin rpl\_semi\_sync\_master soname 'semisync\_master.so';
2. MariaDB [mydb]> set global rpl\_semi\_sync\_master\_enabled=on;
4. MariaDB [mydb]> stop slave io\_thread;
5. Query OK, 0 rows affected (0.01 sec)
7. MariaDB [mydb]> start slave io\_thread;
8. Query OK, 0 rows affected (0.00 sec)
10. mysql> show global status like '%semi%'; #查看半同步客户端
11. +--------------------------------------------+-------+
12. | Variable\_name | Value |
13. +--------------------------------------------+-------+
14. | Rpl\_semi\_sync\_master\_clients | 0 |
15. | Rpl\_semi\_sync\_master\_net\_avg\_wait\_time | 0 |
16. | Rpl\_semi\_sync\_master\_net\_wait\_time | 0 |
17. | Rpl\_semi\_sync\_master\_net\_waits | 0 |
18. | Rpl\_semi\_sync\_master\_no\_times | 1 |
19. | Rpl\_semi\_sync\_master\_no\_tx | 3 |
20. | Rpl\_semi\_sync\_master\_status | OFF |
21. | Rpl\_semi\_sync\_master\_timefunc\_failures | 0 |
22. | Rpl\_semi\_sync\_master\_tx\_avg\_wait\_time | 0 |
23. | Rpl\_semi\_sync\_master\_tx\_wait\_time | 0 |
24. | Rpl\_semi\_sync\_master\_tx\_waits | 0 |
25. | Rpl\_semi\_sync\_master\_wait\_pos\_backtraverse | 0 |
26. | Rpl\_semi\_sync\_master\_wait\_sessions | 0 |
27. | Rpl\_semi\_sync\_master\_yes\_tx | 0 |
28. +--------------------------------------------+-------+
29. 14 rows in set (0.00 sec)

然后自行验证吧！

总结：使用半同步复制机制，性能也许会受到影响，但其主要是为了维持数据完整性，安全性的的一个策略，虽会损失一点性能，但还是值得的。

**请简洁地描述下MySQL中InnoDB支持的四种事务隔离级别名称，以及逐级之间的区别？**

未提交读（uncommited read），提交读（commited read），重复读（repeatable read），串行读（serializable）

这四种隔离级别逐个提高，区别表现在脏读，非重复读，幻读这三点，还有对并发的影响，隔离级别越高，并发性越差；

所谓脏读，就是同一事务中，会读取还未提交的事务修改的数据；

非重复读，是指在同一事务中，在t1时刻，读取某行数据时为A，t2时刻读取同一行数据时，由于其他事务更新，这行数据已经发生改变；

幻读，是指在同一事务中，同一查询多次进行时，由于其他事务的提交，插入新纪录，导致每次查询的结果都不同；

　　区别在于：

未提交读：会造成脏读，非重复读，幻读；

提交读：不会造成脏读，但是会有非重复读，幻读；

重复读：可能会造成幻读；

串行读：不会造成脏读，非重复读，幻读；

5.关系型数据库

**数据库的三范式：**

①字段不可分。

②有主键，非主键字段依赖主键。

③非主键字段不能互相依赖。

**PreparedStatement 相 对 Statement有以下 优 点：**

1.防注入攻击

2.多次运行速度快

3.防止数据库缓冲区溢出

4.代 码 的可 读 性可 维护 性好

这 四点使得 PreparedStatement 成 为访问 数据 库 的 语 句 对 象的首 选 ，缺点是灵活性不 够 好，有些 场 合 还 是必 须 使用 Statement。

1.PreparedStatement是预编译的,对于批量处理可以大大提高效率. 也叫JDBC存储过程

2.使用 Statement 对象。在对数据库只执行一次性存取的时侯，用 Statement 对象进行处理。PreparedStatement 对象的开销比Statement大，对于一次性操作并不会带来额外的好处。

3.statement每次执行sql语句，相关数据库都要执行sql语句的编译，preparedstatement是预编译得,preparedstatement支持批处理

**那么怎么才能做到防止SQL注入攻击呢?**

在上面那段代码中，Statement的对象是用来执行SQL语句的，Statement有一个子类叫做PreparedStatement，可以做到防止SQL注入攻击，接下来我们来看看PreparedStatement有什么特点以及怎么使用：

PreparedStatement是Statement的孩子，不同的是，PreparedStatement使用预编译机制，在创建PreparedStatement对象时就需要将sql语句传入，传入的过程中参数要用?替代，这个过程会导致传入的sql被进行预编译，然后再调用PreparedStatement的setXXX将参数设置上去，由于sql语句已经经过了预编译，再传入特殊值也不会起作用了。

而且PreparedStatement使用了预编译机制，sql语句在执行的过程中效率比Statement要高。

String sql = "select\* from users where username=? and password=?";

        Connection conn = null;

        PreparedStatement state = null;

        ResultSet result;

        conn = JdbcUtil.getConnection();

        System.out.println(sql);

        try {

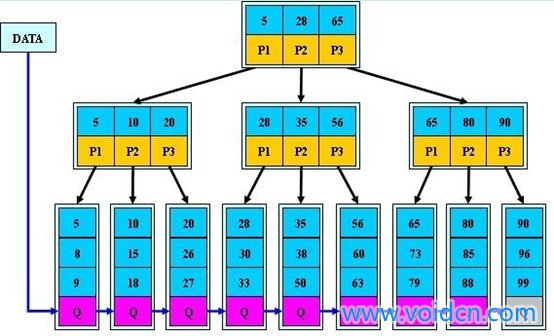
            state = conn.prepareStatement(sql);

            state.setString(1, userName);

            state.setString(2, passWord);

            result = state.executeQuery();

B+树



**为什么说B+-tree比B 树更适合实际应用中操作系统的文件索引和数据库索引？**

**1) B+-tree的磁盘读写代价更低**

B+-tree的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。

举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘快。而B+ 树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B 树就比B+ 树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

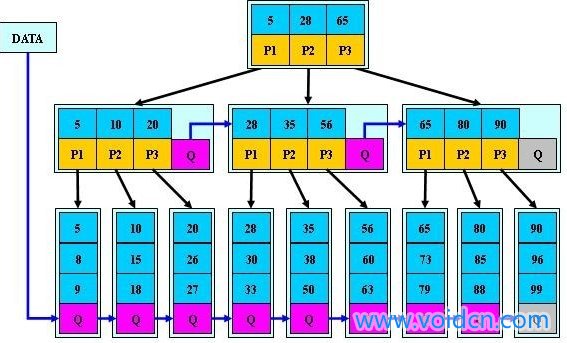
**2) B+-tree的查询效率更加稳定**

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。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。

由于并不是所有节点都具有相同的域，因此B+Tree中叶节点和内节点一般大小不同。这点与B-Tree不同，虽然B-Tree中不同节点存放的key和指针可能数量不一致，但是每个节点的域和上限是一致的，所以在实现中B-Tree往往对每个节点申请同等大小的空间。

B\*-tree

B\*-tree是B+-tree的变体，在B+树的基础上(所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针)，B\*树中非根和非叶子结点再增加指向兄弟的指针；B\*树定义了非叶子结点关键字个数至少为(2/3)\*M，即块的最低使用率为2/3（代替B+树的1/2）。给出了一个简单实例，如下图所示：



B+树的分裂：当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针。

B\*树的分裂：当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针。

**所以，B\*树分配新结点的概率比B+树要低，空间使用率更高；**

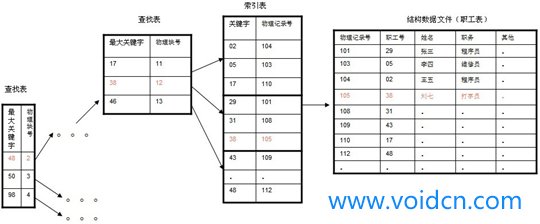
通过以上介绍，大致将B树，B+树，B\*树总结如下：

B树：有序数组+平衡多叉树；

B+树：有序数组链表+平衡多叉树；

B\*树：一棵丰满的B+树。

在大规模数据存储的文件系统中，B~tree系列数据结构，起着很重要的作用，对于存储不同的数据，节点相关的信息也是有所不同，这里根据自己的理解，画的一个查找以职工号为关键字，职工号为38的记录的简单示意图。(这里假设每个物理块容纳3个索引，磁盘的I/O操作的基本单位是块（block),磁盘访问很费时，采用B+树有效的减少了访问磁盘的次数。）



走进搜索引擎的作者梁斌老师针对B树、B+树给出了他的意见（为了真实性，特引用其原话，未作任何改动）： “**B+树还有一个最大的好处，方便扫库，B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持。这是数据库选用B+树的最主要原因。**

Bucket Li："mysql 底层存储是用B+树实现的，知道为什么么。内存中B+树是没有优势的，但是一到磁盘，B+树的威力就出来了"。

**为什么使用B-Tree（B+Tree）？**

上文说过，红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构，这一节将结合计算机组成原理相关知识讨论B-/+Tree作为索引的理论基础。

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的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，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。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h（树高）个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页（一个块block，一般为1k，2k或4k），这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

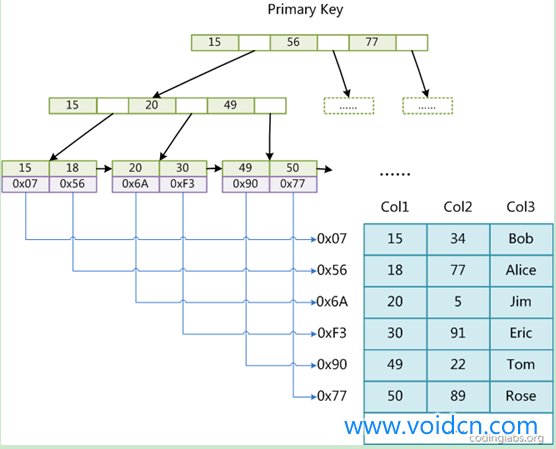
每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

综上所述，用B-Tree作为索引结构效率是非常高的。

**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：



**可以看出MyISAM的索引文件仅仅保存数据记录的地址。**

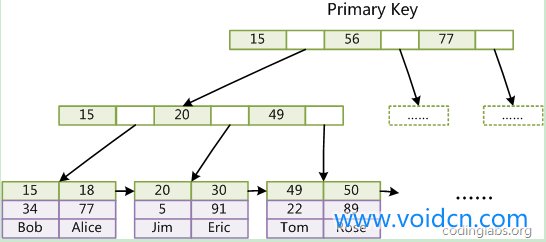
因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。

MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

**InnoDB索引实现**

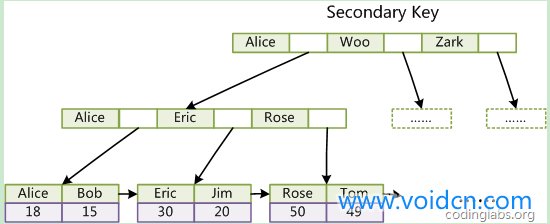
虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

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。**



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，所以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，这个字段长度为6个字节，类型为长整形。

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，下图为定义在Col3上的一个辅助索引：



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。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，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。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗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。

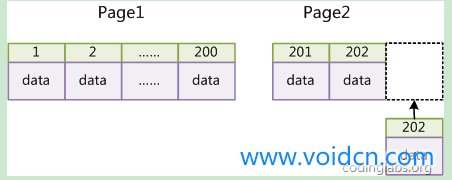
InnoDB的主键选择与插入优化

在使用InnoDB存储引擎时，如果没有特别的需要，请永远使用一个与业务无关的自增字段作为主键。

经常看到有帖子或博客讨论主键选择问题，有人建议使用业务无关的自增主键，有人觉得没有必要，完全可以使用如学号或身份证号这种唯一字段作为主键。不论支持哪种论点，大多数论据都是业务层面的。如果从数据库索引优化角度看，使用InnoDB引擎而不使用自增主键绝对是一个糟糕的主意。

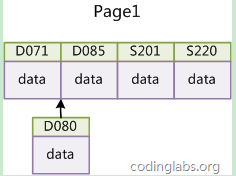
上文讨论过InnoDB的索引实现，InnoDB使用聚集索引，数据记录本身被存于主索引（一颗B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放，因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）。

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：



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。

如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置：



此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

因此，只要可以，请尽量在InnoDB上采用自增字段做主键。

一般怎么建索引

1、充分发挥like的作用

如：select id from t where substring(name,1,3)='abc' ，name以abc开头的id\*\*

应改为:select id from t where name like 'abc%' 这样当name有索引的时候是可以用上索引的，如果改成like '%abc'能索引上么，答案是不能

2、索引字段尽量不要设置为NULL并且进行值的where判断，否则将导致引擎放弃使用索引而进行全表扫描

3、不要在索引字段上使用mysql的函数，如where substr(date,1,10) = '2016-09-07' 这样索引是会失效的，对于这种情况可以改写为 date between '2016-09-07 00:00:00' and '2016-09-07 23:59:59'

4、复合索引建立以后如index\_a\_b\_c建立在a、b、c3个字段上：

where a=XX and b=XX and c=XX能被索引

where a=XX能被索引

where a=XX and b=XX能被索引

where b=XX 不被索引

where c=XX不被索引

where b=XX and c=XX不被索引

where a=XX and c=XX索引较差

where b=XX and c=XX and a= XX不被索引

你可以理解为当顺序不一样时，索引指向就变了。

如果不是这种情况怎么办呢？还能怎么办，修改where顺序啊，总比不同的顺序再建个索引好

5、在= 、group by 和 order by字段上面加上索引

6、在join的时候中结果集更小的部分join更大的部门，这样可以减少缓存的开销

7、索引并不是越多越好不要每一个字段建一个索引，即使这样mysql也会自身优化也只会选择其中的一个索引来执行，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

8、在使用in的时候可以尝试使用exists试试

9、在join的时候减少extra字段中临时表的数量。

**在mysql中的视图概念, 它并不是一个真实存在的表,而是根据自己写的sql语句执行所得的结果集, 方便查询过程和结果比较复杂时候暂存结果以便它用. 使用视图时候, 直接将它作为表来使用即可**

视图创建

-- 创建一个名为vw1的视图, 视图内容为select的语句执行结果

CREATE VIEW vw1 AS

SELECT

userinfo.`name` AS uname, color.tag AS color

FROM

userinfo

LEFT JOIN color ON userinfo.color\_nid = color.nid

视图使用 SELECT \* from vw1;

执行结果

删除视图 drop view vw1

修改视图

-- 修改视图vw1, 修改内容直接写上现今要执行的sql语句即可

ALTER VIEW vw1 AS

SELECT

userinfo.nid,userinfo.`name` AS uname, color.tag AS color

FROM

userinfo

LEFT JOIN color ON userinfo.color\_nid = color.nid

在介绍触发器, 存储过程, 函数以及事务之前我们先简单过一下mysql中的条件和循环语句块

条件判断

if 条件 then

普通sql语句;

elseif 条件 then

普通sql语句;

else

普通sql语句;

end if;

循环语句

while循环

while 条件 do

普通sql语句;

end while;

repeat循环

repeat

普通sql语句;

until 条件;

end repeat;

loop循环

loop\_label: 标签名

普通sql语句;

-- 继续循环

iterate loop\_label;

-- 跳出循环

leave loop\_label;

end loop;

**sql语句应该考虑哪些安全性？**

答：

（1）防止sql注入，对特殊字符进行转义，过滤或者使用预编译的sql语句绑定变量。

（2）最小权限原则，特别是不要用root账户，为不同的类型的动作或者组建使用不同的账户。

（3）当sql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。

**简单描述MySQL中，索引，主键，唯一索引，联合索引的区别，对数据库的性能有什么影响。**

答:

（1）索引是一种特殊的文件（InnoDB数据表上的索引是表空间的一个组成部分），它们包含着对数据表里所有记录的引用指针。

（2）普通索引（由关键字KEY或INDEX定义的索引）的唯一任务是加快对数据的访问速度。

（3）普通索引允许被索引的数据列包含重复的值，如果能确定某个数据列只包含彼此各不相同的值，在为这个数据列创建索引的时候就应该用关键字UNIQE把它定义为一个唯一所以，唯一索引可以保证数据记录的唯一性。

（4）主键，一种特殊的唯一索引，在一张表中只能定义一个主键索引，逐渐用于唯一标识一条记录，是用关键字PRIMARY KEY来创建。

（5）索引可以覆盖多个数据列，如像INDEX索引，这就是联合索引。

（6）索引可以极大的提高数据的查询速度，但是会降低插入删除更新表的速度，因为在执行这些写操作时，还要操作索引文件。

**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 ？**

答：

（1）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。

（2）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，所以重启数据库或者是对表进行OPTIMIZE操作，都会导致最大ID丢失。

**请简述项目中优化sql语句执行效率的方法，从哪些方面。sql语句性能如何分析？**

答：

（1）尽量选择较小的列

（2）将where中用的比较频繁的字段建立索引

（3）select子句中避免使用‘\*’

（4）避免在索引列上使用计算，not，in和<>等操作

（5）当只需要一行数据的时候使用limit 1

（6）保证表单数据不超过200w，适时分割表

（7）**针对查询较慢的语句，可以使用explain来分析该语句具体的执行情况**

**mysql\_fetch\_row()和mysql\_fetch\_array()的区别**

答：这两个函数，**返回的都是一个数组**，区别就是第一个函数返回的数组是只包含值，我们只能row[0],row[1],这样以数组下标来读取数据，而mysql\_fetch\_array()返回的数组既包含第一种，也包含键值对的形式，我们可以这样读取数据，（假如数据库的字段是 username,passwd）:

row[‘username‘],

row[‘passwd‘。

问题1：你如何确定 **[MySQL](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/laoniyouxi123/article/details/_blank)** 是否处于运行状态？   
答案： 在Debian 上运行命令 service mysql status，然后看输出即可。   
在 RedHat 或者 centos 上运行命令 service mysqld status，然后看看输出即可。

Mysql> Show status;

问题2：如何开启或停止 MySQL 服务？   
答案：运行命令 service mysqld start 开启服务；   
运行命令 service mysqld stop 停止服务。

问题3：如何通过 Shell 登入 MySQL？   
答案：运行命令 mysql -u用户名 -p登陆密码

问题4：如何列出所有**[数据库](http://lib.csdn.net/base/mysql" \o "MySQL知识库" \t "http://blog.csdn.net/laoniyouxi123/article/details/_blank)**？   
答案：运行命令 show databases;

问题5： 如何切换到某个数据库并在上面工作？   
答案：（1）运行命令 use database\_name;   
（2）进入名为 database\_name 的数据库。

问题6：如何列出某个数据库内所有表？   
答案：在当前数据库运行命令 show tables;

问题7：如何获取表内所有 Field 对象的名称和类型？   
答案：运行命令 describe 表名;   
简写为desc 表名;

问题8：如何删除表？   
答案：运行命令 drop table 表名;

问题9：如何删除数据库？   
答案：运行命令 drop database 数据库名;

问题10：如何查看表内所有数据？   
答案：运行命令 select \* from 表名;

问题11：如何从表（比如 oc\_users ）中获取一个 field 对象（比如 uid）的所有数据？   
答案：运行命令 select uid from oc\_users;

问题12：假设你有一个名为 ‘xyz’ 的表，它存在多个字段，如 ‘createtime’和 ‘engine’，   
名为 engine 的字段由 ‘Memoty’ 和 ‘MyIsam’ 两种数值组成。   
如何只列出 ‘createtime’ 和 ‘engine’ 这两列，并且 engine 的值为 ‘MyIsam’？   
答案：运行命令 select create\_time, engine from xyz where engine = “MyIsam” ;

问题13：如何列出表 ‘xrt’ 内 name 域值为 ‘tecmint’，web\_address 域值为 ‘tecmint.com’ 的所有数据？   
答案：运行命令 select \* from xrt where name = “tecmint” and web\_address = “tecmint.com” ;

问题14：如何列出表 ‘xrt’ 内 name 域值不为 ‘tecmint’，web\_address 域值为 ‘tecmint.com’ 的所有数据？   
答案：运行命令 select \* from xrt where name != “tecmint” and web\_address = “tecmint.com”;

问题15：如何知道表内行数？   
答案：运行命令 select count(\*) from 表名;

mysql[数据库](http://www.2cto.com/database/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)">1、如何登陆my[sql数据库](http://www.2cto.com/database/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)

mysql -u username -p

## **2、如何开启/关闭mysql服务**

service mysql start/stop

net start/stop mysql

mysqladmin -u root shutdown（在设置了密码的情况下的命令为mysqladmin -u root -p shutdown）；

## **3、查看mysql的状态**

service mysql status

## **4、如何显示数所有数据库**

show databases

## **5、如何获取表内所有字段对象的名称和类型**

describe table\_name;

## **6、MYSQL支持事务吗？**

在缺省模式下，MYSQL是autocommit模式的，所有的数据库更新操作都会即时提交，所以在缺省情况下，mysql是不支持事务的。  
但是如果你的MYSQL表类型是使用InnoDB Tables 或 BDB tables的话，你的MYSQL就可以使用事务处理,使用SET AUTOCOMMIT=0就可以使MYSQL允许在非autocommit模式，在非autocommit模式下，你必须使用COMMIT来提交你的更改，或者用ROLLBACK来回滚你的更改。  
示例如下：  
START TRANSACTION;  
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;  
UPDATE table2 SET summmary=@A WHERE type=1;  
COMMIT;

## **7、MYSQL相比于其他数据库有哪些特点？**

MySQL是一个小型关系型数据库管理[系统](http://www.2cto.com/os/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)，开发者为瑞典MySQL AB公司，现在已经被Sun公司收购，支持FreeBSD、[Linux](http://www.2cto.com/os/linux/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)、MAC、Windows等多种操作系统与其他的大型数据库例如[Oracle](http://www.2cto.com/database/Oracle/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)、[DB2](http://www.2cto.com/database/DB2/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)、SQL Server等相比功能稍弱一些  
1、**可以处理拥有上千万条记录的大型数据**  
2、**支持常见的SQL语句规范**  
3、**可移植行高，安装简单小巧**  
4、良**好的运行效率，有丰富信息的网络支持**  
5、**调试、管理，优化简单**（相对其他大型数据库）

## **8、varchar和char的区别**

Char是一种固定长度的类型，varchar是一种可变长度的类型

## **9、数据库事物有哪几种？**

隔离性、持续性、一致性、原子性

## **10、请简洁地描述下MySQL中InnoDB支持的四种事务隔离级别名称，以及逐级之间的区别？**

SQL标准定义的四个隔离级别为：  
read uncommited：读取未提交内容  
read committed：读取提交内容  
repeatable read：可重读  
serializable：可串行化  
详细解释如下：  
Read Uncommitted（读取未提交内容）  
在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）。  
Read Committed（读取提交内容）  
这是大多数数据库系统的默认隔离级别（但不是MySQL默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的commit，所以同一select可能返回不同结果。  
Repeatable Read（可重读）  
这是MySQL的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读（Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。InnoDB和Falcon存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control 间隙锁）机制解决了该问题。注：其实**多版本只是解决不可重复读问题，而加上间隙锁（也就是它这里所谓的并发控制）才解决了幻读问题**。

**为了解决幻读问题，innodb引入了gap锁。**

在事务A执行：update msg set message=‘订单’ where token=‘asd’;

innodb首先会和RC级别一样，给索引上的记录添加上X锁，此外，还在非唯一索引’asd’与相邻两个索引的区间加上锁。

这样，当事务B在执行insert into msg values (null,‘asd',’hello’); commit;时，会首先检查这个区间是否被锁上，如果被锁上，则不能立即执行，需要等待该gap锁被释放。这样就能避免幻读问题。

Serializable（可串行化）  
这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。  
对于不同的事务，采用不同的隔离级别分别有不同的结果。不同的隔离级别有不同的现象。主要有下面3种现在：  
1、脏读（dirty read）：一个事务可以读取另一个尚未提交事务的修改数据。  
2、非重复读（nonrepeatable read）：在同一个事务中，同一个查询在T1时间读取某一行，在T2时间重新读取这一行时候，这一行的数据已经发生修改，可能被更新了（update），也可能被删除了（delete）。  
3、幻像读（phantom read）：在同一事务中，同一查询多次进行时候，由于其他插入操作（insert）的事务提交，导致每次返回不同的结果集。  
不同的隔离级别有不同的现象，并有不同的锁定/并发机制，隔离级别越高，数据库的并发性就越差，4种事务隔离级别分别表现的现象如下表：

## **11、mysql数据库引擎MyISAM和InnoDB的区别**

## **12、mysql有关权限的表都有哪几个**

MySQL服务器通过权限表来控制用户对数据库的访问，权限表存放在mysql数据库里，由mysql\_install\_db脚本初始化。这些权限表分别user，db，table\_priv，columns\_priv和host。下面分别介绍一下这些表的结构和内容：  
user权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。  
db权限表：记录各个帐号在各个数据库上的操作权限。  
table\_priv权限表：记录数据表级的操作权限。  
columns\_priv权限表：记录数据列级的操作权限。  
host权限表：配合db权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受GRANT和REVOKE语句的影响。

## **13、mysql存储引擎有哪些？如何修改mysql存储引擎？**

MyISAM indexed sequential access method (有索引的顺序访问方法)  
**MyISAM** 具有检查和修复表格的大多数工具。表格可以被压缩，而且支持全文收索  
不是事务安全的，而且不支持外键。  
**MEMORY** 也是以前的(HEAP) 该类型表存储在内存中，表的索引是哈希分布的。  
**merge** 这些表为了查询目的，把myisam 表集合作为单个表，因此你可以在某些操作系统中避开最大文件大小的限制。  
**archive** 这种类型的表只支持，insert ,select 不支持delete,update,replace ,不使用索引。  
**csv** 这些表保存在服务器的单个文件中，它包含了**用逗号间隔的数据。**

**innodb** 这种表是事务安全的。提供了commit（提交） rollback（实务回滚）支持外键，比myisam慢。  
修改mysql存储引擎alter table tablename type = innodb;

## **14、MYSQL 数据表修复及数据恢复面试题**

MYSQL数据表在什么情况下容易损坏？  
**服务器突然断电导致数据文件损坏。**  
强制关机，没有先关闭mysql 服务等。 数据表损坏后的主要现象是什么？  
从表中选择数据之时，得到如下错误：Incorrect key file for table: ‘…’. Try to repair it  
查询不能在表中找到行或返回不完全的数据。  
Error: Table ‘p’ is marked as crashed and should be repaired 。  
打开表失败： Can’t open file: ‘×××.MYI’ (errno: 145) 。 数据表损坏的修复方式有哪些？  
1. 使用 **myisamchk** 来修复，具体步骤：  
1）修复前将mysql服务停止。  
2）打开命令行方式，然后进入到mysql的/bin目录。  
3）执行**myisamchk –recover 数据库所在路径/\*.MYI  
2.** 使用**repair table 或者 OPTIMIZE table**命令来修复，**REPAIR TABLE table\_name 修复表** **OPTIMIZE TABLE table\_name 优化表 REPAIR TABLE 用于修复被破坏的表。**  
**OPTIMIZE TABLE 用于回收闲置的数据库空间，当表上的数据行被删除时**，所占据的磁盘空间并没有立即被回收，使用了OPTIMIZE TABLE命令后这些空间将被回收，并且**对磁盘上的数据行进行重排**（注意：是磁盘上，而非数据库）

## **15、MYSQL数据库服务器性能分析的方法命令有哪些?**

Show status  
一些值得监控的变量值：  
Bytes\_received和Bytes\_sent  
和服务器之间来往的流量。  
Com\_\*服务器正在执行的命令。  
Created\_\*在查询执行期限间创建的临时表和文件。  
Handler\_\*存储引擎操作。  
Select\_\*不同类型的联接执行计划。  
Sort\_\*几种排序信息。  
Show session status like ‘Select’;  
Show profiles  
SET profiling=1;  
Show profiles\G  
Show profile;

## **16、 mysql里记录货币用什么字段类型好**

**NUMERIC和DECIMAL**类型被MySQL实现为同样的类型，这在SQL92标准允许。他们被用于保存值，该值的准确精度是极其重要的值，例如与金钱有关的数据。当声明一个类是这些类型之一时，精度和规模的能被(并且通常是)指定；例如：  
salary DECIMAL(9,2)  
在这个例子中，9(precision)代表将被用于存储值的总的小数位数，而2(scale)代表将被用于存储小数点后的位数。因此，在这种情况下，能被存储在salary列中的值的范围是从-9999999.99到9999999.99。在ANSI/ISO SQL92中，句法DECIMAL(p)等价于DECIMAL(p,0)。同样，句法DECIMAL等价于DECIMAL(p,0)，这里实现被允许决定值p。MySQL当前不支持DECIMAL/NUMERIC数据类型的这些变种形式的任一种。这一般说来不是一个严重的问题，因为这些类型的主要益处得自于明显地控制精度和规模的能力。  
**DECIMAL和NUMERIC值作为字符串存储，而不是作为二进制浮点数，以便保存那些值的小数精度**。一个字符用于值的每一位、小数点(如果scale>0)和“-”符号(对于负值)。如果scale是0，DECIMAL和NUMERIC值不包含小数点或小数部分。  
DECIMAL和NUMERIC值得最大的范围与DOUBLE一样，但是对于一个给定的DECIMAL或NUMERIC列，实际的范围可由制由给定列的precision或scale限制。当这样的列赋给了小数点后面的位超过指定scale所允许的位的值，该值根据scale四舍五入。当一个DECIMAL或NUMERIC列被赋给了其大小超过指定(或缺省的）precision和scale隐含的范围的值，MySQL存储表示那个范围的相应的端点值。

# **2.数据库事务的四个特性及含义**

数据库事务transanction正确执行的四个基本要素。ACID,原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)、持久性(Durability)。  
**原子性**:整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。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**:在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。  
**隔离性**:隔离状态执行事务，使它们好像是[系统](http://www.2cto.com/os/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行 相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为串行化，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。  
**持久性**:在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

# **3.视图的作用，视图可以更改么？**

视图是虚拟的表，与包含数据的表不一样，视图只包含使用时动态检索数据的查询；不包含任何列或数据。使用视图可以简化复杂的sql操作，隐藏具体的细节，保护数据；视图创建后，可以使用与表相同的方式利用它们。  
**视图不能被索引，也不能有关联的触发器或默认值，**如果视图本身内有order by 则对视图再次order by将被覆盖。  
创建视图：create view XXX as XXXXXXXXXXXXXX;  
对于**某些视图比如未使用联结子查询分组聚集函数Distinct Union等，是可以对其更新的**，**对视图的更新将对基表进行更新；**但**是视图主要用于简化检索，保护数据，并不用于更新，而且大部分视图都不可以更新。**

# **4.drop,delete与truncate的区别**

drop直接删掉表 truncate删除表中数据，再插入时自增长id又从1开始 delete删除表中数据，可以加where字句。

（1） DELETE语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行进行回滚操作。TRUNCATE TABLE 则一次性地从表中删除所有的数据并不把单独的删除操作记录记入日志保存，删除行是不能恢复的。并且在删除的过程中不会激活与表有关的删除触发器。执行速度快。

（2） 表和索引所占空间。当表被TRUNCATE 后，这个表和索引所占用的空间会恢复到初始大小，而DELETE操作不会减少表或索引所占用的空间。drop语句将表所占用的空间全释放掉。

（3） 一般而言，drop > truncate > delete

（4） 应用范围。TRUNCATE 只能对TABLE；DELETE可以是table和view

（5） TRUNCATE 和DELETE只删除数据，而DROP则删除整个表（结构和数据）。

（6） truncate与不带where的delete ：只删除数据，而不删除表的结构（定义）drop语句将删除表的结构被依赖的约束（constrain),触发器（trigger)索引（index);依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。

（7） delete语句为**DML（data maintain Language),**这个操作会被放到 rollback segment中,事务提交后才生效。如果有相应的 tigger,执行的时候将被触发。

（8） truncate、drop是**DDL（data define language),**操作立即生效，原数据不放到 rollback segment中，不能回滚

（9） 在没有备份情况下，谨慎使用 drop 与 truncate。要删除部分数据行采用delete且注意结合where来约束影响范围。回滚段要足够大。要删除表用drop;若想保留表而将表中数据删除，如果于事务无关，用truncate即可实现。如果和事务有关，或老师想触发trigger,还是用delete。

（10） Truncate table 表名 速度快,而且效率高,因为:  
truncate table 在功能上与不带 WHERE 子句的 DELETE 语句相同：二者均删除表中的全部行。但 TRUNCATE TABLE 比 DELETE 速度快，且使用的系统和事务日志资源少。DELETE 语句每次删除一行，并在事务日志中为所删除的每行记录一项。TRUNCATE TABLE 通过释放存储表数据所用的数据页来删除数据，并且只在事务日志中记录页的释放。

（11） TRUNCATE TABLE 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用 DELETE。如果要删除表定义及其数据，请使用 DROP TABLE 语句。

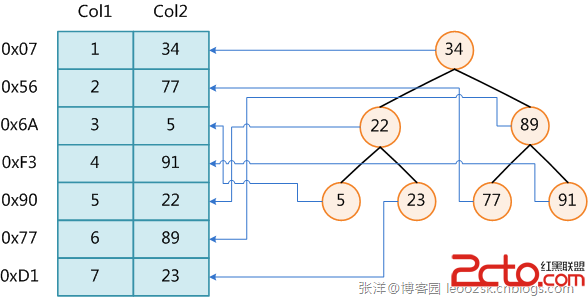
（12） 对于由 FOREIGN KEY 约束引用的表，不能使用 TRUNCATE TABLE，而应使用不带 WHERE 子句的 DELETE 语句。由于 TRUNCATE TABLE 不记录在日志中，所以它不能激活触发器。

# **5.索引的工作原理及其种类**

**数据库索引**，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。**索引的实现通常使用B树及其变种B+树**。

在数据之外，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，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。

**为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。**



图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快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O(log2n)的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，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，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。**一般来说，应该在这些列上创建索引：**在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。**一般来说，不应该创建索引的的这些列具有下列特点：**

**第一，对于那些在查询中很少使用或者参考的列不应该创建索引。**这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

**第二，对于那些只有很少数据值的列也不应该增加索引**。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

**第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。**这是因为，这些列的数据量要么相当大，要么取值很少。

**第四，当修改性能远远大于检索性能时，不应该创建索引**。这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在[数据库设计](http://www.2cto.com/database/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)器中创建三种索引：**唯一索引、主键索引和聚集索引**。

**唯一索引**

唯一索引是不允许其中任何两行具有相同索引值的索引。

当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。例如，如果在employee表中职员的姓(lname)上创建了唯一索引，则任何两个员工都不能同姓。 **主键索引** 数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。 在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。 **聚集索引** 在聚集索引中，**表中行的物理顺序与键值的逻辑（索引）顺序相同**。**一个表只能包含一个聚集索引。**

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。**与非聚集索引相比，聚集索引通常提供更快的数据访问速度。**

### **局部性原理与磁盘预读**

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的**局部性原理**：**当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。**

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，**预读可以提高I/O效率。**

**预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块**，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，**页得大小通常为4k**），主存和磁盘以页为单位交换数据。**当程序要读取的数据不在主存中时，会触发一个缺页异常**，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

### **B-/+Tree索引的性能分析**

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，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。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

**B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。**一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

**而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性**，**所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。**

**综上所述，用B-Tree作为索引结构效率是非常高的。**

# **6.连接的种类**

查询分析器中执行：  
--建表table1,table2：  
create table table1(id int,name varchar(10))  
create table table2(id int,score int)  
insert into table1 select 1,'lee'  
insert into table1 select 2,'zhang'  
insert into table1 select 4,'wang'  
insert into table2 select 1,90  
insert into table2 select 2,100  
insert into table2 select 3,70  
如表  
-------------------------------------------------  
table1 | table2 |  
-------------------------------------------------  
id name |id score |  
1 lee |1 90|  
2 zhang| 2 100|  
4 wang| 3 70|  
-------------------------------------------------  
  
以下均在查询分析器中执行  
一、外连接  
1.概念：包括左向外联接、右向外联接或完整外部联接  
  
2.左连接：left join 或 left outer join  
(1)左向外联接的结果集包括 LEFT OUTER 子句中指定的左表的所有行，而不仅仅是联接列所匹配的行。如果左表的某行在右表中没有匹配行，则在相关联的结果集行中右表的所有选择列表列均为空值(null)。  
(2)sql 语句  
select \* from table1 left join table2 on table1.id=table2.id  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang2100  
4wangNULLNULL  
------------------------------  
注释：包含table1的所有子句，根据指定条件返回table2相应的字段，不符合的以null显示  
  
3.右连接：right join 或 right outer join  
(1)右向外联接是左向外联接的反向联接。将返回右表的所有行。如果右表的某行在左表中没有匹配行，则将为左表返回空值。  
(2)sql 语句  
select \* from table1 right join table2 on table1.id=table2.id  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang2100  
NULLNULL370  
------------------------------  
注释：包含table2的所有子句，根据指定条件返回table1相应的字段，不符合的以null显示  
  
4.完整外部联接:full join 或 full outer join  
(1)完整外部联接返回左表和右表中的所有行。当某行在另一个表中没有匹配行时，则另一个表的选择列表列包含空值。**如果表之间有匹配行，则整个结果集行包含基表的数据值。**  
(2)sql 语句  
select \* from table1 **full join** table2 on table1.id=table2.id  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang2100  
4wangNULLNULL  
NULLNULL370  
------------------------------  
**注释：返回左右连接的和（见上左、右连接）**  
  
二、内连接  
1.概念：内联接是用比较运算符比较要联接列的值的联接  
  
2.内连接：join 或 inner join  
  
3.sql 语句  
select \* from table1 join table2 on table1.id=table2.id  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang2100  
------------------------------  
注释：**只返回符合条件的table1和table2的列**  
  
4.等价（与下列执行效果相同）  
A:select a.\*,b.\* from table1 a,table2 b where a.id=b.id  
B:select \* from table1 cross join table2 where table1.id=table2.id (**注：cross join后加条件只能用where,不能用on**)  
  
三、交叉连接**(完全)**  
  
1.概念：没有 WHERE 子句的交叉联接将产生联接所涉及的**表的笛卡尔积**。第一个表的行数乘以第二个表的行数等于笛卡尔积结果集的大小。（table1和table2交叉连接产生3\*3=9条记录）  
  
2.交叉连接：**cross join (不带条件where...)**  
  
3.sql语句  
select \* from table1 cross join table2  
-------------结果-------------  
idnameidscore  
------------------------------  
1lee190  
2zhang190  
4wang190  
1lee2100  
2zhang2100  
4wang2100  
1lee370  
2zhang370  
4wang370  
------------------------------  
注释：返回3\*3=9条记录，即笛卡尔积  
  
4.等价（与下列执行效果相同）  
A:select \* from table1,table2

# **7.数据库范式**

**1 第一范式（1NF）**  
  
在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。  
**所谓第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项**，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，**第一范式就是无重复的列。**

**2 第二范式（2NF）**  
  
第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。**第二范式（2NF）要求数据库表中的每个实例或行必须可以被惟一地区分**。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为**主关键字或主键、主码**。  
第二范式（2NF）要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。简而言之，**第二范式就是非主属性非部分依赖于主关键字。**

**3 第三范式（3NF）**  
  
满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，**第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。**例如，存在一个部门信息表，其中每个部门有部门编号（dept\_id）、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余。简而言之，**第三范式就是属性不依赖于其它非主属性。（我的理解是消除冗余）**

# **8.数据库优化的思路**

这个我借鉴了慕课上关于数据库优化的课程。

## **1.SQL语句优化**

1）应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。  
2）应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：  
select id from t where num is null  
**可以在num上设置默认值0，确保表中num列没有null值**，然后这样查询：  
select id from t where num=0  
3）很多时候用 exists 代替 in 是一个好的选择  
4）用Where子句替换HAVING 子句 因为HAVING 只会在检索出所有记录之后才对结果集进行过滤

## **2.索引优化**

看上文索引

## **3.数据库结构优化**

1）范式优化： 比如消除冗余（节省空间。。） 2）反范式优化：比如适当加冗余等（减少join） 3）拆分表： 分区将数据在物理上分隔开，不同分区的数据可以制定保存在处于不同磁盘上的数据文件里。这样，当对这个表进行查询时，只需要在表分区中进行扫描，而不必进行全表扫描，明显缩短了查询时间，另外处于不同磁盘的分区也将对这个表的数据传输分散在不同的磁盘I/O，一个精心设置的分区可以将数据传输对磁盘I/O竞争均匀地分散开。对数据量大的时时表可采取此方法。可按月自动建表分区。  
4）拆分其实又分垂直拆分和水平拆分： 案例： 简单购物系统暂设涉及如下表： 1.产品表（数据量10w，稳定） 2.订单表（数据量200w，且有增长趋势） 3.用户表 （数据量100w，且有增长趋势） 以[mysql](http://www.2cto.com/database/MySQL/" \t "http://www.cnblogs.com/wangdaijun/p/_blank)为例讲述下水平拆分和垂直拆分，mysql能容忍的数量级在百万静态数据可以到千万 **垂直拆分：**解决问题：表与表之间的io竞争 不解决问题：单表中数据量增长出现的压力 方案： 把产品表和用户表放到一个server上 订单表单独放到一个server上 **水平拆分：** 解决问题：单表中数据量增长出现的压力 不解决问题：表与表之间的io争夺  
方案： 用户表通过性别拆分为男用户表和女用户表 订单表通过已完成和完成中拆分为已完成订单和未完成订单 产品表 未完成订单放一个server上 已完成订单表盒男用户表放一个server上 女用户表放一个server上(女的爱购物 哈哈)

## **4.服务器硬件优化**

这个么多花钱咯！

# **9.存储过程与触发器的区别**

触发器与存储过程非常相似，触发器也是SQL语句集，**两者唯一的区别是触发器不能用EXECUTE语句调用，而是在用户执行Transact-SQL语句时自动触发（激活）执行。触发器是在一个修改了指定表中的数据时执行的存储过程。**通**常通过创建触发器来强制实现不同表中的逻辑相关数据的引用完整性和一致性。**由于用户不能绕过触发器，所以可以用它来强制实施复杂的业务规则，以确保数据的完整性。触发器不同于存储过程，**触发器主要是通过事件执行触发而被执行的**，而**存储过程可以通过存储过程名称名字而直接调用**。当对某一表进行诸如UPDATE、INSERT、DELETE这些操作时，SQLSERVER就会自动执行触发器所定义的SQL语句，从而确保对数据的处理必须符合这些SQL语句所定义的规则。

**只能对永久表创建触发器，不能在临时表中创建触发器；同一个表不能存在两个相同类型的触发器**，例如不能存在两个insert触发器，对应update触发器可以通过IF 不同的字段执行不一样的操作。

**触发器不允许返回结果**

1.根据部门号从高到低，工资从低到高列出员工的信息

select \* from employee order by dept\_id desc,salary

2.union和union all的区别

**用union连接表后，重复的记录会被删除掉,UNION 在进行表链接后会筛选掉重复的记录**，所以在表链接后会对所产生的结果集进行排序运算，删除重复的记录再返回结果。

**用union all连接表后，记录会被完整保留 ,而UNION ALL 只是简单的将两个结果合并后就返回。**这样，如果返回的两个结果集中有重复的数据，那么返回的结果集就会包含重复的数据了。从效率上说，UNION ALL 要比UNION 快很多，所以，如果可以确认合并的两个结果集中不包含重复的数据的话，那么就使用UNION ALL，

3.分页语句（用sql语句直接分页，效率高）

pageSize = 20;

pageNo = 5;

select \* from tbname limit (pageNo-1)\*pageSize,pageSize

7.分页语句

**取出sql 表中第31 到40 的记录（以自动增长ID 为主键）**

sql server 方案1：

select top 10 \* from t where id not in (select top 30 id from t order by id ) orde by id

sql server 方案2：

select top 10 \* from t where id in (select top 40 id from t order by id) order by id desc

**mysql 方案：select \* from t order by id limit 30,10**

oracle 方案：select \* from (select rownum r,\* from t where r<=40) where r>30

4.数据库三范式

第一范式 字段具有原子性，不可再分

第二范式 表中的每列都和主键相关，通俗理解是任意一个字段都只依赖表中的同一个字段。

第三范式 每列都和主键列直接相关，而不是间接相关，简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主键字段。就是说，表的信息，如果能够被推导出来，就不应该单独的设计一个字段来存放(能尽量外键join就用外键join)。很多时候，我们为了满足第三范式往往会把一张表分成多张表。

即满足第二范式前提，**如果某一属性依赖于其他非主键属性，而其他非主键属性又依赖于主键，那么这个属性就是间接依赖于主键，这被称作传递依赖于主属性**。 通俗解释就是一张表最多只存两层同类型信息。

5.用一条SQL语句查询出每门课都大于80分的学生姓名

name kecheng fenshu

张三 语文 81

张三 数学 75

李四 语文 76

李四 数学 90

王五 语文 81

王五 数学 100

王五 英语 90

SELECT name,`subject`,score,sum(score>80) as jg from result GROUP BY `name` HAVING jg=(SELECT COUNT(DISTINCT rs.subject) from result rs );

另一种写法，高大人写的

SELECT

ttt.name

FROM

(select count(r.name) c, r.name from result r WHERE r.score >80 GROUP BY r.`name`) ttt

WHERE

ttt.c=(select count(DISTINCT `subject`) from result)

6.

自动编号 学号 姓名 课程编号 课程名称 分数

1 2005001 张三 0001 数学 69

2 2005002 李四 0001 数学 89

3 2005001 张三 0001 数学 69

删除除了自动编号不同,其他都相同的学生冗余信息

//如下语句，mysql报告错误，可能删除依赖后面统计语句，而删除又导致统计语句结果不一致。

delete from student2 where id not in(select min(id) from

student2 group by name);

//但是，如下语句没有问题：

select \* from student2 where id not in(select min(id) from

student2 group by name);

//于是，我想先把分组的结果做成虚表，然后从虚表中选出结果，最后再将结果作为删除的条件数据。

delete from student2 where id not in(select mid from (select

min(id) mid

from student2 group by name) as t);

7.一个叫department的表，里面只有一个字段name,一共有4条纪录，分别是a,b,c,d,对应四个球对，现在四个球对进行比赛，用一条sql语句显示所有可能的比赛组合.

SELECT \* FROM

department INNER JOIN department as a

ON

department.`name` <> (a.`name`)

8.列出各个部门中工资高于本部门的平均工资的员工数和部门号，并按部门号排序

SELECT name,deptid,salary from employee where salary>(select avg(salary) from employee) GROUP BY deptid asc

23、JDBC 中的PreparedStatement 相比Statement 的好处

答：一个sql 命令发给服务器去执行的步骤为：语法检查，语义分析，编译成内部指令，缓存指令，执行指令等过程。

select \* from student where id =3----缓存--?xxxxx 二进制命令

select \* from student where id =3----直接取-?xxxxx 二进制命令

select \* from student where id =4--- -?会怎么干？

如果当初是select \* from student where id =?--- -?又会怎么干？

上面说的是性能提高

可以防止sql 注入。

22、**用JDBC 如何调用存储过程**

代码如下：

package com.huawei.interview.lym;

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Types;

public class JdbcTest {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

// TODO Auto-generated method stub

Connection cn = null;

**CallableStatement cstmt = null;**

try {

//这里最好不要这么干，因为驱动名写死在程序中了

Class.forName("com.mysql.jdbc.Driver");

//实际项目中，这里应用DataSource 数据，如果用框架，

//这个数据源不需要我们编码创建，我们只需Datasource ds = context.lookup()

//cn = ds.getConnection();

cn = DriverManager.getConnection("jdbc:mysql:///test","root","root");

cstmt = **cn.prepareCall**("{call insert\_Student(?,?,?)}");

**cstmt.registerOutParameter(3,Types.INTEGER);**

cstmt.setString(1, "wangwu");

cstmt.setInt(2, 25);

cstmt.execute();

//get 第几个，不同的数据库不一样，建议不写

System.out.println(cstmt.getString(3));

} catch (Exception e) {

// TODO Auto-generated catch block

e.printStackTrace();

}f

inally

{

/\*try{cstmt.close();}catch(Exception e){}

try{cn.close();}catch(Exception e){}\*/

try {

if(cstmt != null)

cstmt.close();

if(cn != null)

cn.close();

} catch (SQLException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

25、Class.forName 的作用?为什么要用?

答：按参数中指定的字符串形式的类名去搜索并加载相应的类，如果该类字节码已经被加载过，**则返回代表该字节码的Class 实例对象，否则，按类加载器的委托机制去搜索和加载该类，**如果所有的类加载器都无法加载到该类，则抛出ClassNotFoundException。**加载完这个Class 字节码后，接着就可以使用Class 字节码的newInstance 方法去创建该类的实例对象了。**

**有时候，我们程序中所有使用的具体类名在设计时（即开发时）无法确定，只有程序运行时才能确定，这时候就需要使用Class.forName 去动态加载该类**，这个类名通常是在配置文件中配置的，例如，spring 的ioc 中每次依赖注入的具体类就是这样配置的，jdbc 的驱动类名通常也是通过配置文件来配置的，以便在产品交付使用后不用修改源程序就可以更换驱动类名。

**27、用JDBC 查询学生成绩单, 把主要代码写出来（考试概率极大）.**

Connection cn = null;

PreparedStatement pstmt =null;

Resultset rs = null;

try

{

Class.forname(driveClassName);

cn = DriverManager.getConnection(url,username,password);

pstmt = cn.prepareStatement(“select score.\* from score ,student “ +

“where score.stuId = student.id and student.name = ?”);

pstmt.setString(1,studentName);

Resultset rs = pstmt.executeQuery();

while(rs.next())

{

system.out.println(rs.getInt(“subject”) + “ ” + rs.getFloat(“score”) );

}

}catch(Exception e){e.printStackTrace();}

finally

{

if(rs != null) try{ rs.close() }catch(exception e){}

if(pstmt != null) try{pstmt.close()}catch(exception e){}

if(cn != null) try{ cn.close() }catch(exception e){}

}

**28、这段代码有什么不足之处?**

try {

Connection conn = ...;

Statement stmt = ...;

ResultSet rs = stmt.executeQuery("select \* from table1");

while(rs.next()) {

}}

catch(Exception ex) {

}

答：没有finally 语句来关闭各个对象，另外，使用finally 之后，要把变量的定义放在try 语句块的外面，以便在try 语句块之外的finally 块中仍可以访问这些变量。

**29、说出数据连接池的工作机制是什么?**

J2EE 服务器启动时会建立一定数量的池连接，并一直维持不少于此数目的池连接。客户端程序需要连接时，池驱动程序会返回一个未使用的池连接并将其标记为忙。如果当前没有空闲连接，池驱动程序就新建一定数量的连接，新建连接的数量有配置参数决定。当使用的池连接调用完成后，池驱动程序将此连接标记为空闲，其他调用就可以使用这个连接。实现方式，返回的Connection 是原始Connection 的代理，代理Connection 的close 方法不是真正关连接，而是把它代理的Connection 对象还回到连接池中。

**30、为什么要用ORM? 和JDBC 有何不一样?**

orm 是一种思想，就是把object 转变成数据库中的记录，或者把数据库中的记录转变成objecdt，我们可以用jdbc 来实现这种思想，其实，如果我们的项目是严格按照oop 方式编写的话，我们的jdbc 程序不管是有意还是无意，就已经在实现orm 的工作了。现在有许多orm 工具，它们底层调用jdbc 来实现了orm 工作，我们直接使用这些工具，就省去了直接使用jdbc 的繁琐细节，提高了开发效率，现在用的较多的orm 工具是hibernate。也听说一些其他orm 工具，如toplink,ojb 等。

**MySQL常用引擎的锁机制**

**MySQL引擎默认的锁级别：**

**MyISAM和MEMORY采用表级锁(table-level locking)。**

**BDB采用页面锁(page-level locking)或表级锁，默认为页面锁。**

**InnoDB支持行级锁(row-level locking)和表级锁，默认为行级锁。**

1 **行锁以及死锁的产生**

innodb中.索引分主键索引和非主键索引2种.

如果一条sql语句操作了主键索引，MySQL就会锁定这条主键索引；

如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引;

当两个事务同时执行，一个锁住了主键索引，在等待其他相关索引。另一个锁定了非主键索引，在等待主键索引。这样就会发生死锁。

参考:MySQL死锁分析

2 **表锁 分为共享读锁(又称读锁)和排他锁**

读锁(share lock)特点:用户可以并发读取数据,但任何事物都不能对数据进行修改,除非释放了所有共享锁

使用：**SELECT ... LOCK IN SHARE MODE;**

排他锁（exclusive Lock）又称写锁，如果事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的锁。获准排他锁的事务既能读数据，又能修改数据。

特点:如果事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的锁。

使用: **select ... FOR UPDATE;**

3、**意向锁**

**意向共享锁（IS）**：表示事务准备给数据行加入共享锁，**也就是说一个数据行加共享锁前必须先取得该表的IS锁**

**意向排他锁（IX）**：类似上面，表示事务准备给数据行加入排他锁，说明**事务在一个数据行加排他锁前必须先取得该表的IX锁。**

**意向锁是InnoDB自动加的，不需要用户干预.**

总结:

对于**insert、update、delete，InnoDB会自动给涉及的数据加排他锁；**

**对于一般的Select语句，InnoDB不会加任何锁。**

**在Innodb引擎中既支持行锁也支持表锁，那么什么时候会锁住整张表，什么时候或只锁住一行呢？**

**InnoDB行锁是通过给索引上的索引项加锁来实现的，**这一点MySQL与Oracle不同，Oracle者是通过在数据块中对相应数据行加锁来实现的。

InnoDB这种行锁实现特点意味着：**只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！**

在实际应用中，要特别注意InnoDB行锁的这一特性，不然的话，可能导致大量的锁冲突，从而影响并发性能。

**行级锁都是基于索引的，如果一条SQL语句用不到索引是不会使用行级锁的，会使用表级锁。**

**行级锁的缺点是：如果并发，请求大量的锁资源，所以速度慢，内存消耗大。**

**1、行级锁与死锁**

**MyISAM中是不会产生死锁的，因为MyISAM总是一次性获得所需的全部锁，要么全部满足，要么全部等待。**

**在InnoDB中，锁是逐步获得的，就造成了死锁的可能。**

在MySQL中，**行级锁并不是直接锁记录，而是锁索引。**

索引分为主键索引和非主键索引两种：

如果一条sql语句操作了主键索引，MySQL就会锁定这条主键索引；

如果一条语句操作了非主键索引，MySQL会先锁定该非主键索引，再锁定相关的主键索引。

在UPDATE、DELETE操作时，MySQL不仅锁定WHERE条件扫描过的所有索引记录，而且会锁定相邻的键值，即所谓的next-key locking。

**当两个事务同时执行，一个锁住了主键索引，在等待其他相关索引。另一个锁定了非主键索引，在等待主键索引。这样就会发生死锁。**

**死锁一般是事务相互等待对方资源，最后形成环路造成的。**

发生死锁后，InnoDB一般都可以检测到，并使一个事务释放锁回退，另一个获取锁完成事务。

**有多种方法可以避免死锁，我们来介绍常见的三种**

1. 如果不同程序会并发存取多个表，**尽量约定以相同的顺序访问表，可以大大降低死锁机会。**

2. **在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁产生概率**；

3. **对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；**

2、表锁

**表锁分为表共享读锁（共享锁）与表独占写锁（排他锁）**

共享锁(Share Lock)又称读锁，是读取操作创建的锁。其他用户可以并发读取数据，但任何事务都不能对数据进行修改（获取数据上的排他锁），直到已释放所有共享锁。

**如果事务T对数据A加上共享锁后，则其他事务只能对A再加共享锁，不能加排他锁。**获准共享锁的事务只能读数据，不能修改数据。

使用：SELECT ... LOCK IN SHARE MODE;

在查询语句后面增加LOCK IN SHARE MODE， Mysql会对查询结果中的每行都加共享锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请共享锁，否则会被阻塞。其他线程也可以读取使用了共享锁的表，而且这些线程读取的是同一个版本的数据。

排他锁（exclusive Lock）又称写锁，**如果事务T对数据A加上排他锁后，则其他事务不能再对A加任任何类型的锁。**获准排他锁的事务既能读数据，又能修改数据。

使用：SELECT ... FOR UPDATE;

在查询语句后面增加FOR UPDATE， Mysql会对查询结果中的每行都加排他锁，当没有其他线程对查询结果集中的任何一行使用排他锁时，可以成功申请排他锁，否则会被阻塞。

**3、意向锁**

**意向共享锁（IS）：**表示事务准备给数据行加入共享锁，也就是说一个数据行加共享锁前必须先取得该表的IS锁

**意向排他锁（IX）：**类似上面，表示事务准备给数据行加入排他锁，说明事务在一个数据行加排他锁前必须先取得该表的IX锁。

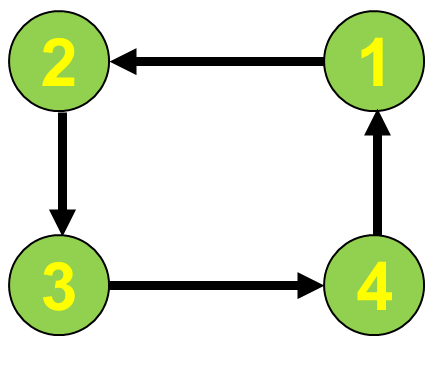
**意向锁是InnoDB自动加的，不需要用户干预。**

**问题来了，innodb是怎么探知死锁的？**

直观方法是在两个事务相互等待时，**当一个等待时间超过设置的某一阀值时，对其中一个事务进行回滚，另一个事务就能继续执行。**这种方法简单有效，**在innodb中，参数innodb\_lock\_wait\_timeout用来设置超时时间。**

仅用上述方法来检测死锁太过被动，innodb还提供了**wait-for graph算法**来主动进行死锁检测，**每当加锁请求无法立即满足需要并进入等待时，wait-for graph算法都会被触发。**

他们相互等待对方的资源，而且形成环路！我们将每辆车看为一个节点，当节点1需要等待节点2的资源时，就生成一条有向边指向节点2，最后形成一个有向图。**我们只要检测这个有向图是否出现环路即可，出现环路就是死锁！这就是wait-for graph算法。**



**innodb将各个事务看为一个个节点，资源就是各个事务占用的锁，当事务1需要等待事务2的锁时，就生成一条有向边从1指向2，最后行成一个有向图。**

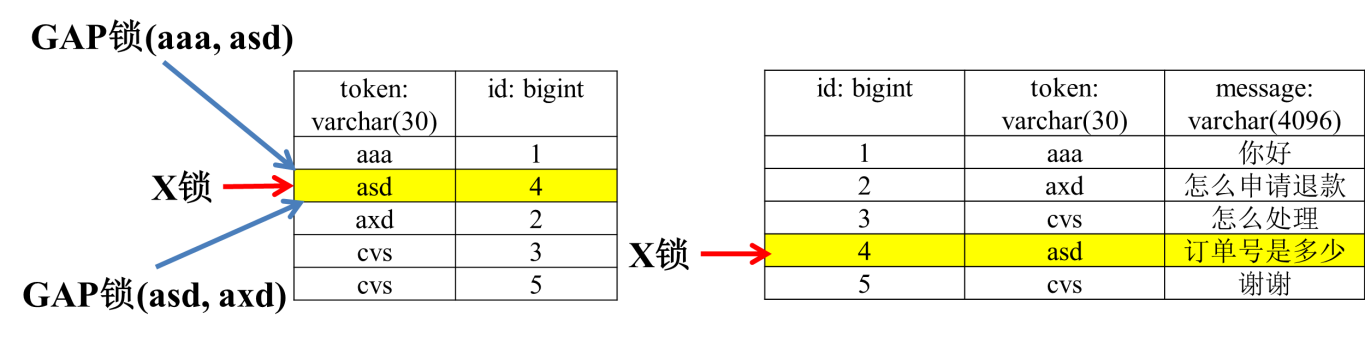
**innodb的RR隔离级别可以避免幻读发生，怎么实现？当然需要借助于锁了！**

为了解决幻读问题，innodb引入了**gap锁。**

在事务A执行：update msg set message=‘订单’ where token=‘asd’;

innodb首先会和RC级别一样，给索引上的记录添加上X锁，此外，还在非唯一索引’asd’与相邻两个索引的区间加上锁。

这样，当事务B在执行insert into msg values (null,‘asd',’hello’); commit;时，会首先检查这个区间是否被锁上，如果被锁上，则不能立即执行，需要等待该gap锁被释放。这样就能避免幻读问题。



**4 如何尽可能避免死锁**

1）**以固定的顺序访问表和行**。比如对第2节两个job批量更新的情形，简单方法是对id列表先排序，后执行，这样就避免了交叉等待锁的情形；又比如对于3.1节的情形，将两个事务的sql顺序调整为一致，也能避免死锁。

2）**大事务拆小。**大事务更倾向于死锁，如果业务允许，将大事务拆小。

3）**在同一个事务中，尽可能做到一次锁定所需要的所有资源，减少死锁概率**。

4）降低隔离级别。如果业务允许，将隔离级别调低也是较好的选择，比如将隔离级别从RR调整为RC，可以避免掉很多因为gap锁造成的死锁。

1. **为表添加合理的索引。**可以看到如果不走索引将会为表的每一行记录添加上锁，死锁的概率大大增大。

**很多人容易搞混不可重复读和幻读，确实这两者有些相似。但不可重复读重点在于update和delete，而幻读的重点在于insert。**

如果使用锁机制来实现这两种隔离级别，在可重复读中，该sql第一次读取到数据后，就将这些数据加锁，其它事务无法修改这些数据，就可以实现可重复读了。但这种方法却无法锁住insert的数据，所以当事务A先前读取了数据，或者修改了全部数据，事务B还是可以insert数据提交，这时事务A就会发现莫名其妙多了一条之前没有的数据，这就是幻读，不能通过行锁来避免。需要**Serializable隔离级别 ，读用读锁，写用写锁，读锁和写锁互斥，这么做可以有效的避免幻读、不可重复读、脏读等问题，但会极大的降低数据库的并发能力。**

所以说不可重复读和幻读最大的区别，就在于如何通过锁机制来解决他们产生的问题。

上文说的，是使用悲观锁机制来处理这两种问题，但是MySQL、ORACLE、PostgreSQL等成熟的数据库，出于性能考虑，都是使用了**以乐观锁为理论基础的MVCC（多版本并发控制）来避免这两种问题。**

**####悲观锁和乐观锁####**

**悲观锁**

正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

**在悲观锁的情况下，为了保证事务的隔离性，就需要一致性锁定读。读取数据时给加锁，其它事务无法修改这些数据。修改删除数据时也要加锁，其它事务无法读取这些数据。**

**数据库中，悲观锁的流程：**

对任意记录进行修改前，先尝试为该记录加上排他锁（exclusive locking）。如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常。 具体响应方式由开发者根据实际需要决定。 如果成功加锁，那么就可以对记录做修改，事务完成后就会释放锁了。 其间如果有其他对该记录做修改或加排他锁的操作，都会等待我们解锁或直接抛出异常。

**悲观并发控制实际上是"先取锁再访问"的保守策略**，为数据处理的安全提供了保证。但是在效率方面，处理加锁的机制会让数据库产生额外的开销，还有增加产生死锁的机会；另外，在只读型事务处理中由于不会产生冲突，也没必要使用锁，这样做只能增加系统负载；还有会降低了并行性，一个事务如果锁定了某行数据，其他事务就必须等待该事务处理完才可以处理那行数。

**乐观锁**

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

而乐观锁机制在一定程度上解决了这个问题。**乐观锁，大多是基于数据版本（ Version ）记录机制实现。**何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，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。

**相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。**

**相对于悲观锁，在对数据库进行处理的时候，乐观锁并不会使用数据库提供的锁机制。一般的实现乐观锁的方式就是记录数据版本。**

实现数据版本有两种方式：

**第一种是使用版本号version，**

**第二种是使用时间戳timestamp（时间戳精度）。**

数据版本,为数据增加的一个版本标识。当读取数据时，将版本标识的值一同读出，数据每更新一次，同时对版本标识进行更新。**当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的版本标识进行比对，如果数据库表当前版本号与第一次取出来的版本标识值相等，则予以更新，否则认为是过期数据。**

DBMS会假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据（修改该数据的事务已提交，或未提交；事务未提交会带来数据丢失）。如果其他事务有更新的话，正在提交的事务会进行回滚。

注：对于以上两种方式,Hibernate自带实现方式：在使用乐观锁的字段前加annotation: @Version, Hibernate在更新时自动校验该字段。

**优势劣势**

**乐观并发控制相信事务之间的数据竞争(datarace)的概率是比较小的， 因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。**

**乐观锁虽然没有依赖数据库提供的锁机制，也可以保证数据一致性。**

要说明的是，MVCC的实现没有固定的规范，每个数据库都会有不同的实现方式，这里讨论的是InnoDB的MVCC。

####**MVCC在MySQL的InnoDB中的实现**

**在InnoDB中，会在每行数据后添加两个额外的隐藏的值来实现MVCC，这两个值一个记录这行数据何时被创建，另外一个记录这行数据何时过期（或者被删除）。 在实际操作中，存储的并不是时间，而是事务的版本号，每开启一个新事务，事务的版本号就会递增。** 在可重读Repeatable reads事务隔离级别下：

SELECT时，读取创建版本号<=当前事务版本号，删除版本号为空或>当前事务版本号。

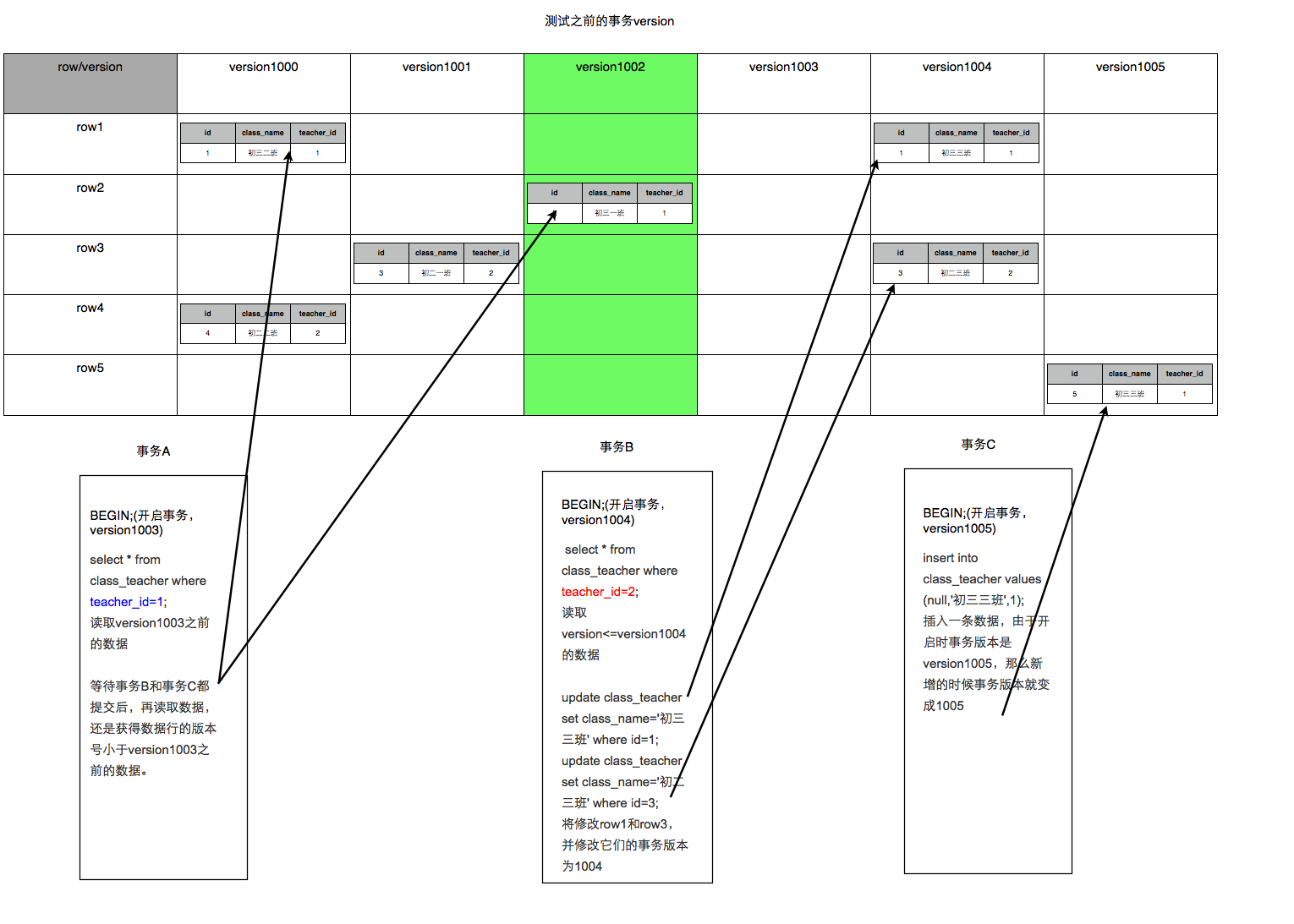
INSERT时，保存当前事务版本号为行的创建版本号

DELETE时，保存当前事务版本号为行的删除版本号

UPDATE时，插入一条新纪录，保存当前事务版本号为行创建版本号，同时保存当前事务版本号到原来删除的行

通过MVCC，虽然每行记录都需要额外的存储空间，更多的行检查工作以及一些额外的维护工作，但可以减少锁的使用，大多数读操作都不用加锁，读数据操作很简单，性能很好，并且也能保证只会读取到符合标准的行，也只锁住必要行。

我们不管从数据库方面的教课书中学到，还是从网络上看到，大都是上文中事务的四种隔离级别这一模块列出的意思，RR级别是可重复读的，但无法解决幻读，而只有在Serializable级别才能解决幻读。于是我就加了一个事务C来展示效果。在事务C中添加了一条teacher\_id=1的数据commit，RR级别中应该会有幻读现象，事务A在查询teacher\_id=1的数据时会读到事务C新加的数据。但是测试后发现，在MySQL中是不存在这种情况的，在事务C提交后，事务A还是不会读到这条数据。**可见在MySQL的RR级别中，是解决了幻读的读问题的。**



**MySQL性能优化的最佳20条**

1.为查询缓存优化你的查询

2.EXPLAIN 你的 SELECT 查询

3.当只要一行数据时使用 LIMIT 1

4.为搜索字段建索引

5.在Join表的时候使用相当类型的例，并将其索引

6.千万不要 ORDER BY RAND()

7.使用 ENUM 而不是 VARCHAR

8.从 PROCEDURE ANALYSE() 取得建议

9.Prepared Statements

1. 垂直分割 也就是拆分表

**聚簇索引(Clustered Indexes)**

**InnoDB存储引擎的数据组织方式,是聚簇索引表：完整的记录,存储在主键索引中,通过主键索引,就可以获取记录所有的列.**

每个InnoDB的表有一个特殊的索引称之为聚簇索引,每行的数据就是存储在聚簇索引中.**通常,聚簇索引和主键同义.**

当你在你的表上面定义一个主键时,InnoDB将其作为聚簇索引.建议为你的表都创建一个主键.如果没有唯一并且非空的一列或者多列(用来做你的主键),那么可以创建一个自动填充的自增列(比如ID)

如果你的表没有定义主键,MySQL会将第一个所有列都非空的UNIQUE索引作为聚簇索引.

如果你的表不存在这样的UNIQUE索引(见上),InnoDB内部会自动隐式生成一个包含行ID的列并在其上面建立聚簇索引.这一列按行ID排序.行ID是一个6-byte的严格单调自增的字段.因此,按照行在物理上是按照插入顺序排序的.

总额来说聚簇索引是由一个或多个列组成，用于唯一性标识数据表中的某一条记录。一个表可以没有主键，但最多只能有一个主键，并且主键值不能包含NULL。

在MySQL中，InnoDB引擎的数据表的主键设计我们通常遵循几个原则：

采用一个没有业务用途的自增属性列作为主键；

主键字段值总是不更新，只有新增或者删除两种操作；

不选择会动态更新的类型，比如当前时间戳等。

之所以这么做的几点优势：

新增数据时，由于主键值是顺序增长的，innodb page发生分裂的概率降低了；

业务数据有变更时，不修改主键值，物理存储位置发生变化的概率降低了，innodb page中产生碎片的概率也降低了。

聚簇所有是如何加速查询的呢?

通过聚簇所有访问一行非常快,这是因为在聚簇索引上搜索会直接定位到包含你需要的行的数据所在的页上(page).

**二级索引(Secondary Indexes)**

**除了聚簇索引其他索引都是二级索引.其可以划分为：唯一索引、非唯一索引。**

**唯一索引其实应该叫做唯一性约束，它的作用是避免一列或多列值存在重复，是一种约束性索引。**

在InnoDB中每个二级索引记录都包含了这一行的主键列和当前这个二级索引包含的列.

InnoDB使用二级索引中包含的主键取索引这一行对应的聚簇索引,进而找到这一行完整的数据.

注意：如果主键很长,则二级索引会占有更多的空间,因此建议使用短的列做主键.

**从查询性能上来说，在MyISAM表中主键索引和不允许有NULL的唯一索引的查询性能是相当的，在InnoDB表通过唯一索引查询则需要多一次从二级索引到主键索引的转换过程。**InnoDB表基于普通索引的查找代价更高，因为每次检索到结果后，还需要至少再多检索一次才能确认是否还有更多符合条件的结果，主键索引和唯一索引就不需要这么做了。

**我们为什么要建索引，原因很简单索引可以提高查询效率，可以类比图书中目录。这里就不再赘述索引的优势，请自行查阅资料。**

**##索引原理**

**除了图书的目录，我们在生活中会经常发现类似的事情，如字典、火车站的车次表等。它们的原理都是一样的，通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随机的事件变成顺序的事件，也就是我们总是通过同一种查找方式来锁定数据。**

**数据库也是一样，但显然要复杂许多，因为不仅面临着等值查询，还有范围查询(>、<、between、in)、模糊查询(like)、并集查询(or)等等。**

**数据库应该选择怎么样的方式来应对所有的问题呢？**

**我们回想字典的例子，能不能把数据分成段，然后分段查询呢？最简单的如果1000条数据，1到100分成第一段，101到200分成第二段，201到300分成第三段......这样查第250条数据，只要找第三段就可以了，一下子去除了90%的无效数据。但如果是1千万的记录呢，分成几段比较好？稍有算法基础的同学会想到搜索树，其平均复杂度是lgN，具有不错的查询性能。但这里我们忽略了一个关键的问题，复杂度模型是基于每次相同的操作成本来考虑的，数据库实现比较复杂，数据保存在磁盘上，而为了提高性能，每次又可以把部分数据读入内存来计算，因为我们知道访问磁盘的成本大概是访问内存的十万倍左右，所以简单的搜索树难以满足复杂的应用场景。**

**###磁盘IO与预读**

**前面提到了访问磁盘，那么这里先简单介绍一下磁盘IO和预读，磁盘读取数据靠的是机械运动，每次读取数据花费的时间可以分为寻道时间、旋转延迟、传输时间三个部分。**

**寻道时间指的是磁臂移动到指定磁道所需要的时间，主流磁盘一般在5ms以下；**

**旋转延迟就是我们经常听说的磁盘转速，比如一个磁盘7200转，表示每分钟能转7200次，也就是说1秒钟能转120次，旋转延迟就是1/120/2 = 4.17ms；**

**传输时间指的是从磁盘读出或将数据写入磁盘的时间，一般在零点几毫秒，相对于前两个时间可以忽略不计。**

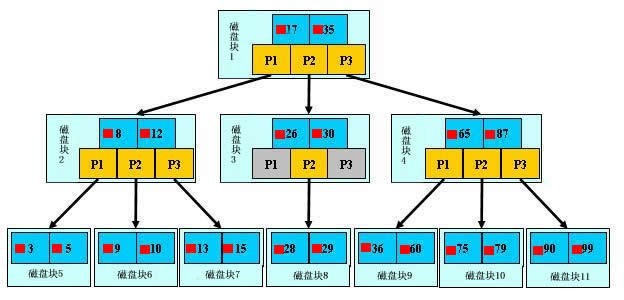
**那么访问一次磁盘的时间，即一次磁盘IO的时间约等于5+4.17 = 9ms左右，听起来还挺不错的，但要知道一台500 -MIPS的机器每秒可以执行5亿条指令，因为指令依靠的是电的性质，换句话说执行一次IO的时间可以执行40万条指令，数据库动辄十万百万乃至千万级数据，每次9毫秒的时间，显然是个灾难。**

**考虑到磁盘IO是非常高昂的操作，计算机操作系统做了一些优化，当一次IO时，不光把当前磁盘地址的数据，而是把相邻的数据也都读取到内存缓冲区内，因为局部预读性原理告诉我们，当计算机访问一个地址的数据的时候，与其相邻的数据也会很快被访问到。每一次IO读取的数据我们称之为一页(page)。具体一页有多大数据跟操作系统有关，一般为4k或8k，也就是我们读取一页内的数据时候，实际上才发生了一次IO，这个理论对于索引的数据结构设计非常有帮助。**

**###索引的数据结构**

**前面讲了生活中索引的例子，索引的基本原理，数据库的复杂性，又讲了操作系统的相关知识，目的就是让大家了解，任何一种数据结构都不是凭空产生的，一定会有它的背景和使用场景，我们现在总结一下，我们需要这种数据结构能够做些什么，其实很简单，那就是：每次查找数据时把磁盘IO次数控制在一个很小的数量级，最好是常数数量级。那么我们就想到如果一个高度可控的多路搜索树是否能满足需求呢？就这样，b+树应运而生。**

**###详解b+树**



**如上图，是一颗b+树，关于b+树的定义可以参见B+树，这里只说一些重点，浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示），如磁盘块1包含数据项17和35，包含指针P1、P2、P3，P1表示小于17的磁盘块，P2表示在17和35之间的磁盘块，P3表示大于35的磁盘块。真实的数据存在于叶子节点即3、5、9、10、13、15、28、29、36、60、75、79、90、99。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如17、35并不真实存在于数据表中。**

**###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。**

**###b+树性质**

**1.通过上面的分析，我们知道IO次数取决于b+数的高度h，假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=㏒(m+1)N，当数据量N一定的情况下，m越大，h越小；而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越多，树的高度越低。这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。当数据项等于1时将会退化成线性表。**

**2.当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数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来确定下一步的所搜方向，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。**

**慢查询优化**

**关于MySQL索引原理是比较枯燥的东西，大家只需要有一个感性的认识，并不需要理解得非常透彻和深入。了解完索引原理之后，我们是不是对慢查询有了一些想法哪？先总结一下索引的几大基本原则：**

**建索引的几大原则**

**1.最左前缀匹配原则，非常重要的原则，mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，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，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。**

**2.=和in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**

**3.尽量选择区分度高的列作为索引，区分度的公式是count(distinct col)/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，那可能有人会问，这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录**

**4.索引列不能参与计算，保持列“干净”，比如from\_unixtime(create\_time) = ’2014-05-29’就不能使用到索引，原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该写成create\_time = unix\_timestamp(’2014-05-29’);**

**5.尽量的扩展索引，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**

**查询优化神器 - explain命令**

**关于explain命令相信大家并不陌生，具体用法和字段含义可以参考官网explain-output，这里需要强调rows是核心指标，绝大部分rows小的语句执行一定很快（有例外，下面会讲到）。所以优化语句基本上都是在优化rows。**

**慢查询优化基本步骤**

**0.先运行看看是否真的很慢，注意设置SQL\_NO\_CACHE**

**1.where条件单表查，锁定最小返回记录表。这句话的意思是把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查询，看哪个字段的区分度最高**

**2.explain查看执行计划，是否与1预期一致（从锁定记录较少的表开始查询）**

**3.order by limit 形式的sql语句让排序的表优先查**

**4.了解业务方使用场景**

**5.加索引时参照建索引的几大原则**

1. **观察结果，不符合预期继续从0分析**

****1.触发器的作用？****  
触发器是一种特殊的存储过程，主要是通过事件来触发而被执行的。它可以**强化约束，来维护数据的完整性和一致性，可以跟踪数据库内的操作从而不允许未经许可的更新和变化。**

可以联级运算。如，某表上的触发器上包含对另一个表的数据操作，而该操作又会导致该表触发器被触发。

****2.什么是存储过程？用什么来调用？****  
存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需创建一次，以后在该程序中就可以调用多次。

如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。可以用一个命令对象来调用存储过程。

****3.索引的作用？和它的优点缺点是什么？****  
索引就是一种特殊的查询表，数据库的搜索引擎可以利用它加速对数据的检索。它很类似与现实生活中书的目录，不需要查询整本书内容就可以找到想要的数据。

索引可以是唯一的，创建索引允许指定单个列或者是多个列。缺点是它减慢了数据录入的速度，同时也增加了数据库的尺寸大小。

****4.什么是内存泄漏？****  
一般我们所说的内存泄漏指的是堆内存的泄漏。堆内存是程序从堆中为其分配的，大小任意的，使用完后要显示释放内存。

在Java中，内存泄漏就是存在一些被分配的对象，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

在C++中，内存泄漏的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于C++中没有GC，这些内存将永远收不回来。在Java中，这些不可达的对象都由GC负责回收，因此程序员不需要考虑这部分的内存泄露。

****5.维护数据库的完整性和一致性，你喜欢用触发器还是自写业务逻辑？为什么？****  
我是这样做的，尽可能使用约束，如check,主键，外键，非空字段等来约束，这样做效率最高，也最方便。

其次是使用触发器，这种方法可以保证，无论什么业务系统访问数据库都可以保证数据的完整新和一致性。

最后考虑的是自写业务逻辑，但这样做麻烦，编程复杂，效率低下。

****6.什么叫视图？游标是什么？****  
视图是一种虚拟的表，具有和物理表相同的功能。可以对视图进行增，改，查，操作，视图通常是有一个表或者多个表的行或列的子集。

**对视图的修改不影响基本表**。它使得我们获取数据更容易，相比多表查询。

游标是对查询出来的结果集作为一个单元来有效的处理。

游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

****7.表空间的管理方式有哪几种？****  
****数据字典管理方式****  
****本地文件管理方式****

****8.说说索引的组成?****  
索引列、rowid

****9.DELETE和TRUNCATE的区别?****  
1、TRUNCATE在各种表上无论是大的还是小的都非常快。如果有ROLLBACK命令DELETE将被撤销，而TRUNCATE则不会被撤销。  
2、**TRUNCATE是一个DDL语言，向其他所有的DDL语言一样，他将被隐式提交，不能对TRUNCATE使用ROLLBACK命令。**3、**TRUNCATE将重新设置高水平线和所有的索引**。在对整个表和索引进行完全浏览时，经过TRUNCATE操作后的表比DELETE操作后的表要快得多。  
4、TRUNCATE不能触发任何DELETE触发器。  
5、不能授予任何人清空他人的表的权限。  
6、当表被清空后表和表的索引讲重新设置成初始大小，而delete则不能。  
7、不能清空父表。

****10.什么是唯一索引？****  
唯一索引可以确保索引列不包含重复的值。

在多列唯一索引的情况下，该索引可以确保索引列中每个值组合都是唯一的。例如，如果在 last\_name、first\_name 和 middle\_initial 列的组合上创建了唯一索引 full\_name，则该表中任何两个人都不可以具有相同的全名。  
聚集索引和非聚集索引都可以是唯一的。因此，只要列中的数据是唯一的，就可以在同一个表上创建一个唯一的聚集索引和多个唯一的非聚集索引。  
只有当唯一性是数据本身的特征时，指定唯一索引才有意义。如果必须实施唯一性以确保数据的完整性，则应在列上创建 UNIQUE 或 PRIMARY KEY 约束，而不要创建唯一索引。例如，如果打算经常查询雇员表（主键为 emp\_id）中的社会安全号码 (ssn) 列，并希望确保社会安全号码的唯一性，则在 ssn 列上创建 UNIQUE 约束。如果用户为一个以上的雇员输入了同一个社会安全号码，则会显示错误。

****11.SQL里面IN比较快还是EXISTS比较快？****  
EXISTS比较快因为EXISTS返回一个Boolean型而IN返回一个值。  
****12.Oracle中，rowid和rownum有什么不同？****  
注意 MySQL不支持行号的使用。  
RowId是一个数据库内部的概念，表示表的一行，用来快速的访问某行数据  
Rownum是结果集的一个功能， 例如select \* from Student where rownum = 2 就是得到结果集的第二行。

****13.delete, truncate 和 drop的区别？****  
Delete命令用来删除表的全部或者一部分数据行，执行delete之后，****用户需要提交(commmit)或者回滚(rollback) transaction 来执行删除或者撤销删除****， delete命令会触发这个表上所有的delete触发器。  
Truncate删除表中的所有数据， 这个操作不能回滚，也不会触发这个表上的触发器，****TRUNCATE比delete更快，占用的空间更小****。

Drop命令从数据库中删除表， 所有的数据行，索引和权限也会被删除，所有的DML触发器也不会被触发，这个命令也不能回滚。

****14.MySQL存储过程常用的方法有哪些？****

创建 MySQL 存储过程的简单语法为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | create procedure 存储过程名字()  (  [in|out|inout] 参数 datatype  )  begin  MySQL 语句;  end; |

可以使用call来调用存储过程。

|  |  |
| --- | --- |
| 1  2  3  4  5 | Java中，类CallableStatement为Java程序提供了一种调用存储过程的方法。CallableStatement对象可以带有用于输入数据 (IN模式参数)，输出结果(OUT模式参数) 或者用于上面这两种功能的参数(IN OUT模式参数)。  下面是在JDBC中调用存储过程的语法。需要注意的是语法中的方括号只是用于表示可选参数的，它不是命令语法的一部分。  {call procedure\_name([?, ?, ...])}  下面是调用可以返回结果参数值的存储过程的语法：  {? = call procedure\_name([?, ?, ...])} |

****15.mysql如何实现分页查询？****

****16.建数据库需要遵循哪些原则？应该注意哪些方面？****

****17.MySQL数据库有哪些类型的索引？****

MySQL主要提供2种方式的索引：B-Tree索引，Hash索引。  
B树索引具有范围查找和前缀查找的能力，对于有N节点的B树，检索一条记录的复杂度为O(LogN)。相当于二分查找。  
哈希索引只能做等于查找，但是无论多大的Hash表，查找复杂度都是O(1)。  
显然，如果值的差异性大，并且以等值查找（=、 <、>、in）为主，Hash索引是更高效的选择，它有O(1)的查找复杂度。  
如果值的差异性相对较差，并且以范围查找为主，B树是更好的选择，它支持范围查找。

****18.如何创建索引，什么时候该创建、什么时候不应该创建？****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | 创建索引的标准语法:  CREATE INDEX 索引名 ON 表名 (列名)  TABLESPACE 表空间名;    创建唯一索引:  CREATE unique INDEX 索引名 ON 表名 (列名)  TABLESPACE 表空间名;    创建组合索引:  CREATE INDEX 索引名 ON 表名 (列名1,列名2)  TABLESPACE 表空间名;    创建反向键索引:  CREATE INDEX 索引名 ON 表名 (列名) reverse  TABLESPACE 表空间名; |

****什么情况下应不建或少建索引****  
1. **表记录太少**如果一个表只有5条记录，采用索引去访问记录的话，那首先需访问索引表，再通过索引表访问数据表，一般索引表与数据表不在同一个数据块，这种情况下DB至少要往返读取数据块两次。而不用索引的情况下DB会将所有的数据一次读出，处理速度显然会比用索引快。  
2. **经常插入、删除、修改的表**  
对一些经常处理的业务表应在查询允许的情况下尽量减少索引。  
3. **数据重复且分布平均的表字段**  
假如一个表有10万行记录，有一个字段A只有T和F两种值，且每个值的分布概率大约为50%，那么对这种表A字段建索引一般不会提高数据库的查询速度。

****19.mysql如何实现分库分表，分哪几个步骤？****  
一般情况下，都可以对主键ID取模，做Hash，散列到多个表中。  
比如我要对User表做分表操作，分散到一百个表中：  
<?php  
for(i=0;i=0;i< 100; i++ ){ //echo "CREATE TABLE db2.members{i++ ){ //echo "CREATE TABLE db2.members{i} LIKE db1.members<br>";  
echo "INSERT INTO members{i} SELECT \* FROM members WHERE mid%100={i} SELECT \* FROM members WHERE mid%100={i}<br>";  
}  
?>

****20.如何实现范式和反范式？****

范式是关系数据库理论的基础，也是我们在设计数据库结构过程中所要遵循的规则和指导方法。  
目前关系数据库有六种范式：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式(4NF）和第五范式（5NF，还又称完美范式）。  
各种范式呈递次规范，越高的范式数据库冗余越小。  
不满足范式的模型，就是反范式模型。

反范式跟范式所要求的正好相反，在反范式的设计模式，**我们可以允许适当的数据的冗余**，用这个冗余去取操作数据时间的缩短。本质上就是用空间来换取时间，把数据冗余在多个表中，当查询时可以减少或者是避免表之间的关联；  
RDBMS模型设计过程中，常常使用范式约束我们的模型，但在NOSQL模型中则大量采用反范式。

****21.举一个反范式设计的例子？****

数据库设计要严格遵守范式，这样设计出来的数据库，虽然思路很清晰，结构也很合理，但是，有的时候，却要在一定程度上打破范式设计。

这里其实并不矛盾，**因为范式越高，设计出来的表可能越多，关系可能越复杂，但是性能却不一定会很好，因为表一多，就增加了关联性。特别是在高可用的OLTP数据库中，这一点表现得很明显。**

**最明显的打破范式的设计方法就是冗余法**，**以空间换取时间的做法，把数据冗余在多个表中，当查询时可以减少或者是避免表之间的关联。**

还是用上面的例子，学生表与课程表，假定课程表要经常被查询，而且在查询中要显示学生的姓名，查询语句则为：

|  |
| --- |
| SQL>select code,name,subject from course c,  student s where s.id=c.code where code=? |

这个语句如果被大范围、高频率执行，可能会因为表关联造成一定程度的影响，现在，假定评估到学生改名的需求是非常少的，那么，就可以把学生姓名冗余到课程表中，又变回了如表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| ID(PK) | 编号 | 姓名 | 选修科目 |
| 1 | 001 | 张三 | 语文 |
| 2 | 001 | 张三 | 数学 |
| 3 | 001 | 张三 | 英语 |
| 4 | 002 | 李四 | 物理 |
| 5 | 002 | 李四 | 化学 |
| 6 | 003 | 王五 | 历史 |
| 7 | 003 | 王五 | 地理 |
| 8 | 003 | 王五 | 生物 |

注意：我这里并没有省略学生表，不过是把学生姓名冗余在了课程表中，如果万一有很少的改名需求，只要保证在课程表中改名正确即可。  
  
那么，修改以后的语句可以简化为：

|  |
| --- |
| SQL>select code,name,subject from course c where code=? |

**字符应该是最常见的一种了，但似乎各个数据库都有所不同，比如oracle中就有啥varchar2之类。不过mysql似乎最多的还是集中在char和varchar上。**

**说说区别。char是固定长度的，而varchar会根据具体的长度来使用存储空间。比如char(255)和varchar(255)，在存储字符串"hello world"的时候，char会用一块255的空间放那个11个字符，而varchar就不会用255个，他先计算长度后只用11个再加上计算的到字符串长度信息，一般1-2个byte来，这样varchar在存储不确定长度的时候会大大减少存储空间。**

**如此看来varchar比char聪明多了，那char有用武之地吗？还是很不少优势的。**

**一，存储很短的信息，比如门牌号码101，201……这样很短的信息应该用char，因为varchar还要占个byte用于存储信息长度，本来打算节约存储的现在得不偿失。**

**二，固定长度的。比如使用uuid作为主键，那用char应该更合适。因为他固定长度，varchar动态根据长度的特性就消失了，而且还要占个长度信息。**

**三，十分频繁改变的column。因为varchar每次存储都要有额外的计算，得到长度等工作，如果一个非常频繁改变的，那就要有很多的精力用于计算，而这些对于char来说是不需要的。**

**还有一个关于varchar的问题是，varchar他既然可以自动适应存储空间，那我varchar(8)和varchar(255)存储应该都是一样的，那每次表设计的时候往大的方向去好了，免得以后不够用麻烦。这个思路对吗？答案是否定的。mysql会把表信息放到内存中（查询第一次后，就缓存住了，linux下很明显，但windows下似乎没有，不知道为啥），这时内存的申请是按照固定长度来的，如果varchar很大就会有问题。所以还是应该按需索取。**