Report of Lab 3

Yuxuan Duan 516030910573

1 Design Decisions

1.1 About the Operators

Considering that an operator may not actually do the operation, or may be closed & reopened or rewound, I chose to do necessary operations only at the first time *fetchNext()*. It may seems unnatural but it saves time.

1.2 About HashEquiJoin

1. When the first time *fetchNext()* is called, a HashMap(child2Field.hashCode() –> ArrayList<Tuple>) is created. And child2 is sequentially scanned and appended to the corresponding ArrayList in the HashMap. The code above will only be run once.

2. Every time *fetchNext()* is called, child1Field.hashCode() is calculated and is used to index the ArrayList in HashMap. The iterator takes the next tuple in the ArrayList, uses Predicate to judge the two fields are indeed equal, and finally makes a new tuple and returns it.

3. Each time an ArrayList is indexed and the first eligible tuple is returned, the rest of the tuples in that ArrayList are all possibly eligible. Also, all the possibly eligible tuples are in that ArrayList. So until we have finished scanning that ArrayList, we can simply store that ArrayList, mark *fetching = true*, and continue scanning (directly starting Step 3) that ArrayList in the next few times we call fetchNext(). After we finish scanning, mark *fetching = false*, and we will head back to Step 2 next time we call fetchNext(). Predicate is also needed in Step 3.

Note: Using Predicate in Step 2 and 3 is necessary. Different fields may have a same hash code, so we use Predicate to further ensure the two fields are the same, even after we have known that their hash codes are the same.

1.3 About Join Operator Choice

As we currently haven't implemented the optimizer, so I added a special judge in the Join class. When a Join class is created with *op == Predicate.Op.EQUAL*, operations of HashEquiJoin will be done if fetchNext() is called.

In the future labs when I implement the optimizer, I will remove the above usage.

2 API Changes

2.1 BufferPool.replacePage()

Sometimes we need to create a new page without getting it through the BufferPool, edit it and write it back (e.g. Tests in BufferPoolWriteTest). If we try to mark a page dirty which is not in the BufferPool, flushPage() will actually do nothing. So we need a method to get a page into the BufferPool directly from a existing page in the memory.

BufferPool.replacePage():

- input a existing page in the memory
- if this page (of an old version) is already in the BufferPool, replace the old page in the BufferPool with the new page.
- if this page is not in the BufferPool, simply add it.

2.2 TupleDesc.setFieldName()

It provides a convenient method to edit the name of a field and is used when an Aggregate class edits the groupByFieldName of the TupleDesc of its returnIterator according to the groupByFieldName of its childIterator. (IntegerAggregator and StringAggregator cannot do this because they do not know the groupByFieldName until mergeTupleIntoGroup() is called.)

3 Missing or Incomplete Elements

There was no more missing or incomplete element than those in Lab 2 to the best of my knowledge.

4 Time Spent and Difficulties

I roughly spent 20 hours for Lab 3. In this lab, I spent much time debugging and some of the bugs are in the classes I implemented in Lab 1 or 2. I think this happened because now I'm confronted with far more complicated tasks, and some special situations were not thought over in the previous labs. I still have some minor difficulties in the structure of SimpleDB yet they are not as confusing as they were in Lab 1 and 2.