# ZunSQL设计书报告

**1 概要设计和具体功能 4**

**1.1 架构概述 4**

**1.2 具体功能介绍 4**

**1.3 关键难点主要数据结构介绍 7**

**2 功能设计（补充1） 10**

**2.1 SQL语句转译模块 10**

**2.2 VirtualEnvironment模块 10**

**2.3 CacheMgr模块 12**

**2.4 Transaction模块 12**

**2.5 Page模块 12**

**2.6 DBInstance模块 12**

**2.7 Shell模块 15**

**2.8 Node.java模块 16**

**2.7 Node\_index.java模块 17**

**2.8 Node\_itree.java模块 18**

**3 具体的类方法列表 19**

**3.1 SQL解析类Parser 25**

**3.2 字节码生成类CodeGenerator 27**

**3.3 VirtualEnvironment模块 28**

**3.4 Btree模块 30**

**4 经验总结和规模 50**

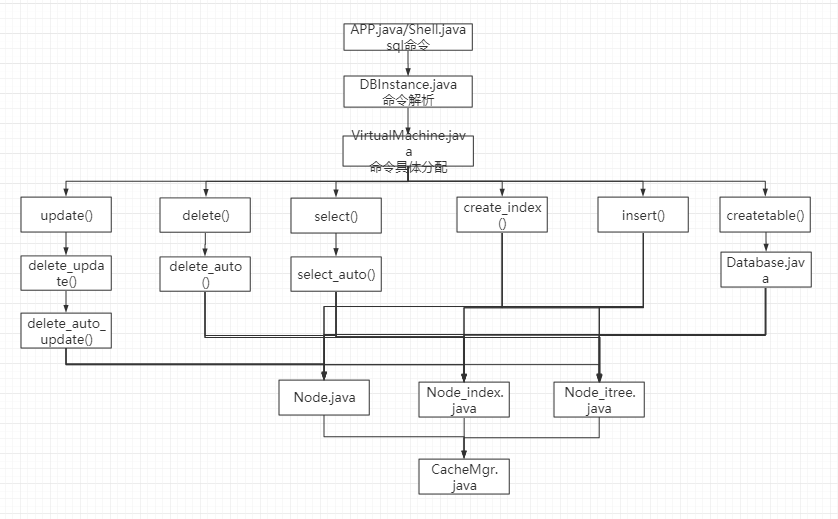
**4.1 经验总结 52**

**4.2 规模 53**

# 概要设计和具体功能

1. **架构概述**

采用了java程序编写，以类的相互操作为基础，用多个.java文件进行交互，实现对数据库文件的创建，关闭，再重复打开，对数据库的增删改查和创建索引等功能，在内存中使用了两级cache，用来加速对文件的读写，具体流程为从DBInstance.java进行命令解析，从VirtualMachine.java进行命令调度，从各个树节点类调用自身的方法进行具体操作，在底层的文件CacheMgr.java对文件读写进行统一，流程图如下：



1. **具体功能介绍**
2. 实现update功能（原先不完整）；
3. 实现delete功能（原先不完整）；
4. 实现insert功能（原先不完整）；
5. 实现select功能（原先不完整）；
6. 实现createtable功能（原先不完整）；
7. 实现createindex功能（原先没有）；
8. 实现同名表格，同名索引的查重功能（原先没有）；
9. 实现select的多个表格的连接查询（原先没有）；
10. 实现select \* 功能（原先没有）；
11. 实现select 多个不重复列名 from功能（原先没有）；
12. 实现select对重复结果的删除功能（原先没有）；
13. 实现select时间显示功能和查询分组功能（原先没有）；
14. 实现不带括号的任意and or组合的删改查功能（原先没有）；
15. 实现delete from 表名不带where条件功能（原先没有）；
16. 实现B树组织表格，B树组织索引功能（原先没有）；
17. 实现B树插入任意重复键的功能（原先没有）；
18. 实现对多个键创建索引的功能（原先没有）；
19. 实现伴随插入删除修改自己更新索引对应内容功能（原先没有）；
20. 实现自动根据有无索引来进行查询的功能（原先没有）；
21. 实现二级cache功能（原先没有）；
22. 实现未用页的收集功能（原先没有）；
23. 实现最近最少使用的二级cache功能（原先不完整）；
24. 实现对一些sql命令的识别和原程序的查错和重新组织；
25. 实现重复读写相同文件功能（原先没有）；
26. 实现对文件的dump功能（原先没有）；
27. 实现对master表和其他表一样用B树组织用于超过很多数量表的创建（原先没有，原先的master表的根节点和页号固定，创建的表数目超过一定数量后会出现错误）
28. **关键难点主要数据结构介绍**
29. **B树**

整体的表格和索引采用B树来组织，B树有下面的特征：

1. 每个节点最多有M个关键字；
2. 根节点最少可以只有一个关键字；
3. 除了根节点至少有M/2-1个关键字；
4. 每个节点的关键字都是从小到大排序；
5. 每个节点指向儿子节点的指针数比关键字数多1；
6. 每个节点的的关键字所在位置的左子树中的关键字都小于它，关键字所在位置的右子树的关键字都大于等于它；
7. 所有叶子节点在同一层；

在实现时为了能够更方便进行B树的操作，做了下面的一些设定：

1. 和B树节点的键值位置相同的指针指向的是B树的左子树，比B树节点的键值位置多1的指针指向的是B树的右子树；
2. 每个B树节点增加fatherNodeID指向父节点的页号，当fatherNodeID是-1时表示没有父节点；
3. 加入两种构造方法，一种用来对已有的B树节点进行读取，这个构造方法会直接调用CacheMgr的读取页的操作，另一种构造方法申请新的B树节点，这个构造方法也会直接调用CacheMgr的分配页的操作，并在得到分配的新页后还要进行一次写回操作，否则会让新申请的页丢失；
4. 加入intoBytes方法，用来对B树节点的内容进行写回操作，这个方法调用CacheMgr的writePage方法，对需要将B树节点页进行跟新的内容进行写回，具体写回的方式由CacheMgr进行组织；
5. 加入update\_root方法，用来在删除，插入，更新过后的根节点的变化进行写回，这个方法直接操作了master中对应的表的内容，如果是Node\_itree，这个方法直接操作了master表和索引表记录的根节点的更新；
6. 加入public Node rhizine，指向根节点的指针，在根节点因为插入，删除，更新而调整后，不需要重新读取一遍根节点，而可以就用原来是根节点的节点，只要操作还是调用原来根节点的方法，就不会因为根节点更新产生错误；
7. 加入pageOne来指明当前节点的页号，便于进行B树的操作；
8. 所有内部节点的键值个数比指针个数少1，所有叶子节点的指针数为0；
9. 每个键值对应表中的一个记录，键值中列的位置和表中列的位置也对应，如果表中有主键，那键值中也会有主键，并且主键所在的位置和表中主键对应列所在位置相同；
10. 在搜索需要插入的叶子节点时，如果搜索的不是叶子节点在找到第一个大于插入键值的键值位置后停下，这样可以允许插入多个重复键；
11. 在有很多重复键的情况下，那么和键值位置相同的指针位置的子树中，可能会存在和这个键值一样的键值，在比键值位置多1的指针位置的子树中，也可能会存在和这个键值一样的键值；

对B树的插入操作，原理如下：

1. 搜索B树，找到可以插入对应键值的叶子节点，并插入；
2. 如果当前叶子节点的键值个数小于等于M则完成插入，否则进行第3步；
3. 把当前节点的键值和指针分成两份，处在中间的键值上升到父节点中成为父节点中新的键值，和这个键值相等的指针位置指向当前节点分成的键值较小的一部分，比这个键值多1的指针位置指向当前节点分成的键值较大的一部分，当前节点指向父节点，如果父节点的键值数小于等于M的插入完成，否则把父节点变成当前节点，继续进行第3步；

对B树的删除操作，原理如下：

1. 找到对应键值所在的B树节点，如果这个节点是叶子节点，且删除后的键值个数大于M/2-1则删除结束，否则进行第2步，如果这个节点不是叶子节点，从和要删除位置相等的指针指向的子树的叶节点中寻找最大值，或者从比删除位置大1的指针指向的子树的叶子节点中寻找最小值，把这个值覆盖要删除的值，然后在叶子节点中删除对应的值，如果叶子节点删除后键值个数大于M/2-1则删除结束，否则进行第2步；
2. 如果当前节点的兄弟节点的键值个数大于M/2-1，且如果是当前节点的左兄弟，则把父节点比指向当前节点指针位置少1的键值位置的键值加入当前节点，并把左兄弟的最大键值加入到父节点加入到当前节点的键值的位置，如果是当前节点的右兄弟节点，则把父节点和指向当前节点指针位置一样的键值位置的键值加入当前节点，并把右兄弟的最小键值加入到父节点加入到当前节点的键值的位置，删除结束，如果当前节点的兄弟的键值个数小于M/2-1，则把父节点对应指向兄弟节点和当前节点的指针位置的键值删除并加入到兄弟节点和当前节点合并成的新节点中，删除原先父节点指向当前节点或兄弟节点的指针，而指向新合并的节点，如果父节点的键值个数大于M/2-1则删除结束，否则，把父节点变成当前节点，继续进行第2步；

对B树的查找操作，原理如下：

1. get\_node\_all（Node root, Transaction thisTran）方法，得到传入的root根节点和根节点对应的子树的所有键值并存入CacheMgr的相应结果中，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作；
2. search\_greater\_all(Node root, Cell key, Integer keyp, Transaction thisTran)方法，搜索以root为根节点的B树中主键值比key中主键值大的键，且主键所在的列为keyp的所有键并存入CacheMgr的相应结果中，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体是先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
3. search\_letter\_all(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中主键值比key中主键值小的键，主键所在的列为keyp的所有键并存入CacheMgr的相应结果中，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
4. search\_equal\_all(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中主键值和key主键值一样的键，主键所在的列为keyp的所有键并存入CacheMgr的相应结果中，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
5. search\_greater\_or\_equal\_all(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中主键值大于或等于key中主键值的键，主键所在的列为keyp的所有键并存入CacheMgr的相应结果中，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
6. search\_letter\_or\_equal\_all(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中主键值小于或等于key中主键值的键，主键所在的列为keyp的所有键并存入CacheMgr的相应结果中，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
7. search\_greater\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中列号为keyp的属性值大于key中列号为keyp的属性值的键，由于没有主键，需要对整个B树进行搜索，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
8. search\_letter\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中列号为keyp的属性值小于key中列号为keyp的属性值的键，由于没有主键，需要对整个B树进行搜索，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
9. search\_equal\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中列号为keyp的属性值等于key中列号为keyp的属性值的键，由于没有主键，需要对整个B树进行搜索，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
10. search\_greater\_or\_equal\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中列号为keyp的属性值大于等于key中列号为keyp的属性值的键，由于没有主键，需要对整个B树进行搜索，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
11. search\_letter\_or\_equal\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) 搜索以root为根节点的B树中列号为keyp的属性值小于等于key中列号为keyp的属性值的键，由于没有主键，需要对整个B树进行搜索，使用Stack<Node>s1=new Stack<Node>()来避免函数的压栈入栈操作，尽量加快操作速度，具体先将根节点入栈，用while循环，当栈非空时，出栈一个元素，将这个元素的对应节点取出，并对它的键值进行搜索，如果这个节点还有儿子节点，把儿子节点也进行入栈；
12. **cache**

在CacheMgr中使用两级cache，第一级cache用事务号对应的Map来组织，Map<Integer, List<Page>> transOnPage，主要用来存储对应事务号所写回的页，当调用CacheMgr中的方法进行写回时并不直接进行写回，而是放到当前事务号所对应的写回页中，每次新写回的页都加入到当前事务写回页的最后面，当超过20页时，由当前事务写回页的最前面的页被替换到cacheList中，cacheList是第二级cache，并且用ConcurrentMap<Integer, Integer> cachePageMap来加速对对应页号是否在cacheList中的判断，cacheList也是最新加入的页放入最后面，最久没有使用的页放入最前面，当cacheList的页数超过20页时，一次将前5页写回硬盘；

在进行readPage方法调用的时候，先访问对应事务号的页表中是否有该页，如果有则返回，如果没有则访问cacheList，判断是否有这个页时用ConcurrentMap<Integer, Integer> cachePageMap加速，因为cacheList是线性结构，顺序访问很慢，如果在cacheList中，则将这页重新调整到cacheList的最后面，实现最近最少使用的方法，如果这页不在cacheList中则从文件中读取对应页号的页，并加入到cacheList的最后，在ConcurrentMap<Integer, Integer> cachePageMap中加入(页号，1)。

1. **修改查询删除where中不带括号的and or选择和查找中多个表的选择以及列号的选择带\*的选择**

在带where条件的sql语句中，假设where条件部分是如下：

1. Where 表1的列名1 =或>=或<=或>或< 属性值 and或or 表1的列名2 =或>=或<=或>或< 属性值 and或or 。。。 表1的列名n =或>=或<=或>或< 属性值 and或or 表2的列名1 =或>=或<=或>或< 属性值 and或or 。。。表m的列名i =或>=或<=或>或< 属性值 and或or。。。
2. 在update和delete中只会有一张表，但select中会有多张表，并且from后表的出现顺序和上面where后表的编号顺序并不对应；
3. 比较符号只有>,<,=,>=,<=；
4. 可以任意选择列名，但只支持不同表的列名完全不一样的情况；
5. 把由and或者or分隔的条件变成一个独立的部分，由or分隔的是一个大的部分，由and分隔的部分都在由or分隔的部分里面；
6. 在识别where条件结束后再加入一条instruction，这个instruction以Operator开头赋值是public enum OpCode中的Operator，p1操作数设置为or，p2,p3都设置为null，用来分隔所有的where条件；

对where条件进行特殊处理，在DBInstance识别相应的sql语句时，修改原有的识别顺序，在每次读到where之后，将按照如下的顺序进行识别：

1. 在Instructions中加入一条instruction，这个instruction以BeginFilter开头赋值是public enum OpCode中的BeginFilter，且这个instruction的3个操作数p1,p2,p3都设置为null；
2. 在Instructions中加入一条instruction，这个instruction以Operand开头赋值是public enum OpCode中的Operand，p1操作数设置为第一个and或者or分隔出的一部分的列名，p2,p3都设置为null；
3. 在Instructions中加入一条instruction，这个instruction以Operand开头赋值是public enum OpCode中的Operand，p2操作数设置为第一个and或者or分隔出的一部分的属性值，p1,p3都设置为null；
4. 在Instructions中加入一条instruction，这个instruction以Operator开头赋值是public enum OpCode中的Operator，p1操作数设置为第一个and或者or分隔出的一部分的比较符号，p2,p3都设置为null；
5. 在Instructions中加入一条instruction，这个instruction以Operator开头赋值是public enum OpCode中的Operator，p1操作数设置为第一个and或者or分隔出的一部分的and或者or符号，p2,p3都设置为null；
6. 重复进行第2步没有条件后，在Instructions中先加入一条instruction，这个instruction以Operator开头赋值是public enum OpCode中的Operator，p1操作数设置为or，p2,p3都设置为null，再加入一条instruction，这个instruction以EndFilter开头赋值是public enum OpCode中的EndFilter，p1,p2,p3都设置为null；
7. 在Instructions中加入一条instruction，这个instruction 以Execute开头赋值是public enum OpCode中的Execute，p1,p2,p3都设置为null；
8. 在Instructions中加入一条instruction，这个instruction 以Commit开头赋值是public enum OpCode中的Commit，p1,p2,p3都设置为null；

把sql语句识别完后进入VirtualMachine.java，修改相应模块，把所有得到的where后的条件用where\_condition全部存起来，等待进行相应的操作。

在确定是进行更新，删除，还是查询后，对where\_condition按照or分成不同的部分，每个部分的内部只有and，具体如下：

1. sname1=’…’ and sname2=’…’ and sname3=’…’and。。。snamen=’…’or

如果是select，假设select除掉where部分的格式为：

1. select sname 。。。from 表1,表2,表3。。。

取出由or分隔的一个大部分，这个大部分内部只有and分隔，将只有and分隔的列名属性和比较符寻找对应的表格，并把所有条件都列到selectfrom后的表格的下方，例如，sname1在表1中，sname3在表1中，sname9在表1中，表1中的条件就有三个，sname1=’…’，sname3=’…’，sname9=’…’

对一个大部分的所有由and分隔的条件找到对应的表格并放在表格的下方后，对from后的表格依次进行如下：

1. 按顺序找到一个表格，在这个表格下的条件中寻找是否存在需要比较的列名是主键，或者已经在这个列名上建立了索引，如果找到，就把这个列名和对应的条件调整到第一个；
2. 用这个表的表名在master表中查找，得到表对应的B树的根节点所在页，用Node.java的构造方法，得到B树根节点，接着，调用Node.java的对应方法，用主键或索引或不用键进行查找，查找的结果放进CacheMgr的对应部分中；
3. 查找结束后，用这个表的其他列名对应的条件对找到的结果再进行选择，由于在内存中，过程会比查找硬盘更快；
4. 重复第1步，直到所有selectfrom后的表格都已经查找完；
5. 对每个表查找的部分已经用分隔符进行分隔，之后，用select后的列名对查找最终的结果的相应列进行选择，并放入List<String> l2=new ArrayList<String>();中，如果对应列名的搜索结果为空，则加入空字符来代替

所有select 后的列名的结果按顺序放入List<List<String>> l1=new ArrayList<List<String>>();中；

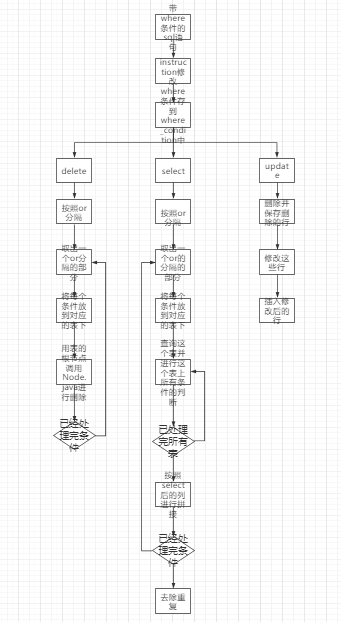
1. 用Stack<c1> st1=new Stack<c1>()避免函数入栈出栈操作，class c1中有3个成员变量，Integer i2用来表示对应的是select中的第几个列名，Integer i1用来表示已经合并到这个列名对应搜索结果串的第几个，List<String> l1存储这个列名所对应的所有结果，用while循环直到栈空为止，用String s1得到完整的由select列名进行选择的合并的结果，s1初始设置为空串，在不断出栈入栈时加入把对应的串合并到s1的尾部，或从s1的尾部删除；
2. 如果select后是\*，就直接把每个表的搜索结果按Row行合并在一起，用Stack<c1> st1=new Stack<c1>();避免函数入栈出栈操作，class c1中有两个成员变量，Integer i1用来表示当前表格已经合并到搜索结果中的第几行，Integer i2用来表示当前表格在CacheMgr的search\_result中的第几个，每个表格的搜索结果之间用分隔符分隔，具体的位置由Integer [] position=new Integer[10000]指向某个表在search\_result中的开始不是分隔符的位置，值是position[i2] 用while循环直到栈空为止，得到完整的由select\*进行选择的合并的结果；

在一个由or分隔的部分搜索完后，继续取出下一个由or分隔的部分，重复进行上述操作，直到把条件处理完，最后去除重复结果。

如果where之前是update或者delete，由于只有一个表，并且没有select后的列选择，只要按照上述操作，先按or对where条件分隔，再对所有分隔的部分顺序处理，每次处理只要调用Node.java中的相应成员方法，就可以完成规定操作；

处理delete时用deleteRow\_nokey\_pre(Node root, List<Cell> keys, List<Integer> keyps, List<String> condition, Transaction thisTran)作为Node.java的成员方法调用的入口，之后再由这个方法调用其他方法，具体删除所有对应项，其中，keys是一个由or分隔的部分的所有and条件的属性值，keyps是所有and条件的列名，condition对应and条件的比较符。

如果是update，由于在更新B树之后，对应键值的位置不一定还在B树中原来的位置，所以先按照where后的条件进行删除，之后再更新删除的结果后，对删除的结果进行插入，最后完成update操作。



1. **Java序列化的操作**

由于原来的程序对文件的读写操作主要由Java序列化来完成，将Java对象序列化为二进制文件，并且整个程序的文件读写操作都是在序列化，反序列化中完成，给原来程序的修改带来了很多困难，具体如下：

1. 虚拟机的反序列化，不仅要求类路径和功能代码一致，还要求两个类的序列化 ID一致（private static final long serialVersionUID = 1L）。序列化 ID 在 Eclipse 下有两种生成策略，一个是固定，一个是随机生成，使用默认的 1L确保代码一致时反序列化成功；
2. 序列化不保存静态变量，并且要求实现Serializable接口才允许序列化；
3. Java 序列化机制为了节省磁盘空间，具有特定的存储规则，当写入文件的为同一对象时，并不会再将对象的内容进行存储，而只是再次存储一份引用，反序列化时，恢复引用关系， 第一次写入对象以后，第二次再试图写相同的对象时，虚拟机根据引用关系知道已经有一个相同对象已经写入文件，因此只保存第二次写的引用，这样第二次读取会得到和第一次读取相同的对象值；
4. **索引的组织和创建索引的自动修改**

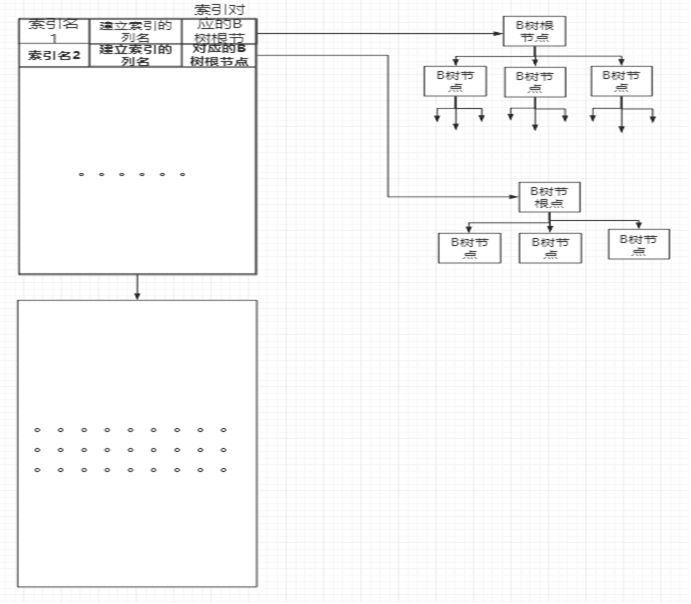
为了能够在原来的程序中加入索引，对原来的master表进行重新的组织，master表的列名由原来的tableName，pageNumber两个修改为tableName，pageNumber，indexpage三个，其中indexpage是指向索引表节点的指针，当没有索引时值为-1，当有索引时值为索引表所在的页号。

一个索引表保存了原来一个普通表的所有索引信息，主要是如下：

1. 索引的名字；
2. 一个索引名对应的原来表重新按照索引列名组成的B树的根节点页号；
3. 在上面建立这个索引的列的名字，会有不止一个列名字；
4. 索引表对应的普通表的名字；
5. 当前索引表节点所在的页；
6. 当前索引表节点的儿子节点所在的页，如果没有是-1；

为了方便对索引表的访问用另外定义的Node\_index.java控制索引表的访问和增加，这个节点具体有以下成员方法：

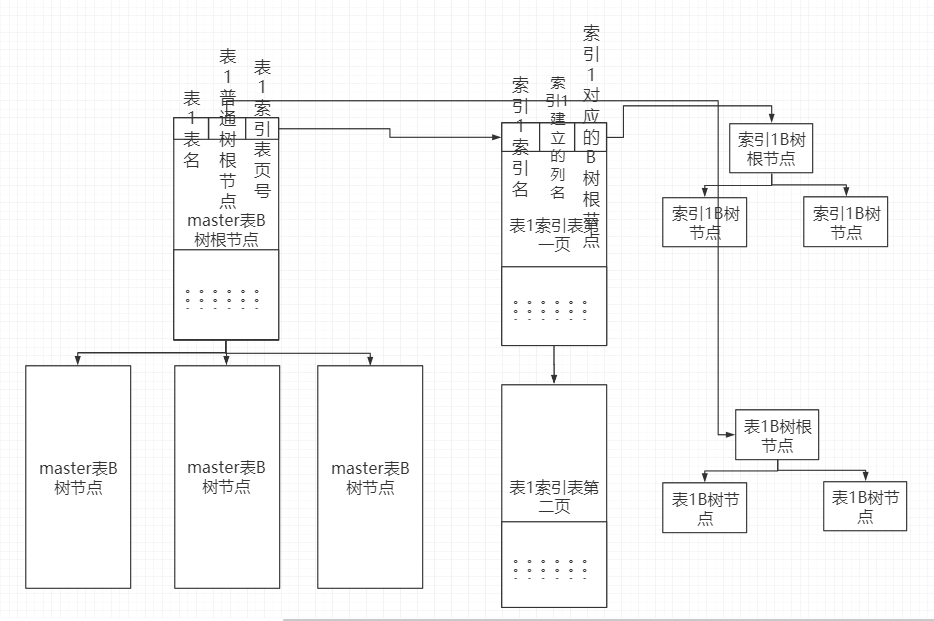
1. Node\_index(int thisPageID, CacheMgr cacheManager, Transaction thisTran)构造方法，用来读取相应页号的索引表的一页；
2. intoBytes (Transaction thisTran)方法，用来将对索引表的更新写回；
3. add(int thisPageID, String indexname2,List<String> keynames2,CacheMgr cacheManager,Transaction thisTran)方法，用来对索引表增加一项新的索引，当已有的索引个数小于20个时，继续在索引表对应页顺序增加一个索引，如果当前页的索引个数大于20个，且没有儿子节点，就新建一页，并把索引写到新页上，让儿子节点的页号等于新建页的页号，新建页的儿子节点的页号是-1，如果有儿子节点，就递归调用儿子节点的add方法加入新的索引；

****

索引节点对应的B树用Node\_itree.java进行控制，和普通表的Node.java有相同的成员方法，但成员变量中多了当前B树对应的索引名和建立索引的列名，具体的比较也用下面的函数进行：

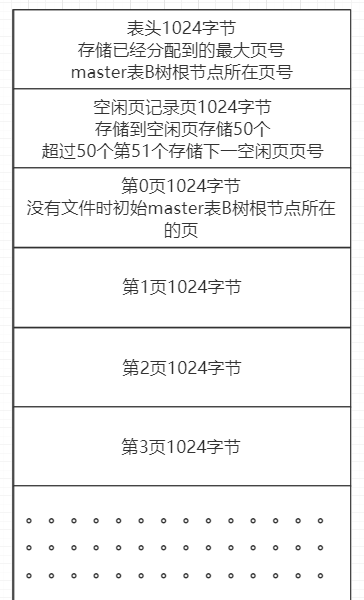
1. boolean bigerThan(Row r1,Row r2)记录r1如果在索引所在的列中大于记录r2，就返回真，否则返回假，当索引在多个列上建立时，默认用第一个列名所在记录的列优先比较，如果大于返回真，如果等于继续比较下一个列名所在记录的列，如果小于返回假；
2. boolean letterThan(Row r1,Row r2) 记录r1如果在索引所在的列中小于记录r2，就返回真，否则返回假，当索引在多个列上建立时，默认用第一个列名所在记录的列优先比较，如果小于返回真，如果等于继续比较下一个列名所在记录的列，如果大于返回假；
3. boolean equalTo(Row r1,Row r2) 记录r1如果在索引所在的列中等于记录r2，就返回真，否则返回假，当索引在多个列上建立时，默认用第一个列名所在记录的列优先比较，如果等于继续比较下一个列名所在记录的列，如果不等于返回假；

为了能在增删改时同时修改普通表的所有索引树对应的B树，需要再在VirtualMachine.java中的增删改部分增加对master表第三列索引表的访问，获得索引表的所有索引对应B树的树根，由于有多个B树，把所有B树的树根放在List<Node\_itree> nit2=new ArrayList<Node\_itree>()中，进行增删改的同时遍历nit2中的所有B树也进行同样的操作，具体的增删改可以用Node\_itree.java中的成员方法来完成。



1. **页和文件的存储格式**

文件分成文件头页，未用页记录页，master表根节点页和其他页，每页的大小是1024字节，其中，第0页固定从文件的2048字节处开始，具体如下：

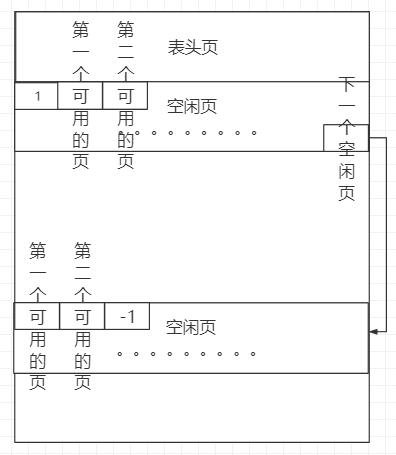


1. **master表空闲页的文件组成访问**

master在还没有文件时，在创建文件的同时，用Database.java中的addMaster(Transaction initTran, int M)方法，创建一个空的B树，且这个树的根节点初始页号是0，当不断有新创建的表被加入时，B树的根节点会发生变化，这时，在文件的头部（第-2页）的第二项中记录下B树根节点的页号，每次访问B树根节点时，必须先访问第-2页中对应的B树根节点页号，再根据页号访问master表的B树根节点。



空闲页由CacheMgr.java中的unusedList\_PageID来保存在内存中，如果文件还没有创建，空闲页记录表的第一页中第一项如果是1就代表有空闲页，如果是-1就代表没有空闲页，如果文件已经创建，每页空闲页最多不超过50页，且最后以-1结束代表空闲页记录表终止。



1. **对已创建的文件不重复创建重复打开相同文件**

在过程中遇到了很多困难，例如调试和对已经修改的程序的内容的记录，对所有.java文件保持修改后的一致性，在调试时遇到了插入很多条记录后，始终出现异常的情况，由于原有的程序已经被改了很多，对问题的发现不得不一个个点进去，最后才进入了第40页和第25页出现异常的页，又经过多次调试才发现是未用页在unusedList\_PageID.size()==0时，没有对原先已经创建的页的空闲页进行修改，导致原先被分配出去的空闲页，在第二次打开文件时继续当成空闲页被分配出去，使得出现了读写异常的情况。

对PageID的分配进行修改，不是每次将PageID增加，而是先查空闲页表，再对原先文件的最大页号增加，分配新的页。

在重复读写相同文件时，也遇到了writePageList.size()<20时，没有在事务对应的页进行查重，寻找相同的页并替换，导致和文件重复读的其他错误混在一起，很难分清具体的问题，也调试了很长时间。

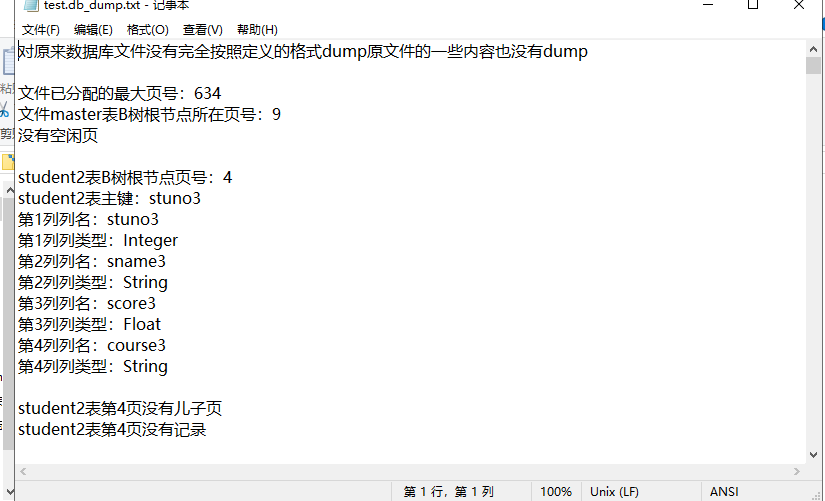
1. **Dump（函数是DBInstance.java的dump（））**

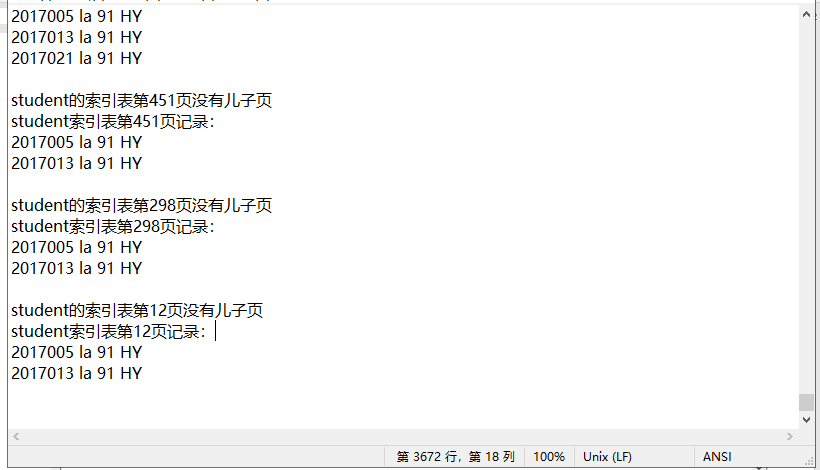
为了能够进行dump操作，把原先java序列化的内容按照一定的格式dump到另一个文件中，需要能够遍历原先的master表，在一个个访问master表的各个记录，一个记录分成三项，表名，表的页号，索引列表的页号，为了尽量加快访问速度，使用栈结构，主要用到：

1. Stack<Node>st1=new Stack<Node>();
2. Stack<Node\_itree>st2=new Stack<Node\_itree>();
3. Stack<Node>master=new Stack<Node>();

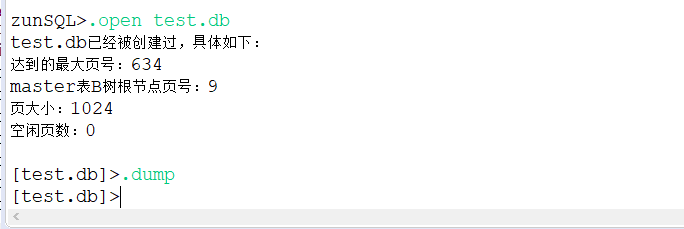
最外层的master栈用来访问master表的B树，一个节点的每个记录采用st1栈来访问对应表的B树，再先进入Node\_index索引列表，接着，一个个访问索引列表的每一项，将每个索引对应的B树都用st2进行遍历，文件头和空闲页已经在最开始处理过。



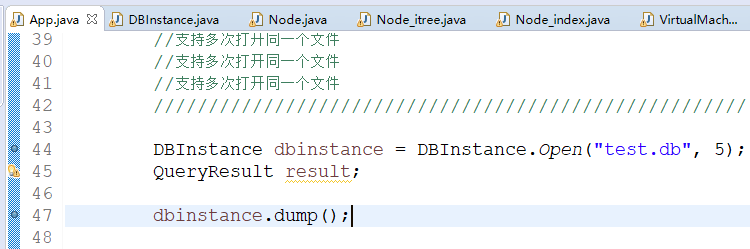




在Shell.java下执行：



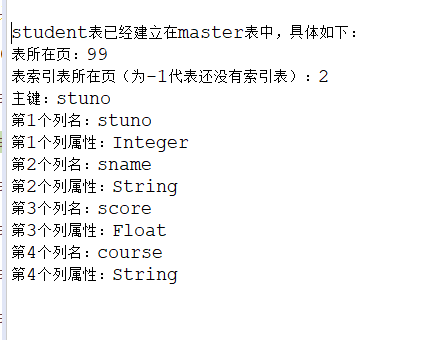
在App.java下执行：

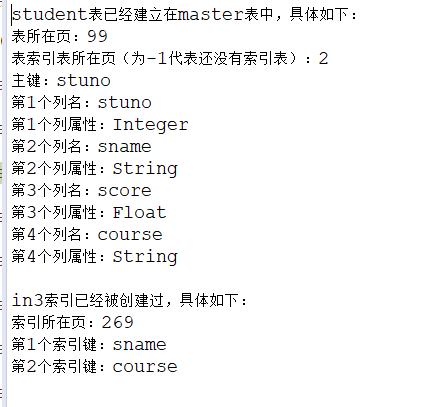


1. **对重复表和重复索引创建的提醒**

在输入创建表和创建索引的时候，在先检查一遍master表和对应的索引列表，如果已经有这个名字存在就不再继续创建表和索引，并输出已经创建过的表和索引的相关信息，具体在Database.java中的createTable(String tableName, String keyName, List<String> columnNameList, List<BasicType> tList,Transaction thisTran)函数和VirtualMachine.java的create\_Index()函数。

在App.java下执行：





1. **功能设计（补充1）**
2. **SQL语句转译模块**
3. **SQL 语句解析接口：**

模块提供了 Parser 类的静态方法 parse 进行 SQL 语句解析，使用方法非常简单，直接 Parser.parse(“select \* from t”) 这样使用即可，参数为单一 SQL 语句，结尾不需要分号。如果一个文件中有多个以分号隔开的 SQL 语句，只需：全部读入 -> 用分号切割开 -> 逐条传递给 Parser.parse 方法解析，即可。Parser.parse 返回值是 Relation 类型，这个类型是所有关系语句语法树节点的公共父类，使用者无需关心，只需要将结果传递给字节码生成模块即可。

1. **字节码生成接口：**

本模块提供了 CodeGenerator 类的静态方法 GenerateByteCode 方法，使用方法同样非常简单，只需传入 Parser.parse 的返回值即可。该方法接受 Relation 类型的参数，返回一个 Instruction （字节码）类型的列表。将此列表传递给虚拟机模块即可。

1. **新加入的对sql原始句子的识别：**

**原先的句子分析存在一些错误，有：**

1. 对where条件识别有很多错误和顺序混乱的地方；
2. 对Begin关键字有的地方没有加；
3. 在where后的第一个条件后的and或or没有识别；
4. 在where条件识别的最后加上原本没有的and或or关键字；
5. 原先也不能识别student.sname这样的访问，会把student和sname识别错误，把student当成属性
6. Select后选择的列名和from后选择的表名有限制；
7. 没有创建索引的关键字；

新加入对创建索引语句的识别功能，先在OpCode.java中加入Index,

Index\_Column两个关键字，之后，在DBInstance.java中加入index(String s1)函数，用来判断，整个sql语句中是否有index关键字，如果没有就先按照原来程序的处理来识别句子，之后再对句子识别中错误的部分做出调整，如果有就重新扫视整个创建索引的sql语句用来得到相应的命令，对原来句子识别错误的修改在第一部分有介绍，对创建索引句子的识别具体如下（假设instructions是识别出的单个instruction的按顺序构成的结构，在DBInstance.java中具体是List<Instruction> Ins=new ArrayList<Instruction>();，并且为了能够对原始的sql语句更好地进行判断，创建了index(String s1)，where(String s1)，where2(String s1)，where3(String s1)函数辅助识别过程，下面只是大概流程，不具体介绍函数调用等详细过程）：

1. 在空的instructions中加入一条Instruction，这个Instruction的OpCode是OpCode.Transaction，p1,p2,p3操作数是null；
2. 在instructions中加入一条Instruction，这个Instruction的OpCode是OpCode.Index，p1是索引的名字,p2是要建立的索引的表的名字,p3操作数是null；
3. 在instructions中加入一条Instruction，这个Instruction的OpCode是OpCode.Index\_Column，p1是要建立索引的第一个列名,p2,p3操作数是null，重复第3步直到所有要建立索引的列名都读入完；
4. 在instructions中加入一条Instruction，这个Instruction的OpCode是OpCode.Index\_Column，p1是要建立索引的第一个列名,p2,p3操作数是null，重复第3步直到所有要建立索引的列名都读入完；
5. 在instructions中加入一条Instruction，这个Instruction的OpCode是OpCode.Execute，p1,p2,p3操作数是null;
6. 在instructions中加入一条Instruction，这个Instruction的OpCode是OpCode.Commit，p1,p2,p3操作数是null;

在识别完之后将instructions调用VirtualMachine.java中的方法按顺序进行解析和分配执行。

1. **VirtualEnvironment模块（在原程序中所有的记录属性不论是字符串还是数字都是用String来表示，并通过Cell中的字符串比较区分大小，没有考虑到比较的可能是数字的情况，改写Cell中的方法，先把String用match方法和数字进行匹配，如果是数字就比较数值大小，如果是字符串继续按字符串的字符串比较，具体为sValue.matches("-?[0-9]+.?[0-9]\*")，原程序虚拟机模块存在很多错误，完全不能做增删改查的功能，也不能进行运算操作，进行连接运算的join函数忽视了利用where后的条件搜索B树结构，只是搜索整个表（表结构没有实现B树插入20条以上数据会崩溃，插入多条数据也没有对原表结构的根节点进行更新），和没有主键一样，也没有实现列选择，也没有实现不同表对应不同条件的查询结果的拼接，和原来Zunsql设计书介绍的存在很大差距,对增删改查创建表和索引的分派都做了整体的修改）**

本模块接收上层模块提供的instruction类型的列表，并对该命令进行解析，首先更新状态，然后执行命令，其中用到选择器和解释器，并调用下层的方法来完成命令。本层同时完成了一些其他功能，比如说解析并计算表达式查询的值以及数据库自然连接操作等等。值得注意的是本模块内大部分机制都是内部机制，比如过滤器，计算和链接操作等等，提供给上层的只有“字节码”类和虚拟机的执行方法。

设计的指令集：

约定：不支持计算

数据库支持的数据类型：1）整数 Integer 2）字符串 String 3）浮点型 Float

虚拟机支持的指令格式为：opcode(String) p1(String) p2(String) p3(String)

|  |  |  |  |
| --- | --- | --- | --- |
| **opcode** | **p1** | **p2** | **p3** |
| **Transaction** |  |  |  |
| **Commit** |  |  |  |
| **Rollback** |  |  |  |
|  |  |  |  |
| **CreateTable** |  |  | table\_name |
| **DropTable** |  |  | table\_name |
|  |  |  |  |
| **Insert** |  |  | table\_name |
| **Delete** |  |  | table\_name |
| **Select** |  |  | table\_name |
| **Update** |  |  | table\_name |
|  |  |  |  |
| **Set** | col\_name |  |  |
|  |  |  |  |
| **AddCol** | col\_name | col\_type |  |
| **BeginPK** |  |  |  |
| **AddPK** | col\_name |  |  |
| **EndPK** |  |  |  |
|  |  |  |  |
| **BeginItem** |  |  |  |
| **AddItemCol** | col\_name | col\_type | value |
| **EndItem** |  |  |  |
|  |  |  |  |
| **BeginColSelect** |  |  |  |
| **AddColSelect** | col\_name |  |  |
| **EndColselect** |  |  |  |
|  |  |  |  |
| **BeginFilter** |  |  |  |
| **Operand** |  | col\_name | constant |
| **Operator** |  | op |  |
| **EndFilter** |  |  |  |
|  |  |  |  |
| **BeginJoin** |  |  |  |
| **AddTable** | table\_name |  |  |
| **EndJoin** | table\_name |  |  |
|  |  |  |  |
| **Execute** |  |  |  |
| **BeginExpression** |  |  |  |
| **EndExpression** |  |  |  |

上述变量中各变量取值情况如下：

1. constant、value、col\_name和table\_name均由数字、字母、下划线组成的标识符，并且第一个字符不能为数字，临时变量允许使用$作为首字符以区别于用户使用的标识符。

2. op仅能使用下列的字符串作为取值：GT、GE、LT、LE、EQ。

(原先的程序VirtualEnvironment模块也就是VirtualMachine.java文件存在大量错误，完全不能实现正常的功能，和原来ZunSQl设计书存在很大差距，对这个文件的大部分内容都做了修改，结构进行了调整，下面是修改命令识别的相关内容后的示例，还有一些具体内容没有写出)

[sql]

select a,b from x,y where z<10 and w>100

[bytecode]

Transaction null null null

Begin null null null

BeginJoin null null null（修改在读入这个操作时并不连接）

AddTable x null null

AddTable y null null

EndJoin null null null

Select null null null

BeginColSelect null null null

AddColSelect a null null

AddCOlSelect b null null

EndColSelect null null null

BeginFilter null null null

Operand z null null

Operand null 10 null

Operator LT null null

Operator And null null

Operand w null null

Operand null 100 null

Operator GT null null

Operator Or(修改后加上去，方便对where后条件的划分) null null

EndFilter null null null

Execute null null null

Commit null null null

[sql]

update t1 set a=100,b=100 where v=100 and w<9000

[bytecode]

Transaction null null null

Begin null null null

Update null null t1

Set a null null

BeginExpression null null null

Operand null 100 null

EndExpression null null null

Set b null null

BeginExpression null null null

Operand null 100 null

EndExpression null null null

BeginFilter null null null

Operand v null null

Operand null 100 null

Operator EQ null null

Operator And null null

Operand w null null

Operand null 9000 null

Operator LT null null

Operator Or(修改后加上去，方便对where后条件的划分) null null

EndFilter null null null

Execute null null null

Commit null null null

[sql]

delete from t1 where v=100 and w<9000

[bytecode]

Transaction null null null

Begin null null null

delete null null t1

BeginFilter null null null

Operand v null null

Operand null 100 null

Operator EQ null null

Operator And null null

Operand w null null

Operand null 9000 null

Operator LT null null

Operator Or(修改后加上去，方便对where后条件的划分) null null

EndFilter null null null

Execute null null null

Commit null null null

[sql]

create index in1 on t1(sname,course,age)

[bytecode]

Transaction null null null

Begin null null null

index in1 t1 null

Index\_Column sname null null

Index\_Column course null null

Index\_Column age null null

Execute null null null

Commit null null null

1. **CacheMgr模块**
2. **事务开始操作**

客户层B+树模块调用beginTransaction（String s）接口，并传入表示读操作(“r”)还是写操作(“w”)的字符串s。

服务层page模块接收到字符串s，直接调用Transaction部分的构造函数，并传入s和成员lock,来开启一个新事务并获得相应的锁。

Page模块以事务ID 作为键，将事务对象存储到自身的

transMgr并返回事务的ID。

1. **事务提交操作**

客户层B+树模块调用commitTranssation(int)接口并传入事务ID,

服务层Page模块根据事务ID从transMgr中获取对应的事务对象，并根据事务类型采取进一步的处理。

如果是读事务直接提交，并从transMgr中删除事务ID和对应的事务对象。

如果是写事务，根据事务ID 从tranOnPage中获取该事务所修改过的page对象,对每一个修改过的page对象，如果该page对象在cache中，按照LRU策略更新cacheList。如果该page对象不在cache中，如果cache已满，那么根据LRU策略从cache中替换出一页，否则直接从文件中读取该page，然后将page加入cacheList和cache中。将page对象写入journal日志文件，同时将该页写入数据库文件中。

如果函数调用成功返回true,否则引发IO Exception.

1. **读页操作（原先的不完整且没有两个cache之间的页满之后的调度）**

客户层Ｂ＋树模块调用readPage(int transID,int pageID)接口，传入事务ID和页ID。

如果该page在二级cache中，按照LRU策略更新cacheList。

如果该页不在二级cache中，如果cache已满，按照LRU策略替换出一页。从文件中读入该page对象，并加入cacheList和cache中。

调用成功，返回对应的page对象。修改后过程中同时使用了ConcurrentMap<Integer, Integer> cachePageMap加快判断速度。

1. **写页操作（原先的不完整且没有两个cache之间的页满之后的调度）**

客户层Ｂ＋树模块调用writePage(int transID,Page tempBuffer)接口，传入事务ID和修改过的page对象。

先将对应事务的写的页挂到一级cache的写页下，如果原来的一级cache已经有这一页要将原先的页删除，如果页数超过20页，将一级cache中的内容写入二级cache直到一级cache中只有15页为止，写出一级cache用LRU，在二级cache如果原来的页已经在二级cache中，用LRU将原来的页调整到末尾，如果二级cache超过20页则继续将最久没有使用的页写回硬盘。

1. **数据库关闭操作（为了能够重复打开同一个文件进行了修改）**

在关闭文件时，由于数据库的文件结构的需要，必须在文件头写入当前已经分配到的最大页号和master表B树根节点所在的的页号，在文件头的下一页写入可供使用的空闲页，并且空闲页按空闲页表进行组织，有特定的读写方法，在关闭文件的同时要把两级cache中的内容写回硬盘，顺序是先将二级cache中的内容写回硬盘，再将一级cache中的内容写回硬盘。

1. **isNew操作（为了能够重复打开同一个文件进行了修改）**

在创建数据库文件时，Database.java调用isNew来进行判断，如果这个文件已经存在，就打开这个文件，并读入这个文件的文件头中保存的已经分配的最大页号，master表的B树根节点页号，并将原来文件中的空闲页全部读出来放入到内存中，如果这个文件不存在，就创建新的文件并设置已经分配的最大页号为0，设置master表根节点页号为0，设置空闲页表的第一项为-1表示目前没有空闲页，并通过Database.java中的addmaster方法，创建一个空的master表的B树，且这个B树的根节点的页号为0，再向其中插入一项记录为master 0 -1，代表表名是master的表，表的B树根节点页号是0，目前没有建立索引。

1. **Transaction模块（原来的Transaction模块，并没有实现原来ZunSQl设计书所讲的功能，原来的程序中每次输入一条sql命令打开的都是写事务，却还用各种用户事务的调用来掩盖实际最终是写事务的单一情况，并且事务的主要作用是区分事务的事务号，用事务号来利用CacheMgr中的原本应该是一级cache的读写，每个sql命令都会换一个新的事务号，严重影响了一级cache的使用，导致一级cache失去了原有的意义，事务的其他作用没有得到体现，回滚和日志存在错误，根本无法执行原来的程序）**
2. **事务提供0号的事务号**

将所有事务的事务号设定为0号，保持原先事务的创建功能，但事务号固定成0号事务，方便进行一级cache的有关操作，节省内存。

1. **Page（对页号的增长进行修改，原先页号每次都会增加1，文件一打开就设置成0，完全忽视了被打开的文件已有的页的分配情况，将原来文件已经分配出去的页会再进行分配，也没有存储上一次文件打开已经分配到的最大页号，也没有存储由于删除操作多出来的空闲页，所已原先的程序无法实现重复打开相同文件的功能）**
2. **Page的构造方法：**

将申请到的内存传入，结合页号组成一个和硬盘中的页对应的实体页，和硬盘中的页联系主要取决于这个页的页号，在CacheMgr.java中的unusedList\_PageID保存了现有的空闲页，在增加新页前，先访问空闲页在有空闲页的情况下，得到一个空闲页的页号，如果没有空闲页，将已经分配到的最大页号增加1，得到新的页号。

1. **获取页号操作**

调用int getPageID()接口，返回对应页号

1. **获取page对象中存储的内容操作**

调用 ByteBuffer getPageBuffer()接口，获取page对象中存储的内容。存储内容是用ByteBuffer的对象来保存的。

1. **DBInstance模块（原来的程序不能实现多次打开同一个数据库文件的功能）**
2. **读入数据库名，并打开该数据库，如果文件已经存在就打开上次的数据库。**

用Database.java的构造方法，调用Database中的相应函数打开文件，对调用过程做了修改。

1. **对输入的SQL语句进行执行，得到返回结果集，以便应用程序后续的操作**

对原先的识别sql语句的模块做了相应修改，调整了原先有错误的地方，增加对创建索引的识别。

1. **在使用完数据库后，能够关闭数据库，断开连接。（调用CacheMgr中经过修改的close()方法）**
2. **Shell模块**

设计命令如下：

.open \*.db 打开一个数据库，如果不存在则新建一个数据库

.close \*.db 关闭一个数据库

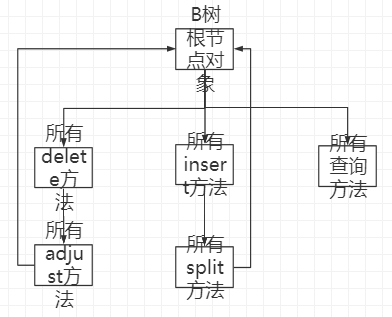
.help 显示提示信息

.dump 将数据库的信息从java序列化的二进制文件转成普通文件

1. **Node.java模块（所有内容都做了修改，没有用到原来程序的方法）**

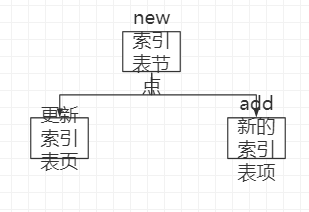
主要用于B树模块构建访问等操作，和硬盘中的B树模块的一个节点通过页号对应，但有类里面本身的结构，可以如果是在根节点的内部进行访问，可以实现不重新读入根节点就可以在根节点改变的情况下，继续使用原来旧的根节点进行增删改查的所有操作，主要依赖于Node中rhizine根节点成员变量的使用，但要求第一次访问的必须是根节点。

主要方法是实现有主键的查询，随着程序的推进持续更新B树根节点，维护B树的结构，主要利用了Cell中修改后的比较方法。具体的调用在第一部分有介绍，大概流程图如下：



1. **Node\_index.java模块（新创建的.java文件）**

实现对索引表进行组织的类，使得索引表的访问不用上层直接访问文件，通过页号和硬盘中索引表的一页对应，其中有add方法和更新方法，构造方法，提升索引表的访问安全性和便捷程度，具体访问流程如下：



1. **Node\_itree.java模块(新创建的.java文件)**

用来对表建立的不同的索引B树进行更加快速和方便的访问，便于各种B树操作对上层掩盖细节，使得底层对B树的维护可以独立于上层对B树的操作，让索引树的建立更加有效率，具体原理在第一部分有介绍，模块中的比较操作，由于有多个索引列需要进行更改，具体如下：

1. public boolean bigerThan(Row r1,Row r2)如果行r1对应索引列号的值大于行r2则返回真，否则返回假，使用时，r1和r2的具体索引所在的列号已经先计算好，且比较操作按照索引列的优先次序对多个索引列的情况进行比较；
2. public boolean letterThan(Row r1,Row r2)如果行r1对应索引列号的值小于行r2则返回真，否则返回假，使用时，r1和r2的具体索引所在的列号已经先计算好，且比较操作按照索引列的优先次序对多个索引列的情况进行比较；
3. public boolean equalTo(Row r1,Row r2) 如果行r1对应索引列号的值等于行r2则返回真，否则返回假，使用时，r1和r2的具体索引所在的列号已经先计算好，且比较操作按照索引列的优先次序对多个索引列的情况进行比较；
4. **具体的类方法列表**
5. **SQL解析类Parser**

SQL语句解析类Parser相关类一览表

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **包名** | **类/接口名** | **说明** |
| 1 | npu.zunsql.sqlparserparser.ast | Assignment | 赋值语句表达式 |
| 2 | Begin | 开始事务语句 |
| 3 | BinaryExpression | 二元表达式 |
| 4 | Column | 创建语句中表的行 |
| 5 | Commit | 提交事务语句 |
| 6 | Create | 创建表语句 |
| 7 | DataType | 枚举类，支持的数据类型 |
| 8 | Delete | 删除语句 |
| 9 | Drop | 删表语句 |
| 10 | Expression | 表达式公共接口 |
| 11 | Insert | 插入语句 |
| 12 | NumberExpression | 数字字面值表达式 |
| 13 | Op | 枚举类，支持的运算符 |
| 14 | QualifiedName | 名字，包括列名等 |
| 15 | QualifiedNameExpression | 名字表达式 |
| 16 | Relation | 关系语句公共接口 |
| 17 | Rollback | 回滚事务语句 |
| 18 | Select | Select 语句 |
| 19 | StringExpression | 字符串字面值表达式 |
| 20 | TableRelation | 表名 |
| 21 | UnaryExpression | 一元运算表达式 |
| 22 | Update | 更新语句 |
| 23 | WildcardExpression | 星号表达式 |
| 24 | npu.zunsql.sqlparser.parser.parser | ExpressionParser | 表达式解析器实现类 |
| 25 | RelationParser | 关系语句解析器实现类 |
| 26 | TerminalParser | 终止符解析器实现类 |
| 27 | npu.zunsql.sqlparser.parser | Parser | 提供给用户的类 |

各类关系：

所有 Expression 结尾的类和 Assignment 类均实现了 Expression 接口。

Begin Commit Create Delete Drop Insert Rollback Select Update 类实现了 Relation 接口。

主要类详细设计：

各语法树节点类没有提供特殊的方法，基本只会用到自定义的构造函数。各Parser 类中的方法为各种语法结构的解析器，从 TerminalParser -> ExpressionParser-> RelationParser 逐渐复杂，后者使用前者提供的较为简单的解析器组合成更为复杂的解析器。详见源码与 SQL 语法规范。

1. **字节码生成类 CodeGenerator**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **包名** | **类/接口名** | **说明** |
| 1 | npu.zunsql.codegen | CodeGenerator | 中间码生成的实现类 |

CodeGenerator类方法一览

类详细设计：

CodeGenerator 在遍历关系语句语法树节点的过程中，会对各节点做断言，根据具体语句类型进行生成。CodeGenerator 的 GenerateByteCode 方法接受列表而不是单个节点的原因是，转译过程是上下文相关的，在转译时需要知道此语句是否处在事务上下文中。

CodeGenerator提供以下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | GenerateByteCode(List<Relation> statements) | 生成字节码 |
| 2 | private | ExpressionToInstruction(Expression exp) | 表达式生成到表达式上下文的字节码 |
| 3 | private | WhereToInstruction(Expression where) | 表达式生成到过滤器上下文的字节码 |
| 4 | public | ExpressionToInstructionsInternal(Expression expr) | 无上下文的表达式生成到字节码，上面两者皆调用此方法。public 仅出于测试目的 |
| 5 | private | OpToInstruction(Op op) | 运算符转换为字节码 |
| 6 | private | OpToString(Op op) | 运算符转换为字符串 |
| 7 | private | DataTypeToString(DataType dt) | 数据类型转换为字符串 |

1. **VirtualEnvironment 模块（进行了很多修改）**
2. **Activity**

枚举类型，用于描述本条指令所要执行的“动作”，内部处理机制。提供以下方法:

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public |  | 描述不同指令的“动作” |

1. **AttInstance**

用于描述插入数据信息的类，内部机制，提供以下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | AttrInstance(String pAttrName,String pType,String pValue) | 构造方法（名称，类型和值） |
| 2 | public | String getAttrName() | Getter |
| 3 | public | String getValue() | Setter |

1. **ByteCode**

描述“字节码”的类，包括一个枚举类型的操作码和单三个字符串类型的额操作数，提供给上层用于构造指令，只提供一个构造方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | ByteCode(OpCode opCode, String p1, String p2, String p3) | 构造器（操作码和三个操作数） |

1. **Column**

用于描述表中某一列的类，即表头信息，内部处理机制，提供以下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | Column(String pName,String pType) | 根据名称和类型构造 |
| 2 | public | Column(String pName) | 根据名称构造 |
| 3 | public | boolean equals(Object obj) | 重写equals方法 |
| 4 | public | String getColumnName() | 取得列名 |
| 5 | public | String getColumnType() | 取得列类型 |

1. **EvalDiscription**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | EvaliDiscription(OpCode pCmd, String pColName, String pConstant) | 构造器 |

1. **Expression**

负责解析并计算表达式的类，与UnionOperand联系密切，内部处理机制，提供以下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | Expression() | 构造器 |
| 2 | public | addOperand(UnionOperand obj) | 添加操作数 |
| 3 | public | applyOperator(OpCode op) | 开始计算表达式 |
| 4 | public | UnionOperand getAns() | 获取计算结果 |

1. **VirtualMachine**

负责整合所有数据处理功能并执行指令的类，上层直接调用run方法进行即可执行指令，提供以下方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | VirtualMachine(Database pdb) | 构造器，指定数据库 |
| 2 | public | run(ByteCode instruction) throws IOException | 执行一条指令 |
| 3 | private | execute() | 预处理完成后，开始执行包括4-107种类型。 |
| 4 | private | createTable() |  |
| 5 | private | select() |  |
| 6 | private | delete() |  |
| 7 | private | update() |  |
| 8 | private | insert() |  |
| 9 | private | ConditonClear() |  |
| 10 | public | create\_Index() |  |
| 11 | public | bigerThan(String s1,String s2) |  |
| 12 | public | letterThan(String s1,String s2) |  |
| 13 | public | equalTo(String s1,String s2) |  |
| 14 | public | select\_auto(List<Node> root\_node\_table,List<List<Node\_itree>> nit3) |  |
| 15 | public | delete\_auto(Node root\_node\_table,List<Node\_itree> nit2) |  |
| 16 | public | delete\_update() |  |
| 17 | public | delete\_auto\_update(Node root\_node\_table,List<Node\_itree> nit2) |  |

1. **Btree模块(原来没有实现，用了其他方式来掩盖，并且有很多错误，全都进行了重新的编写)**
2. **Cell**

用来一个记录的一个元组的内容

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | bigerThan(Cell cell1) | 当前元组内容比参数元组内容大返回真否则返回假如果是数字按数字来比较 |
| 2 | public | letterThan(Cell cell1) | 当前元组内容比参数元组内容小返回真否则返回假如果是数字按数字来比较 |
| 3 | public | equalTo(Cell cell) | 当前元组的内容和参数元组的内容一样大返回真否则返回假 |
| 4 | public | getValue\_s() | 返回当前元组的值 |
| 5 | public | Cell(String givenValue) | 构造方法 |

1. **Database**

创建删除处理数据库，打开数据库，在没有master表时，添加master表

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | private | addMaster(Transaction initTran) | 创建master表B树根节点 |
| 2 | public | colse() | 调用CacheMgr.java中的close()方法关闭当前数据库 |
| 3 | public | createTable(String tableName, String keyName, List<String> columnNameList,List<BasicType> tList, Transaction thisTran) | 为当前数据库创建一个表 |
| 4 | public | Database(String name, int M) | 打开数据库 |
| 5 | public | createTable(String tableName, String keyName, List<String> columnNameList, List<BasicType> tList,  Transaction thisTran, int M) | 只由addMaster方法在创建master表B树根节点后加入master 0 -1记录调用 |

1. **Node**

用作访问硬盘操作B树的接口，方便进行各种B树操作，提高B树编写的效率，降低模块之间的耦合程度，并通过页号和硬盘的具体B树节点建立联系

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | Node(int thisPageID, CacheMgr cacheManager, Transaction thisTran) | 构造方法获得一个已存在的B树节点 |
| 2 | public | Node(int order2,String tablename2,String keyname2,List<String>list1,List<String>list2,CacheMgr cacheManager, Transaction thisTran) | 构造方法创建一个新的B树节点 |
| 3 | public | intoBytes (Transaction thisTran) | 更新B树节点的值到硬盘 |
| 4 | public | update\_root(Integer page,String name,Transaction thisTran) | 更新B树根节点 |
| 5 | public | split(Node root,Transaction thisTran) | 对插入后行数大于阶数的节点进行分裂 |
| 6 | public | insertRow(Row row,Transaction thisTran) | 插入一个新行 |
| 7 | public | get\_brother\_Node(Node root,Transaction thisTran) | 获得当前节点的阶数大于M/2-1的左兄弟或右兄弟 |
| 8 | public | adjust(Node root,Transaction thisTran) | 对删除行后阶数小于M/2的节点进行调整 |
| 9 | public | deleteRow(Cell key,int liehao,Transaction thisTran) | 删除一行 |
| 10 | public | search\_auto(int keyp,Cell key,Node root,Transaction thisTran) | 只找一个相等节点 |
| 11 | public | search\_all(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索所有主键元组值是key的B树中的行 |
| 12 | public | delete\_all(Node root, Cell key, Integer keyp,Transaction thisTran) | 删除所有主键元组值是key的B树中的行 |
| 13 | public | count\_key\_number(Node root, Cell key, Integer keyp, Transaction thisTran) | 统计以root为根的B树中主键元组值是key元组值得行的个数 |
| 14 | public | get\_node\_all(Node root, Transaction thisTran) | 获得所有以root为根的B树的行 |
| 15 | public | search\_greater\_all(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值大于key元组值的行 |
| 16 | public | search\_letter\_all(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值小于key元组值的行 |
| 17 | public | search\_equal\_all(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值等于key元组值的行 |
| 18 | public | search\_greater\_or\_equal\_all(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值大于等于key元组值的行 |
| 19 | public | search\_letter\_or\_equal\_all(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值小于等于key元组值的行 |
| 20 | public | search\_greater\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值大于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 21 | public | search\_letter\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值小于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 22 | public | search\_equal\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值等于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 23 | public | search\_greater\_or\_equal\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值大于等于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 24 | public | search\_letter\_or\_equal\_all\_nokey(Node root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值小于等于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 25 | public | get\_brother\_Node\_nokey(Node root,Transaction thisTran) | 获得当前节点的阶数大于M/2-1的左兄弟或右兄弟 |
| 26 | public | adjust\_nokey(Node root,Transaction thisTran) | 对删除行后阶数小于M/2的节点进行调整 |
| 27 | public | deleteRow\_nokey(Node root,Integer p1\_position\_arg,Transaction thisTran) | 删除指定根节点处的第p1\_position\_arg行 |
| 28 | public | deleteRow\_nokey\_pre(Node root, List<Cell> keys, List<Integer> keyps, List<String> condition, Transaction thisTran) | 删除以root为根的B树对应位置元组值为keys相应条件是condition的行的提前处理 |
| 29 | public | count\_key\_number\_nokey(Node root, List<Cell> keys, List<Integer> keyps, List<String> condition,Transaction thisTran) | 统计以root为根的B树对应位置元组值为keys相应条件是condition的行的个数 |
| 30 | public | delete\_all\_nokey(Node root, List<Cell> keys, List<Integer> keyps,List<String> condition,Transaction thisTran) | 删除所有以root为根的B树对应位置元组值为keys相应条件是condition的行 |
| 31 | public | count\_key\_number\_nokey2(Node root, List<Cell> keys, List<Integer> keyps, List<String> condition,Transaction thisTran) | 在更新之前统计所有以root为根的对应位置元组值为keys相应条件是condition的行的个数为插入修改后的行做准备 |
| 32 | public | delete\_all\_nokey2(Node root, List<Cell> keys, List<Integer> keyps, List<String> condition,Transaction thisTran) | 用在更新操作先进行的删除操作 |
| 33 | public | delete\_zheng\_ge\_shu\_bao\_cun\_geng\_ye(Node root,Transaction thisTran) | 删除以root为根的B树的所有行，只保留B树树根 |

1. **Node\_index**

作为管理索引页的接口，以页号和硬盘中的索引页联系

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | Node\_index(int thisPageID, CacheMgr cacheManager, Transaction thisTran) | 构造方法访问一个已经存在的索引页 |
| 2 | public | intoBytes (Transaction thisTran) | 把索引页的内容更新到硬盘 |
| 3 | public | add(int thisPageID, String indexname2,List<String> keynames2,CacheMgr cacheManager,Transaction thisTran) | 在索引页中新加一项索引 |

1. **Node\_itree**

用作管理建立的B树索引的接口，以页号和硬盘中具体的索引联系

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | Node(int thisPageID, CacheMgr cacheManager, Transaction thisTran) | 构造方法获得一个已存在的B树节点 |
| 2 | public | Node\_itree(int order2,String tablename2,String indexname2,List<String> keyname2,List<String>list1,CacheMgr cacheManager, Transaction thisTran) | 构造方法创建一个新的B树节点 |
| 3 | public | intoBytes (Transaction thisTran) | 更新B树节点的值到硬盘 |
| 4 | public | update\_root(Integer page,String name,Transaction thisTran) | 更新B树根节点 |
| 5 | public | split(Node\_itree root,Transaction thisTran) | 对插入后行数大于阶数的节点进行分裂 |
| 6 | public | insertRow(Row row,Transaction thisTran) | 插入一个新行 |
| 7 | public | get\_brother\_Node(Node\_itree root,Transaction thisTran) | 获得当前节点的阶数大于M/2-1的左兄弟或右兄弟 |
| 8 | public | adjust(Node\_itree root,Transaction thisTran) | 对删除行后阶数小于M/2的节点进行调整 |
| 9 | public | deleteRow(Cell key,int liehao,Transaction thisTran) | 删除一行 |
| 10 | public | search\_auto(int keyp,Cell key,Node \_itree root,Transaction thisTran) | 只找一个相等节点 |
| 11 | public | search\_all(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索所有主键元组值是key的B树中的行 |
| 12 | public | delete\_all(Node\_itree root, Cell key, Integer keyp,Transaction thisTran) | 删除所有主键元组值是key的B树中的行 |
| 13 | public | count\_key\_number(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 统计以root为根的B树中主键元组值是key元组值得行的个数 |
| 14 | public | get\_node\_all(Node\_itree root, Transaction thisTran) | 获得所有以root为根的B树的行 |
| 15 | public | search\_greater\_all(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值大于key元组值的行 |
| 16 | public | search\_letter\_all(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值小于key元组值的行 |
| 17 | public | search\_equal\_all(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值等于key元组值的行 |
| 18 | public | search\_greater\_or\_equal\_all(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值大于等于key元组值的行 |
| 19 | public | search\_letter\_or\_equal\_all(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有主键元组值小于等于key元组值的行 |
| 20 | public | search\_greater\_all\_nokey(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值大于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 21 | public | search\_letter\_all\_nokey(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值小于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 22 | public | search\_equal\_all\_nokey(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值等于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 23 | public | search\_greater\_or\_equal\_all\_nokey(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值大于等于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 24 | public | search\_letter\_or\_equal\_all\_nokey(Node\_itree root, Cell key, Integer keyp, Transaction thisTran) | 搜索以root为根的所有对应元组值小于等于key元组值的行(不是主键不能用B树的结构来加速搜索) |
| 25 | public | get\_brother\_Node\_nokey(Node\_itree root,Transaction thisTran) | 获得当前节点的阶数大于M/2-1的左兄弟或右兄弟 |
| 26 | public | adjust\_nokey(Node\_itree root,Transaction thisTran) | 对删除行后阶数小于M/2的节点进行调整 |
| 27 | public | deleteRow\_nokey(Node\_itree root,Integer p1\_position\_arg,Transaction thisTran) | 删除指定根节点处的第p1\_position\_arg行 |
| 28 | public | deleteRow\_nokey\_pre(Node\_itree root, List<Cell> keys, List<Integer> keyps, List<String> condition, Transaction thisTran) | 删除以root为根的B树对应位置元组值为keys相应条件是condition的行的提前处理 |
| 29 | public | count\_key\_number\_nokey(Node\_itree root, List<Cell> keys, List<Integer> keyps, List<String> condition,Transaction thisTran) | 统计以root为根的B树对应位置元组值为keys相应条件是condition的行的个数 |
| 30 | public | delete\_all\_nokey(Node\_itree root, List<Cell> keys, List<Integer> keyps,List<String> condition,Transaction thisTran) | 删除所有以root为根的B树对应位置元组值为keys相应条件是condition的行 |
| 31 | public | count\_key\_number\_nokey2(Node\_itree root, List<Cell> keys, List<Integer> keyps, List<String> condition,Transaction thisTran) | 在更新之前统计所有以root为根的对应位置元组值为keys相应条件是condition的行的个数为插入修改后的行做准备 |
| 32 | public | delete\_all\_nokey2(Node\_itree root, List<Cell> keys, List<Integer> keyps, List<String> condition,Transaction thisTran) | 用在更新操作先进行的删除操作 |
| 33 | public | delete\_zheng\_ge\_shu\_bao\_cun\_geng\_ye(Node\_itree root,Transaction thisTran) | 删除以root为根的B树的所有行，只保留B树树根 |
| 34 | public | bigerThan(Row r1,Row r2) | 比较行r1是否在索引列大于行r2多个索引列的情况下，按优先级依次比较 |
| 35 | public | letterThan(Row r1,Row r2) | 比较行r1是否在索引列小于行r2多个索引列的情况下，按优先级依次比较 |
| 36 | public | equalTo(Row r1,Row r2) | 比较行r1是否在索引列等于行r2多个索引列的情况下，按优先级依次比较 |

1. **Row**

行类型，用以存储表一行的数据

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | protected | Row(List<String> SList) | 行列表 |
| 2 | protected | getStringList() | 获取行的值 |
| 3 | protected | getCell(int array) | 获取行里指定Cell存储的值 |

1. **Transaction**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | Commit() | 提交当前事务 |
| 2 | public | RollBack() | 回写当前事务 |

1. **Cache中类一览及关系图**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **包名** | **类/接口名** | **说明** |
| 1 | npu.zunsql.cache | CacheMgr | 内存和文件管理类 |
| 2 | Page | 页管理类 |
| 3 | Transaction | 事务管理类 |

1. **CacheMgr**

具体管理两级cache，管理空闲页，管理文件打开创建写回

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | boolean isNew() | 如果对应数据库名的文件已经创建打开后继续上次的文件，否则创建新文件 |
| 2 | public | void close() | 关闭数据库文件，保存已分配的最大页号，master表B树根节点页号，将cache内容写回硬盘 |
| 6 | public | Page readPage(int transID, int pageID) | 读取对应页号的页 |
| 7 | public | boolean writePage(int transID, Page tempBuffer) | 写回对应页号的页 |
| 9 | private | Page getPageFromFile(int pageID) | 直接从硬盘读取对应页号的页 |

1. **Page**

管理页号分配，用页号和硬盘中的实际页联系

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **可见性** | **方法名** | **说明** |
| 1 | public | getPageID() | 获取当前页的页号 |
| 2 | public | getPageBuffer() | 获取当前页在内存中的内容 |
| 3 | public | Page(ByteBuffer buffer) | 构造方法生成新页同时申请一个页号，这个页号可能是未用页中的一个页号 |

1. **经验总结和规模**
2. **经验总结**

通过这次实验，更加深刻地掌握了数据库索引的相关概念和内容，对B树的构造和修改有了更加深入的理解，尤其对java的程序编写有了进一步的熟悉，能够运用简单的List，Map，Stack等数据结构来辅助程序编写，对java序列化的概念有了一定的认识，对java文件交互和内存间的缓冲交互有了一定的熟悉，会用system.out和system.in和scanner类来进行控制台的各种交互。

对java继承，封装，多态和抽象类的概念有了相应的接触，对程序间的高内聚低耦合和分模块编写有了非常深刻的映象，深刻感受到模块间独立的好处，也感受到写耦合较高程序时程序修改和调试的困难，往往修改一处后要修改大量的地方，同时又很难逐一修改，调试又面临着很多数据，很难再现出错的情况，只能逐行看原来的程序，耽误了很多时间，也降低了程序的可读性和可扩展性。

也初步认识了面向对象编程，对一些类的构造和定义有了一些尝试，初步感受到对象之间交互的深刻和困难，使得原来完全不用面向对象编写程序的情况彻底发生了改变，获得了一次能够近距离接触面向对象编程的宝贵机会。

对程序的调试和相关问题有了一些新的运用，初步认识到调试程序在程序编写中所占的重要地位，尽管只是非常浅显地进行了各种类的交互，但调试的复杂程度和困难程度远远超过想象，很难复现出现问题的情况，而出错的情况在很多想不到的地方发生，在不断增加规模的时候，错误发掘的困难几乎是膨胀一样的增长，完全不在可控的范围内，甚至会认为重头再来反而更合理。

1. **规模**

实验中全部程序编写，修改，查阅，跟踪，错误的程序删除再重新编写总和预计超过很多千行以上，展现出来的完全独立原来程序的程序约有4000行，主要集中（但不止在这些地方）在App.java(在提交的文件中只保留了一些测试用例，完整的测试用例也有很多)，Node.java，Node\_itree.java，Node\_index.java，CacheMgr.java，page.java，Cell.java，Database.java，DBInstance.java，VirtualMachine.java，Shell.java等，为了验证一些java使用的规范，进行的测试大约有800行。