



南開大學
Nankai University

计算机学院
SimpleDB 实验报告

Lab2

姓名：朱霞洋

学号：2113301

专业：计算机科学与技术

2023 年 4 月 6 日

摘要

lab2 实现数据库的一些基本操作，例如 Insert 和 Delete 等，从而能够对数据库中的数据进行更改以及分组或过滤挑选等。Lab2 中的实现会在 Lab1 的基础上进行，并且还会对 Lab1 的一些方法进行完善和修改。

代码链接: [朱霞洋的 Gitlab 仓库](#)

目录

1 提交历史	2
2 设计思路	2
2.1 Exercise1 Filter and Join	2
2.1.1 Predicate	2
2.1.2 JoinPredicate	3
2.1.3 Filter	3
2.1.4 Join	3
2.2 Exercise2 Aggregates	4
2.2.1 InterAggregate 和 StringAggregate	4
2.2.2 Aggregate	5
2.3 Exercise3 HeapFile Mutability	5
2.3.1 HeapPage	5
2.3.2 HeapFile	6
2.3.3 BufferPool	6
2.4 Exercise4 Insertion and deletion	7
2.4.1 Insert	7
2.4.2 Delete	8
2.5 Exercise5 Page eviction	8
3 重难点	9
3.1 Exercise2 中的 Aggregator	9
3.2 Exercise3 中的 insert 与 delete	9
3.3 Exercise5 evictPage	9
4 改动部分	9
4.1 HeapFile	9
4.2 BufferPool	9
5 测试结果	10
5.1 Exercise Test	10
5.2 Query walkthrough	11
5.3 Query Parser	12

1 提交历史




04 Apr, 2023 4 commits	
	lab2 final tests X_yang2113301 authored 1 day ago
	lab2.exercise5 completed the evict policy and relevant methods X_yang2113301 authored 1 day ago
	lab2.exercise4 completed insert and delete operators and passed all the required tests X_yang2113301 authored 1 day ago
	lab2.exercise3 deleted some useless code X_yang2113301 authored 1 day ago
03 Apr, 2023 1 commit	
	lab2.exercise3 implemented the insert and delete tuple methods, and fixed some little bugs. X_yang2113301 authored 1 day ago
01 Apr, 2023 1 commit	
	lab2.exercise2 implemented the Aggregate method and passed all the required tests X_yang2113301 authored 4 days ago
31 Mar, 2023 1 commit	
	lab2.exercise1 passed all the required tests X_yang2113301 authored 5 days ago

图 1.1: 提交记录

2 设计思路

2.1 Exercise1 Filter and Join

在这部分会实现两个操作符：Filter 和 Join，在此之前会需要实现 SimpleDB 中的 Predicate 和 JoinPredicate 类，这两个类相当于一个需要满足的条件，Filter 会返回满足 Predicate 的 tuples，而 Join 会根据 JoinPredicate 连接 tuples。

2.1.1 Predicate

在这个类的作用和结构大概如下：

```
1 private final int field;
```

```

2 private final Op op;
3 private final Field operand;

```

- Op: 需要满足的关系符号 (Operator), 包括大于、大于等于、等于等;
- field : 实际上是被比较的属性的序号, 在进行 Filter 时, 会将 Tuple 的某一属性与某个操作数 (operand) 进行比较, 此处的 field 就是该属性的序号
- operand: 即被用于比较的操作数, 当 tuple 与这个 operand 间满足 Op 关系时 (tuple.getField(field).compare(op, operand) == True), 则会把该 tuple 加入到结果。

2.1.2 JoinPredicate

这个类的结构与作用与 Predicate 大致相似:

```

1 private final int field1;
2 private final int field2;
3 private final Predicate.Op op;

```

- field1 : 是第一个 Tuple 中被比较的 Field 的序号;
- field2 : 是第二个 Tuple 中被比较的 Field 的序号;
- op : 两个 Tuple 中指定的属性需要满足的关系

2.1.3 Filter

这个类相当于一个过滤器, 过滤出满足 Predicate 的 Tuples, 其结构和重要方法如下:

- 成员变量 Predicate p: 即要满足的条件;
- 成员变量 OpIterator child : child 实际上是一个 Tuple 类型的迭代器, 这个类会从中读取 Tuple 来进行过滤;
- 成员方法 fetchNext(): 这个函数会从 child 中找寻下一个符合 Predicate 的 Tuple 并返回, 如 child 已经迭代到底, 则返回 null。

其余详细代码可见 [仓库](#)

2.1.4 Join

Join 操作符用于联合两个来自不同 table 的 Tuple, 当它们符合条件时将其连接, 重要变量和方法的设计思路如下:

- 成员变量 JoinPredicate : 即两个 Tuple 要连在一起时需要满足的条件;
- 成员变量 OpIterator child1 : 从中读取进行比较的第一个 Tuple;
- 成员变量 OpIterator child2 : 从中读取进行比较的第二个 Tuple;
- 成员方法 getTupleDesc(): 要调用 TupleDesc 中的 merge() 方法, 联合两个 TupleDesc;

- 成员方法 `fetchNext()`：不同于 `Filter` 简单遍历即可，`Join` 中需要进行双层遍历，外层是 `child1` 中的 `Tuple`，内层是 `child2` 中的 `Tuple`，从而遍历完所有 `Tuple` 的组合，并联合满足要求的。

其余详细代码可见[仓库](#)。

2.2 Exercise2 Aggregates

在这部分要实现基础的 SQL Aggregate，即分组聚合。Lab2 中只要求完成 (SUM COUNT AVG MAX MIN) 五种运算符条件下的分组聚合，而如果被用于参考进行聚合分组的属性是一个 `String` 类型的属性，则只用实现 `COUNT` 运算符下的分组。此部分要实现 `InterAggregate` `StringAggregate` 以及 `Aggregate`。

2.2.1 InterAggregate 和 StringAggregate

这两个类之间大同小异，只是 `StringAggregate` 只支持 `COUNT` 运算符，大致的结构和重要方法如下：

```

1 private int gbfield;
2 private Type gbfieldtype; //gbfield的类型
3 private int afield;
4 private Op what; //进行Group分组时的操作符
5 private TupleDesc td; //聚合后返回的Tuple的TupleDesc
6 private Map<Field,Integer> groupMap; //分组后的结果
7 private Map<Field , Integer[]> avgMap;
```

- `gbfield` 和 `afield`：这两个 `field` 容易混淆，`gbfield` 是用于 `groupby` 的参考属性，会根据这一属性进行分组；而 `afield` 则是想要得到的 `groupby` 后的结果属性。例如有一张包含若干学生各科成绩的表，想要获取其每个人总成绩是可以用

```

1 SELECT sum(成绩) FROM 成绩表 GROUP BY 姓名
```

则 `gbfield` 是姓名，`afield` 是成绩；

- `Td`：是返回的分组后的 `tuple` 的 `TupleDesc`，一般情况下是 `(groupValue, aggregateValue)`，而当分组依据 `gbfield=NOGROUPING` 时，只返回一个 `(aggregateValue)` 形式的 `tuple`；
- `Map<Field,Integer> groupMap`：这个 `Map` 中的键是不同的 `gbfieldtype`，对应的值是 `group` 后的结果，如 `sum` 或 `max` 或 `count` 等；
- `Map<Field , Integer[]> avgMap`：这是为了平均值操作符 `AVG` 单独创建的 `Map`，数组 `Integer[]` 中会有两个值，一个是个数 `COUNT`，一个是总和 `SUM`，这样设计的原因是考虑到之后会实现 `SUM_COUNT` 和 `SC_AVG`，则在这个 `Map` 中可以方便的同时读取 `COUNT` 与 `SUM`，再进行进一步运算；
- 成员方法 `mergeTupleIntoGroup(Tuple)`：这个方法会将 `Tuple` 进行分组，但要注意当分组标准是 `NOGROUPING` 时，则不会进行严格分组，而是全部放到一起。

在这个函数中要根据 `Op` 的种类分类讨论，如果是 `MIN`，则 `groupMap` 中的 `value` 就是该组的最小值，以此类推。

- 一个 OpIterator 类型的迭代器 iterator: 这个迭代器返回分组后的 Tuples。可以创建一个 ArrayList，再遍历一遍 groupMap，将 Tuple 加入进去，最后返回这个 ArrayList 的迭代器即可。

2.2.2 Aggregate

这部分会根据一个 Aggregator 进行分组，大致的设计思路如下：

```

1  private OpIterator it; //需要被groupby分组的tuples的迭代器
2  private int afield;    //与上一节的afield含义相同
3  private int gfield;    //与上一节的gfield含义相同
4  private Aggregator.Op aop; //groupby分组时的运算 (MIN, MAX, AVG...)
5  private Aggregator aggregator; //分类器, 根据gfield的类型创建为IntegerAggregator或StringAggregator
6  private TupleDesc child_td; //是it中需要被分组的Tuple的TupleDesc
7  private TupleDesc td;
    //分组后的TupleDesc, 如果是NOGROUPING, 则是简单的(aggregateValue), 此外一般都是(groupValue,
    aggregateValue)
8  private Type gbfieldtype; //gfield的类型
9  private Type afieldtype; //afield的类型
10 private OpIterator opIterator; //分组后的Tuples的迭代器

```

重要的方法有：

- 构造函数 Aggregate(): 在构造函数中, 会根据 gfield 的类型建立相应的 aggregator 和 TupleDesc;
- opIterator 的相关函数: 如 open() 和 close() 等, 在 open() 中会调用 aggregator 的 mergeIntoGroup 来进行分组, 并且构造成一个迭代器, 让其他子操作符使用。

2.3 Exercise3 HeapFile Mutability

从这部分开始实现 tuple 的 insert 和 delete, 首先实现在 HeapFile 和 HeapPage 的物理层面进行的 insert 和 delete, 因此需要实现 HeapFile 和 HeapPage 中剩余的一些方法。

2.3.1 HeapPage

HeapPage 中有一个 Tuple 类型的数组 Tuples, 并且由 Lab1 知 HeapPage 有一个 BitMap 用于记录各块 Tuple 是否记录了数据, 因此在 page 中插入或删除一个 Tuple, 会需要更新 Tuples 中的值和 BitMap 对应的位, 要实现的重要方法有：

- markSlotUsed(int i, boolean value): 这个函数会标记第 i 位 BitMap 为 value(存有数据为 1, 无数据为 0), 此处设计用到了位运算, 能方便的转换第 i 为的值:

```

1  byte b = header[Math.floorDiv(i, 8)];
2  byte m = (byte) (1 << (i%8));
3  if(value){
4      header[Math.floorDiv(i,8)] = (byte) (b|m);
5  }else{
6      header[Math.floorDiv(i,8)] = (byte) (b&(~m)); //利用位运算方便地实现headers头的更改
7  }

```

- insertTuple(Tuple t): 该函数会先遍历 BitMap, 找到空 (为 0) 项, 然后在 Tuples 数组的此处放入 t, 注意要设置 t.RecordId;
- deleteTuple(tuple t): 先从 t.RecordId 中获取 t 的序号, 然后将 BitMap 对应位置 0, 再将 Tuples 中此处 tuple 置为 null。
- 此外还有与脏页有关的函数如 markDirty 等, 主要是用于获取更改此页数据的 Transaction 和设置此页是否为脏页等, 这些函数实现较为简单, 主要和之后 BufferPool 中的 insert 和 delete 有关。

2.3.2 HeapFile

此部分也是实现 Tuple 的 insert 与 delete:

- insertTuple(TransactionId tid, Tuple t): 遍历 HeapFile 中的 HeapPage, 找到有空位的一页, 然后在这一页中调用 HeapPage.insertPage(t), 如果该 HeapFile 没有空页, 则为其写入一页空页,

```

1      HeapPageId newid = new HeapPageId(this.getId(), numPages()); // ceate a new page,
2      HeapPage blankPage = new HeapPage(newid, HeapPage.createEmptyPageData());
3      numPage++;
4      writePage(blankPage);

```

然后在这一页中插入 Tuple, 然后返回这一页, 并将其设置为 dirty。

- deleteTuple(Transaction tid, Tuple t): 根据 t 的 RecordId 找到对应 Page 的 Id, 然后在 HeapFile 中找到此页, 并删去 t。
- writePage(Page page): 往现有的 HeapFile 中继续写入一页数据:

```

1      try(RandomAccessFile f = new RandomAccessFile(File, "rw")){
2          f.seek(page.getId().getPageNumber()*bp.getPageSize());
3          byte[] data = page.getPageData();
4          f.write(data);
5      }

```

2.3.3 BufferPool

BufferPool 中也是实现 insertTuple 和 deleteTuple 函数:

- insertTuple(Transaction tid, int tableId, Tuple t): 首先从 Catalog 中根据 tableId 找到对应要插入 Tuple 的表 table, 再调用 HeapFile.insertTuple 方法, 插入 Tuple, 并将该方法返回的操作过的 Page 标记为 dirty, 然后将插入后的 Page 放入 BufferPool。注意, 当 BufferPool 已满时, 要进行内存页的汰换 evictPage()。

```

1      HeapFile table = (HeapFile) Database.getCatalog().getDatabaseFile(tableId);
2      ArrayList<Page> affectedPages = table.insertTuple(tid, t);
3      for(Page page : affectedPages){
4          page.markDirty(true, tid);
5          if (idToPage.size() == numPages) {

```

```

6         evictPage();           //when bufferpool is full, evict page
7     }
8     idToPage.put(page.getId(), page);
9 }

```

- deleteTuple(Transaction tid, Tuple t): 通过 t 的 RecordId 找到 PageId, 进而找到 tableId, 然后在 Catalog 中找到该 table, 调用 HeapFile.deletePage 删去 Tuple t;

2.4 Exercise4 Insertion and deletion

该部分实现 Insert 和 Delete 的操作符, 通过调用 BufferPool 的 insertTuple 和 deleteTuple 实现插入与删除。

2.4.1 Insert

该部分的大致结构和重要方法有:

```

1 private TransactionId t; //执行insert操作的业务
2 private OpIterator child; //待插入的Tuples的迭代器, 来自其他操作符运算后的结果
3 private int tableId; //要插入的表的id
4 private int count; //已经插入的Tuples的数量
5 private TupleDesc returnTP; //在fetchNext中要返回一个仅一个Field的Tuple, 值为count
6 private boolean isAccessed; //该insert操作是否已经调用

```

```

1 protected Tuple fetchNext() throws TransactionAbortedException, DbException {
2     // some code goes here
3     if ( isAccessed )
4         return null; //已经调用则返回null
5     isAccessed = true;
6     while (this.child.hasNext()) {
7         Tuple t = this.child.next();
8         try {
9             Database.getBufferPool().insertTuple(this.t, this.tableId, t); //不断插入child中的Tuple
10            this.count++;
11        } catch (IOException e) {
12            e.printStackTrace();
13            break;
14        }
15    }
16    Tuple tuple = new Tuple(this.returnTP);
17    tuple.setField(0, new IntField(this.count));
18    return tuple; //返回一个要求的Tuple
19 }

```

其余方法都较为简单, 都是 child 迭代器的 open() 与 close() 等, 详细可见仓库。

2.4.2 Delete

Delete 与 Insert 大同小异，都有一个 OpIterator child，其中是需要被删去的 Tuples，Delete 中重要的方法即是 fetchNext 函数，一个个删去 child 中的 Tuple，然后返回一个单 Field 的 Tuple，值是被删去的 Tuples 的数量 count。基本上与 Insert 相同的结构与方法，详细可见仓库。

2.5 Exercise5 Page eviction

先前在插入 Tuple 时提到，如果此时 BufferPool 中 Page 已满，则调用 evictPage() 方法汰换内存页，从而放入新的插入了 Tuple 的一页。

该部分在进行汰换内存页时，用到了最近最少使用页面置换算法的思想，将最近最少使用的页面替换掉，这样能防止较常调用的页面被替换从而降低效率。大致要实现的重要方法有：

- flushPage(PageId pid): 该方法用于将缓冲区 BufferPool 中的内存页刷新至磁盘，以便如若要将该页替换出去时发生数据丢失。此方法调用 writePage 即可：

```

1      HeapPage dirty_page = (HeapPage) idToPage.get(pid); //要刷新至磁盘的内存页
2      HeapFile table = (HeapFile) Database.getCatalog().getDatabaseFile(pid.getTableId());
3      table.writePage(dirty_page); //写至磁盘
4      dirty_page.markDirty(false, null); //更改脏页

```

- discardPage(PageId pid): 删去一页，只用在 BufferPool 中建立的 idToPage 中调用 remove 方法即可；
- flushAllPages(): 对所有内存页调用 flushPage()。该函数只为了测试用，此外不可调用；
- evictPage(): 此方法是核心。在设计汰换方法时，我先在 BufferPool 中新建了一个链表 recentUsedPages 用于记录最近使用的内存页，以及一个 moveToHead() 函数，用于将最近调用的一页移至顶部：

```

1      private LinkedList<Page> recentUsedPages;
2
3      private void moveToHead(int i){
4          Page page = recentUsedPages.get(i);
5          recentUsedPages.remove(i);
6          recentUsedPages.add(0,page);
7      }

```

在 getPage() 函数中将调用 moveToHead(), 而在新建页的时候会调用 recentUsedPages.add(0,newPage)。由此，在 evictPage() 中只用删去该链表最后一项的页即可。

```

1      Page page = recentUsedPages.removeLast(); //find the last used page
2      try{
3          flushPage(page.getId()); //flush this page to the disk
4      }catch (IOException e){
5          e.printStackTrace();
6      }
7      discardPage(page.getId()); //remove it from the BufferPool

```

3 重难点

3.1 Exercise2 中的 Aggregator

由于要实现不同运算符 (MIN、MAX、AVG...) 情况下的分组, gruopby 的方案就不一样, 需要谨慎考虑。同时在实现 AVG 时用了与其他运算符不同的方法, 这是考虑到未来实现 SUM_COUNT 的方便, 具体可见**Exercise 2**中的实现思路。

3.2 Exercise3 中的 insert 与 delete

该部分是我认为 Lab2 中的重难点, Exercise3 中实现 insertTuple 和 deleteTuple 是一环扣一环, 首先实现 HeapFile 和 HeapPage 中的 insert 和 delete 方法, 要修改对应的 BitMap 等如果 HeapFile 中没有了空页, 还要调用 writePage 新建空白页再插入。而在 BufferPool 中则是在顶层调用 HeapFile 的 insert 和 delete 方法进行修改, 具体可见**Exercise 3**中的实现思路。

3.3 Exercise5 evictPage

此部分尝试了不同的替换方案, 一开始直接打算替换掉 idToPage 中的第一个 Page, 但是这样没有考虑到数据库运行的效率, 最理想的情况自然是替换掉最不常使用的内存页, 于是新增一个表示使用情况的链表 recentUsedPages, 并根据 BufferPool 中内存页的调用实时更新, 替换时只用删去最后一项即可, 用到了 LRU 算法的思想, 具体可见**evictPage**的实现思路。

4 改动部分

在 Lab1 基础上做了一些改动, 主要是在 HeapFile 和 BufferPool 两个文件中。

4.1 HeapFile

- HeapFile 中更改了 numPages() 函数, 这是因为在调试进行 BufferPoolWriteTest 时 handleManyDirtyPages 报错, 原因是没有正确地返回 HeapFile 中的页数, 导致无法正确在 BufferPool 中放入新页。这是由于在 Lab1 中我用了一个 int numPages 变量相对静态的存储页数, 而没有想到为 HeapFile 写入新页时 Page 数目的变化, 导致出错, 因此 Lab2 中改为:

```
1 public int numPages() {  
2     numPage = (int)(File.length()/(bp.getPageSize()));  
3     return numPage;  
4 }
```

4.2 BufferPool

- 完善了 getPage():Lab1 中粗略写了 getPage 函数, 但是没有考虑满页时的替换策略, Lab2 中完善了该函数, 当满页时执行 evictPage() 释放空间。
- 新增 recentUsedPages 链表以记录最近使用内存页, 在 evictPage() 时会删去最末尾项。

5 测试结果

5.1 Exercise Test

<ul style="list-style-type: none"> ✓ PredicateTest (simplifiedb) 18毫秒 <ul style="list-style-type: none"> ✓ filter 18毫秒 	<ul style="list-style-type: none"> ✓ JoinPredicateTest (simplifiedb) 16毫秒 <ul style="list-style-type: none"> ✓ filterVaryingVals 16毫秒 	<ul style="list-style-type: none"> ✓ JoinTest (simplifiedb) 19毫秒 <ul style="list-style-type: none"> ✓ rewind 16毫秒 ✓ getTupleDesc 1毫秒 ✓ eqJoin 1毫秒 ✓ gtJoin 1毫秒
<ul style="list-style-type: none"> ✓ JoinTest (simplifiedb.systemtest) 383毫秒 <ul style="list-style-type: none"> ✓ testNoMatch 367毫秒 ✓ testSingleMatch 8毫秒 ✓ testMultipleMatch 8毫秒 	<ul style="list-style-type: none"> ✓ FilterTest (simplifiedb) 15毫秒 <ul style="list-style-type: none"> ✓ rewind 15毫秒 ✓ filterEqualNoTuples 0毫秒 ✓ filterSomeLessThan 0毫秒 ✓ filterAllLessThan 0毫秒 ✓ getTupleDesc 0毫秒 ✓ filterEqual 0毫秒 	
	<ul style="list-style-type: none"> ✓ FilterTest (simplifiedb.systemtest) 487毫秒 <ul style="list-style-type: none"> ✓ testGreaterThanOrEq 428毫秒 ✓ testLessThan 18毫秒 ✓ testEquals 14毫秒 ✓ testLessThanOrEq 13毫秒 ✓ testGreaterThan 14毫秒 	

图 5.2: Exercise 1

<ul style="list-style-type: none"> ✓ IntegerAggregatorTest (simplifiedb) 25毫秒 <ul style="list-style-type: none"> ✓ mergeMin 21毫秒 ✓ testIterator 1毫秒 ✓ mergeAvg 1毫秒 ✓ mergeMax 1毫秒 ✓ mergeSum 1毫秒 	<ul style="list-style-type: none"> ✓ StringAggregatorTest (simplifiedb) 15毫秒 <ul style="list-style-type: none"> ✓ testIterator 14毫秒 ✓ mergeCount 1毫秒
<ul style="list-style-type: none"> ✓ AggregateTest (simplifiedb) 23毫秒 <ul style="list-style-type: none"> ✓ rewind 16毫秒 ✓ countStringAggregate 1毫秒 ✓ maxAggregate 1毫秒 ✓ avgAggregate 1毫秒 ✓ sumStringGroupBy 1毫秒 ✓ minAggregate 1毫秒 ✓ getTupleDesc 1毫秒 ✓ sumAggregate 1毫秒 	<ul style="list-style-type: none"> ✓ AggregateTest (simplifiedb.systemte) 464毫秒 <ul style="list-style-type: none"> ✓ testCount 415毫秒 ✓ testAverageNoGroup 11毫秒 ✓ testMin 11毫秒 ✓ testMax 10毫秒 ✓ testSum 9毫秒 ✓ testAverage 8毫秒

图 5.3: Exercise 2

<ul style="list-style-type: none"> ✓ HeapPageWriteTest (simplifiedb) 388毫秒 <ul style="list-style-type: none"> ✓ deleteNonexistentTuple 367毫秒 ✓ deleteTuple 1毫秒 ✓ testDirty 1毫秒 ✓ addTuple 19毫秒 	<ul style="list-style-type: none"> ✓ HeapFileWriteTest (simplifiedb) 620毫秒 <ul style="list-style-type: none"> ✓ addTuple 620毫秒 	<ul style="list-style-type: none"> ✓ BufferPoolWriteTest (simplifiedb) 1秒 468毫秒 <ul style="list-style-type: none"> ✓ handleManyDirtyPages 452毫秒 ✓ insertTuple 150毫秒 ✓ deleteTuple 866毫秒
--	---	--

图 5.4: Exercise 3

✓ InsertTest (simplifiedb) 368毫秒		✓ InsertTest (simplifiedb.systemtest) 414毫秒	
✓ getNext	366毫秒	✓ testEmptyToOne	389毫秒
✓ getTupleDesc	2毫秒	✓ testOneToEmpty	10毫秒
		✓ testEmptyToEmpty	7毫秒
		✓ testOneToOne	8毫秒
✓ DeleteTest (simplifiedb.systemtest) 2秒 521毫秒			
✓ testGreaterThanOrEq	840毫秒		
✓ testEquals	429毫秒		
✓ testLessThan	414毫秒		
✓ testLessThanOrEq	419毫秒		
✓ testGreaterThan	419毫秒		

图 5.5: Exercise 4

✓ EvictionTest (simplifiedb.systemtest) 1秒 295毫秒
✓ testHeapFileScanWithManyP 1秒 295毫秒

图 5.6: Exercise 5

5.2 Query walkthrough

some_data_file1.txt		some_data_file1.txt	some_data_file2.txt
1	1,100,200	1	1,100,101
2	2,100,300	2	2,100,202
3	3,400,500	3	2,100,203
4	4,500,600	4	3,400,304
5	5,600,700	5	4,500,405
6		6	

图 5.7: test data

2	100	300	1	100	101
2	100	300	2	100	202
2	100	300	2	100	203
3	400	500	3	400	304
4	500	600	4	500	405

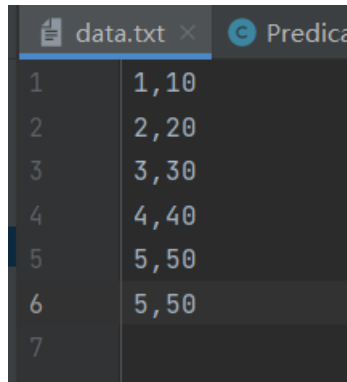
图 5.8: query result

可见返回结果符合

```
1 SELECT *
2 FROM some_data_file1, some_data_file2
3 WHERE some_data_file1.field1 = some_data_file2.field1
4 AND some_data_file1.id > 1
```

5.3 Query Parser

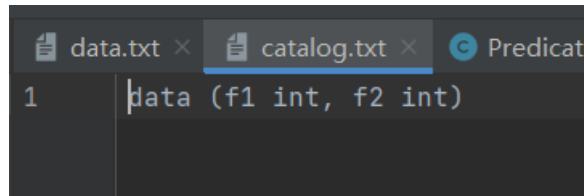
新建以下数据：

A screenshot of a text editor window with two tabs: 'data.txt' and 'Predica'. The 'data.txt' tab is active and shows a table with 7 rows. The first column contains numbers 1 through 7, and the second column contains pairs of numbers separated by a comma. The data is as follows:

1	1,10
2	2,20
3	3,30
4	4,40
5	5,50
6	5,50
7	

图 5.9: Data

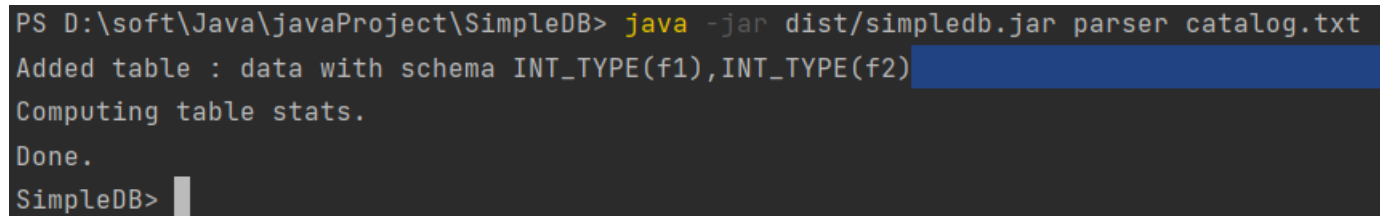
创建以下 catalog.txt:

A screenshot of a text editor window with three tabs: 'data.txt', 'catalog.txt', and 'Predica'. The 'catalog.txt' tab is active and shows a single line of text: 'data (f1 int, f2 int)'. The text is in a monospaced font and is highlighted by the cursor.

1	data (f1 int, f2 int)
---	-----------------------

图 5.10: Catalog.txt

创建数据库：

A screenshot of a Windows command prompt window. The prompt is 'PS D:\soft\Java\javaProject\SimpleDB>'. The user has entered 'java -jar dist/simplydb.jar parser catalog.txt'. The output is: 'Added table : data with schema INT_TYPE(f1),INT_TYPE(f2)', 'Computing table stats.', and 'Done.'. The prompt is now 'SimpleDB>'.

```
PS D:\soft\Java\javaProject\SimpleDB> java -jar dist/simplydb.jar parser catalog.txt
Added table : data with schema INT_TYPE(f1),INT_TYPE(f2)
Computing table stats.
Done.
SimpleDB>
```

图 5.11: SimpleDB

执行 SQL 语句 `select d.f1, d.f2 from data d;`

```
SimpleDB> select d.f1, d.f2 from data d;  
Started a new transaction tid = 0  
Added scan of table d  
  
d.f1    d.f2  
-----  
1       10  
2       20  
3       30  
4       40  
5       50  
5       50  
  
6 rows.  
Transaction 0 committed.  
-----  
0.05 seconds
```

图 5.12: Caption

可见，结果正确，成功创建了一个简单数据库并提取出数据