

























# SimpleDB Lab2 实验报告

## commit记录

17 Apr, 2022 1 commit		
 Lab2_Finish Test TTATT authored 6 minutes ago	0148dbf8	 
14 Apr, 2022 2 commits		
 Lab2_exercise4_Finish Test TTATT authored 2 days ago	54e5bdc4	 
 Lab2_exercise3_Finish Test TTATT authored 2 days ago	9602b17c	 
13 Apr, 2022 2 commits		
 Lab2_exercise3_Finish HeapPage TTATT authored 3 days ago	98b5f951	 
 Lab2_exercise2_Finish Test TTATT authored 3 days ago	1f39d299	 
12 Apr, 2022 1 commit		
 Lab2_exercise2_Finish IntegerAggregator TTATT authored 4 days ago	4d665028	 
07 Apr, 2022 1 commit		
 Lab2_exercise1_Finish Test TTATT authored 1 week ago	971cdbc7	 
01 Apr, 2022 1 commit		
 Lab2_exercise1 TTATT authored 2 weeks ago	014a6723	 

## 实验设计思路

### exercise1

在exercise1中要求实现 `Predicate.java` `JoinPredicate.java` `Filter.java` `Join.java` 这四个类，通过这四个类，就能够实现满足谓词要求的元组的JOIN运算符操作。

在 `Predicate.java` 这个类中，规定了一些谓词，这个类的主要功能就是比较给定的元组在指定的域上是否满足谓词的要求。在这里也是通过 `filter()` 方法来实现的，在方法内，只需要调用 `Field` 类中提供的 `compare()` 方法，传入谓词运算符和指定的域得到一个是符合要求的boolean类型的返回结果，将这个结果作为返回值返回。

在 `JoinPredicate.java` 这个类中，主要实现的在进行JOIN时，对于两个元组给定的两个域上，进行谓词运算，判断其是否满足要求。在这个类中主要需要实现的方法是 `filter()`，实现的方法也是在给定的两个元组上，去获取到指定的编号的域，然后在这两个域上，去调用 `Field` 类中提供的 `compare` 方法得到一个返回值最为结果返回。

在 `Filter.java` 这个类中，主要实现了对于给定的一些列元组，通过谓词对其进行过滤，得到一个满足为此要求的元组的迭代器。在这个类中，核心方法是 `fetchNext()`，这个方法用来获取一个给定的元组序列中，满足谓词要求的元组。在这个方法中，需要判断用来遍历给定元组序列的迭代器中是否还有下一个元素，如果有下一个元素则判断下一个元素是否满足要求，若满足要求，则返回这个元素，如果不满足要求，则继续向下迭代，直到迭代器中不再有元素存在。

在 `Join.java` 这个类中，要求实现一个能够遍历获取两个元组序列中满足为此需求，并通过 JOIN 运算连接起来的一个元组。其核心方法是要实现 `fetchNext()` 方法，这个方法就是获取到满足谓词需求的通过 JOIN 运算拼接起来的元组。大致思路就是采用双重循环，外层循环遍历 `child1`，内层循环遍历 `child2`，分别将得到的两个元组判断其是否满足 `JoinPredicate` 谓词的要求，通过 `filter()` 方法对其进行过滤，将满足谓词需求的两个元组通过 `merge` 方法进行合并，再将这个合并得到的结果返回。

✓	○	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	Q	»
✓	✓	PredicateTest (simplifiedb)							14毫秒
	✓	filter							14毫秒

✓	○	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	Q	»
✓	✓	JoinPredicateTest (simplifiedb)							12毫秒
	✓	filterVaryingVals							12毫秒

✓	○	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	Q	»
✓	✓	FilterTest (simplifiedb)							11毫秒
	✓	rewind							11毫秒
	✓	filterEqualNoTuples							0毫秒
	✓	filterAllLessThan							0毫秒
	✓	filterSomeLessThan							0毫秒
	✓	getTupleDesc							0毫秒
	✓	filterEqual							0毫秒

✓	○	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	Q	»
✓	✓	JoinTest (simplifiedb)							13毫秒
	✓	rewind							11毫秒
	✓	gtJoin							1毫秒
	✓	eqJoin							0毫秒
	✓	getTupleDesc							1毫秒

✓	○	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	Q	»
✓	✓	FilterTest (simplifiedb.systemtest)							578毫秒
	✓	testGreaterThanOrEq							531毫秒
	✓	testLessThanOrEq							14毫秒
	✓	testEquals							12毫秒
	✓	testGreaterThan							10毫秒
	✓	testLessThan							11毫秒

✓

⊘

↓<sup>a</sup>

↓<sub>z</sub>

≡

÷

↑

↓

🔍

»

▼

✓

JoinTest (simplifiedb.systemtest)

463毫秒

✓

testNoMatch

454毫秒

✓

testSingleMatch

4毫秒

✓

testMultipleMatch

5毫秒

## exercise2

在 exercise2 中要求实现 `IntegerAggregator.java` `StringAggregator.java` `Aggregate.java` 这三个类，前两个是对 `Aggregator` 的辅助工具类，在实现完前两个类之后，就可以在 `Aggregate` 类中直接调用前两个类中提供的方法。

`IntegerAggregator.java` 和 `StringAggregator.java` 两个类的实现原理大致相同，以 `IntegerAggregator.java` 类为例介绍设计思路即可。

类中的属性主要有如下

IntegerAggregator	
groupCount	HashMap<Field, Integer>
groupResult	HashMap<Field, Integer>
serialVersionUID	long
groupByFieldType	Type
aggregateField	int
operator	Op
groupByField	int

通过两个hashmap来记录在不同分组中的聚合信息，而count主要是用来辅助计算平均值而单独设立的。

这个类中的核心方法是 `mergeTupleIntoGroup`，即给定一个元组，将其并入到当前的分组状态中去。由于有不同的聚合运算，因此要通过分支进行控制，对于每种聚合运算进行单独的处理即可。整体的思想就是通过一个hashmap实现分组的需求，haspmap的key即是来确定每组的信息，而value则用来记录聚合的结果。对于计算平均值的方法则需要单独记录一下各个分组中元组的个数。

另一个核心的方法是 `iterator`，这个方法用来获得一个分组聚合结果的迭代器。要注意区分是否有分组的需求，因为这两种需求返回的迭代器的内容是不一样的需要单独拆分考虑。两种情况内的基本思路是一致的，都是返回一个匿名内部类产生的对象。在内部类中需要获取到分组hasmap的迭代器，用来获取聚合的结果。

在 `Aggregate.java` 中，实现了聚合运算的基本操作。在这个类中，核心方法是 `generate()`，在这个方法中我们需要判断一下进行分组的域的类型是什么，通过这个类型去进行分支，选择是 `INT_TYPE` 还是 `STRING_TYPE`。然后声明该类型对应的聚合辅助工具，并以此将当前数据的元组序列中的元组merge进去，最后获得结果的迭代器，就完成了整个工作。

✓	⊘	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	🔍	»
✓	IntegerAggregatorTest (simpledb)	13毫秒							
✓	mergeMax	13毫秒							
✓	mergeAvg	0毫秒							
✓	mergeSum	0毫秒							
✓	mergeMin	0毫秒							
✓	testIterator	0毫秒							

✓	⊘	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	🔍	»
✓	StringAggregatorTest (simpledb)	12毫秒							
✓	testIterator	11毫秒							
✓	mergeCount	1毫秒							

✓	⊘	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	🔍	»
✓	AggregateTest (simpledb)	18毫秒							
✓	rewind	13毫秒							
✓	countStringAggregate	1毫秒							
✓	minAggregate	1毫秒							
✓	sumStringGroupBy	1毫秒							
✓	getTupleDesc	1毫秒							
✓	avgAggregate	0毫秒							
✓	sumAggregate	0毫秒							
✓	maxAggregate	1毫秒							

✓	⊘	↓ <sub>a</sub>	↓ <sub>f</sub>	≡	÷	↑	↓	🔍	»
✓	AggregateTest (simpledb.systemtes	570毫秒							
✓	testCount	535毫秒							
✓	testMax	9毫秒							
✓	testSum	7毫秒							
✓	testAverage	6毫秒							
✓	testMin	7毫秒							
✓	testAverageNoGroup	6毫秒							

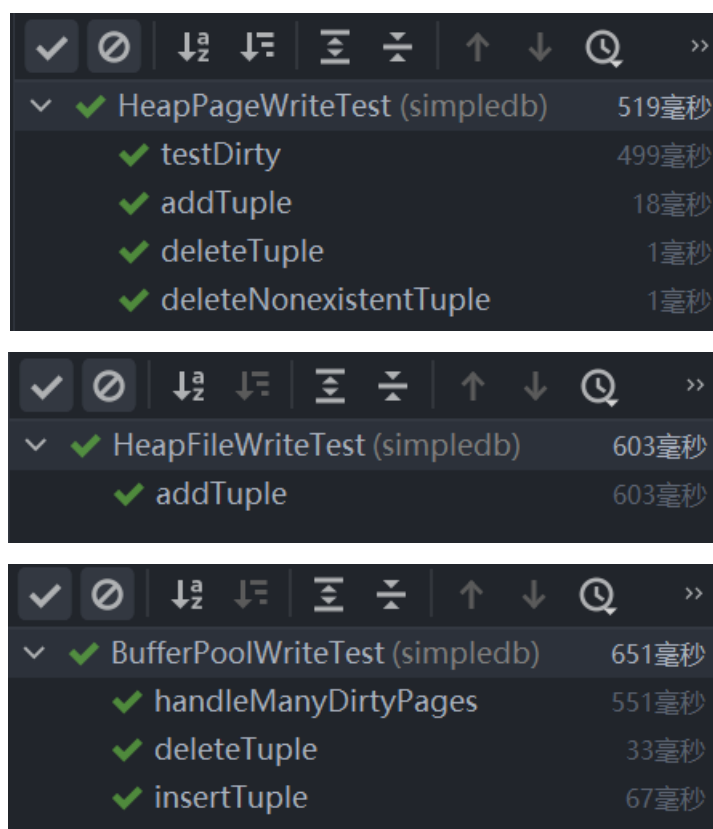
## exercise3

在exercise3中，要求实现 `HeapPage.java` 和 `HeapFile.java` 这两个类。主要的功能是在文件级去删除或者插入元组。

在 `HeapPage.java` 中，主要实现的方法是 `insertTuple()` 和 `deleteTuple()`。在插入元组的时候，要首先判断一下是否有空余的位置，如果不存在空余位置则需要抛出异常，存在空余位置则按顺序遍历header找到空余位置对应的标号，然后将元组插入到该位置处。在删除元组的时候，也需要首先判断一下当前的page是否是空的，如果是空的则需要抛出异常，如果可能存在这个元组，则顺序遍历去寻找，如果没有找到则还是需要抛出异常。删除和插入元组都需要更新header。

在 `HeapFile.java` 中，主要实现的方法是 `insertTuple()` 和 `deleteTuple()`，此外还需要实现一个将文件写入硬盘的方法 `writePage()`。在插入元组的时候，首先在该文件的每一页去查找是否能够插入元组，如果有page存在空余位置，则将这个元组插入在这一页中，然后直接返回插入元组的这个page。如果所有的page都没有空余位置，则需要创建一个新的page，将这个元组插入到这个新的page中，并将这个新的page写入到磁盘中。此外，插入删除元组的时候，都需要将执行该操作的page标记成dirty，方便后续在缓冲池中逐出策略的判断。在执行 `insertTuple()` 和 `deleteTuple()` 时，要注意读取page需要通过缓冲池，这样才能够将页面装入缓冲池当中。

此外还需要实现 `BufferPool.java` 中的 `insertTuple()` 和 `deleteTuple()` 方法。在插入元组的时候，需要根据给出的tableId定位到文件，然后对指定的文件执行插入操作，将返回的page加到缓冲池当中，并将该文件标记为dirty。在删除元组的时候，同样需要根据给定元组来定位他所属的文件，对该文件执行删除元组操作，并将该文件加入到缓冲池中，同时标记为dirty。



✓	⊘	↓ <sup>a</sup>	↓ <sub>z</sub>	≡	÷	↑	↓	🔍	»
✓	✓	HeapPageWriteTest (simplifiedb)							519毫秒
✓	✓	testDirty							499毫秒
✓	✓	addTuple							18毫秒
✓	✓	deleteTuple							1毫秒
✓	✓	deleteNonexistentTuple							1毫秒

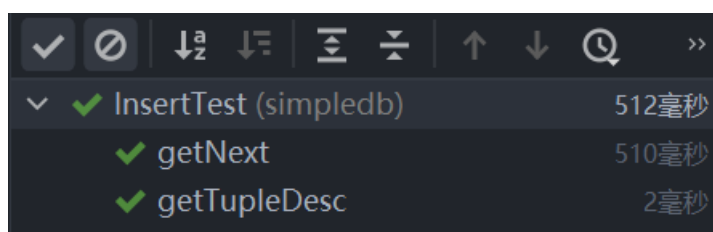
✓	⊘	↓ <sup>a</sup>	↓ <sub>z</sub>	≡	÷	↑	↓	🔍	»
✓	✓	HeapFileWriteTest (simplifiedb)							603毫秒
✓	✓	addTuple							603毫秒

✓	⊘	↓ <sup>a</sup>	↓ <sub>z</sub>	≡	÷	↑	↓	🔍	»
✓	✓	BufferPoolWriteTest (simplifiedb)							651毫秒
✓	✓	handleManyDirtyPages							551毫秒
✓	✓	deleteTuple							33毫秒
✓	✓	insertTuple							67毫秒

## exercise4

在 exercise4 中要求实现 `Insert.java` 和 `Delete.java` 这两个类，以实现插入和删除这两个运算符操作。在 `Insert.java` 中的核心方法是 `fetchNext()`，首先需要判断这个方法是否被调用过，引入了一个boolean类型的flag。一旦调用过 `close()` 方法，就证明已经执行过插入操作了，这个变量就会被置位。在 `fetchNext()` 方法中，迭代元组序列中的元组，将其一次插入的对应的表中。在 `Delete.java` 中要实现的方法与之类似，不再赘述。



✓	⊘	↓ <sup>a</sup>	↓ <sub>z</sub>	≡	÷	↑	↓	🔍	»
✓	✓	InsertTest (simplifiedb)							512毫秒
✓	✓	getNext							510毫秒
✓	✓	getTupleDesc							2毫秒

✓	○	↓↑	↓↑	≡	÷	↑	↓	🔍	»
✓	✓	InsertTest (simplifiedb.systemtest)							517毫秒
✓		testEmptyToOne							505毫秒
✓		testOneToOne							5毫秒
✓		testOneToEmpty							3毫秒
✓		testEmptyToEmpty							4毫秒

✓	○	↓↑	↓↑	≡	÷	↑	↓	🔍	»
✓	✓	DeleteTest (simplifiedb.systemtest)							552毫秒
✓		testLessThanOrEq							495毫秒
✓		testGreaterThan							17毫秒
✓		testLessThan							14毫秒
✓		testEquals							13毫秒
✓		testGreaterThanOrEq							13毫秒

## exercise5

在 exercise5 中要求实现 `BufferPool.java` 这个类，主要来实现当缓冲池已经饱和后的页面逐出策略，即完善方法 `evictPage()`。在页面逐出策略中，本次采用的是逐出当前缓冲池内，所有违背标记为dirty的页面，这还是因为这些页面暂时没有被修改过，短时间内应该不会访问。

还需要实现 `flushPage()` 这个方法，即通过一次刷新操作，将缓冲池内被标记为dirty的页面刷新为不被标记为dirty。

✓	○	↓↑	↓↑	≡	÷	↑	↓	🔍	»
✓	✓	EvictionTest (simplifiedb.systemtest)							1秒 221毫秒
✓		testHeapFileScanWithManyPages							1秒 221毫秒

## 重难点

### exercise1

在 `Join.java` 中，要实现一个能够遍历获取两个元组序列中满足为此需求，并通过JOIN运算连接起来的一个元组。在这个类中，关键是对 `fetchNext()` 方法的实现。在这个方法中采用内外双重迭代的方式来遍历两个元组序列的所有元素，并对其进行一一匹配，判断其是否满足谓词的需求，需要注意的是，每次外层循环结束的时候，都需要对child1的下次取值进行一次判断，如果没有下一个元素要将其置为null，保证循环能够判空退出。

```

1  if(currentTuple == null){
2      if(child1.hasNext()){
3          currentTuple = child1.next();
4      }else{
5          return null;
6      }
7  }

```

```

8         while(currentTuple != null){
9             while(child2.hasNext()){
10                 Tuple tmp = child2.next();
11                 if(joinPredicate.filter(currentTuple,tmp)){
12                     TupleDesc newTupleDesc =
TupleDesc.merge(child1.getTupleDesc(),child2.getTupleDesc());
13                     Tuple newTuple = new Tuple(newTupleDesc);
14                     for(int i = 0;i <
child1.getTupleDesc().numFields(); ++i){
15                         newTuple.setField(i,currentTuple.getField(i));
16                     }
17                     for(int i=0;i<tmp.getTupleDesc().numFields();i++){
18                         newTuple.setField(i +
child1.getTupleDesc().numFields(), tmp.getField(i));
19                     }
20                     return newTuple;
21                 }
22             }
23             currentTuple = child1.hasNext()? child1.next():null;
24             child2.rewind();
25         }
26         return null;

```

## exercise2

在exercise2中，主要的难点就是在于对聚合辅助工具类的实现，在其中要能够记录分组的情况和分组内部的聚合结果。为了能够更好的记录分组情况，方便查询等，因此采用了HaspMap来记录，创建了两个HashMap，groupResult中的key记录的是每一组的field信息，而value则是聚合的结果，groupCount中则是记录了对应分组中一共包含了多少元素，用来辅助计算平均值。每次merge新的元组的时候，只需要查看当前元组对应的field是否在groupResult中，如果在则更新这个地方的值，如果不在则新增值，同时要更新groupCount中的信息，保证两部分数据的一致性。而对于NO\_GROUP这种不需要进行分组的特殊情况，则只需要将key置为null即可轻松实现，不需要再设置额外的存储。

```

1 private void aggregatorSum(Tuple tup) {
2     Field field = tup.getField(this.groupByField);
3     int curData = ((IntField)
tup.getField(this.aggregateField)).getValue();
4     // case NO_GROUPING
5     if(groupByField == NO_GROUPING) {
6         // groupResult is still empty
7         if(groupResult.size() == 0) {
8             groupResult.put(null, curData);
9             groupCount.put(null,1);
10        } else {
11            int sum = groupResult.get(null);
12            int cnt = groupCount.get(null);
13            groupResult.put(null, sum + curData);
14            groupCount.put(null,cnt+1);
15        }

```



```

16     }
17     // case GROUPING
18     else{
19         // groupResult contains the current field
20         if(groupResult.containsKey(field)) {
21             int sum = groupResult.get(field);
22             int cnt = groupCount.get(field);
23             groupResult.put(field, sum + curData);
24             groupCount.put(field,cnt+1);
25         }
26         // groupResult does not contain the current field
27         else {
28             groupResult.put(field, curData);
29             groupCount.put(field,1);
30         }
31     }
32 }

```

## exercise3

在 exercise3 中实现起来比较麻烦的是 HeapPage.java 中所提供的 markSlotUsed() 方法。这个方法实现起来的大致思路是首先定位第i个元组的在header中的位置，然后取出包含第i个元组的header，对其中的某一位进行设置。实验中采用的方式是移位配合加法还原的操作，即先将低位移出，并记录这个插值，然后将需要设置的位设置为指定的值，然后通过移位还原，最后加上之前移位丢失的值。

```

1 private void markSlotUsed(int i, boolean value) {
2     // some code goes here
3     // not necessary for lab1
4     int ith = i/8;
5     int mod = i%8;
6     byte b = header[ith];
7     byte delta = (byte) (b - ((b>>mod)<<mod));
8     b = (byte) ((b >> (mod+1)) * 2 );
9     if(value){
10         b = (byte) (b+1);
11     }
12     b = (byte) (b<<(mod));
13     b += delta;
14     header[ith] = b;
15 }

```