Project 2

Database Management Systems (DM556)

UNIVERSITY OF SOUTHERN DENMARK

Group 2
Mark Jervelund (Mjerv15) Troels B. Petersen (trpet15) IMADA

April 4, 2017

Overall Status

The group managed to complete the tasks and therefore the project is considered complete.

Division of Labor

The group worked on the project either sitting together at the university or at home remotely working together and splitting tasks when possible. A lot of the time was spent understanding how to implement a solution. Especially Sort and Merge-Join was not very straight forward. The work was very evenly divided - both when writing the code, but also when writing the report.

Specification

The group was tasked with implementing four operators; Selection, Projection, Sort and Merge-Join.

Selection

Selection is a very basic operation in database management. It uses relational algebra to select the elements. Now the spec for this operator says that every query is combined with a relational *or*. This means that nothing "fancy" has to be done. It should simply select everytime one or more queries return true.

Projection

The projection is also one of the more basic operations in database management. A projection extracts the columns from a relation, however, unlike in relational algebra, this operator does not eliminate duplicates.

Sort

Sort has to be external. External sorting is used in applications where huge amounts of data has to be sorted, thus the data has to sorted in chunks since it cannot all be in main memory. A variant of merge-sort will be used, since it can sort on parts of the data and then combine the sorted parts.

Merge-Join

The merge-join assumes that the both inputs are sorted. It then has to merge where possible.

Implementation

Selection.java

Selection starts by assigning local protected variables some values from the parameters.

```java
   */
  public Selection(Iterator iter, Predicate... preds) {
    this.iterator = iter;
    this.predicates = preds;
    this.schema = iter.schema;
    this.tuple = null;
```

The selection process takes place in the hasNext() function. Here it keeps checking if there are more elements to be selected using the evalulate() function. It returns true if it allows the selection and false if there are no more elements to be selected.

```java
   */
  public boolean hasNext() {
    while(this.iterator.hasNext()) {
      this.tuple = this.iterator.getNext();

      for(int i = 0; i < this.predicates.length; ++i) {
        if(this.predicates[i].evaluate(this.tuple)) {
          return true;
        }
      }
    }
```

The getNext() function is what actually gets the elements. It returns a tuple containing the next element. If there are no more elements to be selected, it will throw an exception, telling that there are no more tuples.

```
   * @throws IllegalStateException if no more tuples
   */
  public Tuple getNext() {
    if(this.tuple == null) {
      throw new IllegalStateException("no more tuples");
    } else {
      Tuple tuple = this.tuple;
      this.tuple = null;
      return tuple;
```

Projection.java

The projection starts by assigning the schema a new schema with the "fields amount of fields". After that it copies all the fields into new schemas using initField(). The last two lines simply assign the parameters to the variables of the class.

```
  public Projection(Iterator iter, Integer... fields) {
    this.schema = new Schema(fields.length);

    for(int i = 0; i < fields.length; ++i) {
      this.schema.initField(i, iter.schema, fields[i]);
    }

    this.iterator = iter;
    this.integers = fields;
  }
```

HasNext() is very simple in the Projection. Here it simply returns where there is another tuple available.

```
  public boolean hasNext() {
    return this.iterator.hasNext();
  }
```

The getNext() function returns the next tuple. However, in contrast to the getNext() from Selection, this getNext() uses the integer fields and also sets the fields of this new tuple before returning it.

```
  public Tuple getNext() {
```

Sort.java

For sort we used a filescan, and a heapfile for the sorting.
First the data is loaded into a heapfile. from there they can be sorted using a sorting function.

```
        while (iter.hasNext()) {
                // Load the records into the internal memory
                for (int i = 0; i < sortMemSize; i++) {
                        if (iter.hasNext()) {
                                internal[i] = iter.getNext();
                                all[i] = internal[i].getField(0);
                                hashmap.put(all[i], internal[i]);
                        }
                } // for

                ArrayList<Object> queue = new ArrayList<Object>();

                for (Object object : all) {
                        if (object != null) {
```

this sorter functions sorts the elements recursively, by using the heapfile with records of the tuples.

```
        private HeapFile[] sorter(HeapFile[] records, int bufSize, Iterator
            ↪ iter, int sortfield) {
                int heapCount = getHeaps(records);
                if (heapCount == 1) {
                        return records;
                }
                if (heapCount >= bufSize) {
                        heapCount = bufSize - 1;
                }
                FileScan[] scan = new FileScan[heapCount];

                // Create a new filescan on every record in the current
                    ↪ record array
                for (int i = 0; i < heapCount; i++) {
                        scan[i] = new FileScan(iter.schema, records[i]);
                }
                HeapFile file = new HeapFile(null);
                Tuple[] tuples = new Tuple[heapCount];
                int compared = 0;

                // Load the tuples from the filescanner
                for (int i = 0; i < tuples.length; i++) {
                        tuples[i] = scan[i].getNext();
                }

                while (compared != heapCount) {
                        Object[] smallest = { null, null };
                        int smallestPos = 0;
                        int current = 0;
                        for (Tuple tuple : tuples) {
                                Object next = tuple.getField(sortfield);
                                if (smallest[0] == null) {
                                        smallest[0] = next;
                                        smallest[1] = next;

                                } else { // compare
                                        smallest[1] = next;
                                        java.util.Arrays.sort(smallest);

                                        if (smallest[0] == next) {
                                                smallestPos = current;
                                        }
                                }
                                current++;
                        }
                        file.insertRecord(tuples[smallestPos].data);
                        if (scan[smallestPos].hasNext()) {
                                tuples[smallestPos] = scan[smallestPos].
                                    ↪ getNext();
                        } else {
                                tuples[smallestPos].setField(sortfield,
                                    ↪ Integer.MAX_VALUE);
                                compared++;
                        }
                }
                records[heapCount - 1] = file;
                HeapFile[] rest = Arrays.copyOfRange(records, heapCount -
                    ↪ 1, records.length);
                return sorter(rest, bufSize, iter, sortfield);
```

MergeJoin.java

The merge join function first merges the two schemas

```
    public MergeJoin(Iterator left, Iterator right, Integer lcol, Integer
       ↪ rcol) {
        this.left = left;
        this.right = right;
        this.lcol = lcol;
        this.rcol = rcol;
        schema = Schema.join(left.schema, right.schema);
```

The Hasnext function then first selects a left tuple, and compares it to all right tuples, if it two that are comparable it returns true. else it loops over all combinations and returns false if none are found. but also the next element in the array is stored in the next variable

```
    public boolean hasNext() {
        while (true) {
            if (outer == null) {
                if (left.hasNext()) {
                    outer = left.getNext();
                } else { return false; }
            }
            while (this.right.hasNext()) {
                Tuple rightTuple = right.getNext();
                next = Tuple.join(outer, rightTuple, schema);
                if (outer.getField(lcol) == rightTuple.getField(rcol)) {
                    return true;
                }
            }
            outer = null;
            right.restart();
```

The next variable is then stored in a temp variable, set to null, and returned from the temp variable

```
    public Tuple getNext() {
        // validate the next tuple
        if (next == null) {
            throw new IllegalStateException("no more tuples");
        }
```

```java
        // return (and forget) the tuple
        Tuple tuple = next;
        next = null;
        return tuple;
    }
```

Testing

Testing this time around was very successful. It reports that test1, test2 and test3 completed successfully.
Further more when comparing the expected output with the supplied ExpectedOutput.txt we got the same output except for some initializing prints and some lines from the explain print statement.

Appendix

Selection.java

```java
package relop;

/**
 * The selection operator specifies which tuples to retain under a
    ↪ condition; in
 * Minibase, this condition is simply a set of independent predicates
    ↪ logically
 * connected by OR operators.
 */
public class Selection extends Iterator {

  protected Iterator iterator;
  protected Predicate[] predicates;
  protected Tuple tuple;
  /**
   * Constructs a selection, given the underlying iterator and predicates.
   *
   */
  public Selection(Iterator iter, Predicate... preds) {
    this.iterator = iter;
    this.predicates = preds;
    this.schema = iter.schema;
    this.tuple = null;
  }

  /**
   * Gives a one-line explaination of the iterator, repeats the call on any
   * child iterators, and increases the indent depth along the way.
   */
  public void explain(int depth) {
    System.out.print("Selection : ");

    for(int i = 0; i < this.predicates.length - 1; ++i) {
      System.out.print(this.predicates[i].toString() + " OR ");
    }

    System.out.println(this.predicates[this.predicates.length - 1]);
    this.iterator.explain(depth + 1);
  }
  /**
   * Restarts the iterator, i.e. as if it were just constructed.
   */
  public void restart() {
    this.iterator.restart();
    this.tuple = null;
  }

  /**
   * Returns true if the iterator is open; false otherwise.
   */
  public boolean isOpen() {
    return this.iterator != null;
  }

  /**
   * Closes the iterator, releasing any resources (i.e. pinned pages).
   */
  public void close() {
    if(this.iterator != null) {
      this.iterator.close();
      this.iterator = null;
    }
  }

  /**
   * Returns true if there are more tuples, false otherwise.
   */
  public boolean hasNext() {
    while(this.iterator.hasNext()) {
      this.tuple = this.iterator.getNext();

      for(int i = 0; i < this.predicates.length; ++i) {
        if(this.predicates[i].evaluate(this.tuple)) {
          return true;
        }
      }
    }
    return false;
  }

  /**
   * Gets the next tuple in the iteration.
   *
   * @throws IllegalStateException if no more tuples
   */
  public Tuple getNext() {
    if(this.tuple == null) {
      throw new IllegalStateException("no more tuples");
    } else {
      Tuple tuple = this.tuple;
      this.tuple = null;
      return tuple;
    }
  }

} // public class Selection extends Iterator
```

sort.java

```java
package relop;

import global.*;
import heap.HeapFile;

import java.io.File;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;

public class Sort extends Iterator implements GlobalConst {
        protected Iterator iterator;
        protected HeapFile file;
        protected FileScan scan;

  /**
   * Constructs a sort operator.
   * @param sortMemSize the size the memory used for internal sorting. For
       ↪ simplicity, you can assume it is in the unit of tuples.
   * @param bufSize the total buffer size for the merging phase in the unit
       ↪  of page.
   *
   */
  public Sort(Iterator iter, int sortfield, int sortMemSize, int bufSize) {

          this.iterator = iter;
          schema = iter.schema;
          HeapFile[] records = new HeapFile[bufSize];
          Tuple[] internal = new Tuple[sortMemSize];
          Object[] all = new Object[sortMemSize];
          HashMap<Object, Tuple> hashmap = new HashMap<Object, Tuple>();

          int pos = 0;
          // read data into sorting area
          while (iter.hasNext()) {
                  // Load the records into the internal memory
                  for (int i = 0; i < sortMemSize; i++) {
                          if (iter.hasNext()) {
                                  internal[i] = iter.getNext();
                                  all[i] = internal[i].getField(0);
                                  hashmap.put(all[i], internal[i]);
                          }
                  } // for

                  ArrayList<Object> queue = new ArrayList<Object>();

                  for (Object object : all) {
                          if (object != null) {
                                  queue.add(object);
                          }
                  }

                  all = queue.toArray();
                  // Sort the tuples
                  java.util.Arrays.sort(all);

                  records[pos] = new HeapFile(null);
```

```
            for (Object object : all) {
                records[pos].insertRecord(hashmap.get(object).data);
            }
            pos++;
        }

        file = sorter(records, bufSize, iter, sortfield)[0];
        scan = new FileScan(iter.schema, file);
    }

    private HeapFile[] sorter(HeapFile[] records, int bufSize, Iterator iter, int sortfield) {
        int heapCount = getHeaps(records);
        if (heapCount == 1) {
            return records;
        }
        if (heapCount >= bufSize) {
            heapCount = bufSize - 1;
        }
        FileScan[] scan = new FileScan[heapCount];

        // Create a new filescan on every record in the current record array
        for (int i = 0; i < heapCount; i++) {
            scan[i] = new FileScan(iter.schema, records[i]);
        }
        HeapFile file = new HeapFile(null);
        Tuple[] tuples = new Tuple[heapCount];
        int compared = 0;

        // Load the tuples from the filescanner
        for (int i = 0; i < tuples.length; i++) {
            tuples[i] = scan[i].getNext();
        }

        while (compared != heapCount) {
            Object[] smallest = { null, null };
            int smallestPos = 0;
            int current = 0;
            for (Tuple tuple : tuples) {
                Object next = tuple.getField(sortfield);
                if (smallest[0] == null) {
                    smallest[0] = next;
                    smallest[1] = next;

                } else { // compare
                    smallest[1] = next;
                    java.util.Arrays.sort(smallest);

                    if (smallest[0] == next) {
                        smallestPos = current;
                    }
                }
                current++;
            }
            file.insertRecord(tuples[smallestPos].data);
            if (scan[smallestPos].hasNext()) {
```

```
                                        tuples[smallestPos] = scan[smallestPos].
                                            ↪ getNext();
                              } else {
                                        tuples[smallestPos].setField(sortfield,
                                            ↪ Integer.MAX_VALUE);
                                        compared++;
                              }
                    }
                    records[heapCount - 1] = file;
                    HeapFile[] rest = Arrays.copyOfRange(records, heapCount -
                        ↪ 1, records.length);
                    return sorter(rest, bufSize, iter, sortfield);
          }

          private int getHeaps(HeapFile[] records) {
                    int result = 0;
                    while (records[result] != null) {
                              result++;
                    }
                    return result;
          }

          @Override
          public void explain(int depth) {
                    FileScan fs = new FileScan(iterator.schema,file);
                    fs.explain(depth);
          }

          @Override
          public void restart() {
               scan.restart();
          }

          @Override
          public boolean isOpen() {
               return scan.isOpen();
          }

          @Override
          public void close() {
                    if (scan != null){
                              scan.close();
                              scan = null;
                    }
          }

          @Override
          public boolean hasNext() {
                    return scan.hasNext();
          }

          @Override
          public Tuple getNext() {
                    return scan.getNext();
//                  throw new UnsupportedOperationException("Not implemented");
          }

}
```

MergeJoin.java

```java
package relop;


import java.util.IllegalFormatException;

public class MergeJoin extends Iterator {


    /**
     * The underlying left iterator.
     */
    protected Iterator left;

    /**
     * The underlying right iterator.
     */
    protected Iterator right;

    /**
     * left col.
     */
    protected Integer lcol;

    /**
     * right col.
     */
    protected Integer rcol;

    /**
     * Current tuple from left iterator.
     */
    protected Tuple outer;


    /**
     * Next tuple to return.
     */
    protected Tuple next;


    public MergeJoin(Iterator left, Iterator right, Integer lcol, Integer
        ↪ rcol) {
        this.left = left;
        this.right = right;
        this.lcol = lcol;
        this.rcol = rcol;
        schema = Schema.join(left.schema, right.schema);

    }


    @Override
    public void explain(int depth) {

        indent(depth);
        System.out.print("Merge␣join␣:␣");
        for (int i = 0; i < this.schema.names.length - 1; i++) {
            System.out.println("{" + this.schema.names[i] + "}");
        }
        System.out.println("{" + this.schema.names[this.schema.names.length
            ↪ - 1] + "}");
        this.left.explain(depth + 1);
        this.right.explain(depth + 1);
}

    @Override
    public void restart() {
        left.restart();
        right.restart();
        outer = null;
        next = null;
    }

    @Override
    public boolean isOpen() {
        return (left != null);
    }

    @Override
    public void close() {
        if (left != null) {
            left.close();
            right.close();
            left = null;
            right = null;
        }
    }

    @Override
    public boolean hasNext() {
        while (true) {
            if (outer == null) {
                if (left.hasNext()) {
                    outer = left.getNext();
                } else { return false; }
            }
            while (this.right.hasNext()) {
                Tuple rightTuple = right.getNext();
                next = Tuple.join(outer, rightTuple, schema);
                if (outer.getField(lcol) == rightTuple.getField(rcol)) {
                    return true;
                }
            }
            outer = null;
            right.restart();
        }

//                throw new IllegalStateException("debugging crash");

    }

    @Override
    public Tuple getNext() {
        // validate the next tuple
        if (next == null) {
            throw new IllegalStateException("no more tuples");
        }
        // return (and forget) the tuple
        Tuple tuple = next;
        next = null;
        return tuple;
    }

}
```

testing output

```
Creating database...
Replacer: Clock

Running basic relational operator tests...

Test 1: Primative relational operators

  ~> test selection (Age > 65 OR Age < 15)...

Selection : {3} > 65.0 OR {3} < 15.0
    FileScan : null
DriverId  FirstName            LastName             Age        NumSeats
-----------------------------------------------------------------------

1         f1                   l1                   7.7        101
9         f9                   l9                   69.3       109
10        f10                  l10                  77.0       110

  ~> test projection (columns 3 and 1)...

Projection : {3}, {1}
    FileScan : null
Age       FirstName
------------------------------

7.7       f1
15.4      f2
23.1      f3
30.8      f4
38.5      f5
46.2      f6
53.9      f7
61.6      f8
69.3      f9
77.0      f10

  ~> selection and projection (pipelined)...

Projection : {3}, {1}
Selection : {3} > 65.0 OR {3} < 15.0
      FileScan : null
Age       FirstName
------------------------------

7.7       f1
69.3      f9
77.0      f10



Test 1 completed without exception.
```

```
---------------------------------------------
           Reads     Writes    Allocs    Pinned
---------------------------------------------
insert     0         8         7         0
select     0         0         0         0
project    0         0         0         0
both       0         0         0         0
---------------------------------------------


Test 2: Sorting Test

...Inserted

 ~> sort numbers

 cheking the result.


 Test 2 completed without error.

Test 3: MergeJoin operator

  Projection : {DriverId}
{FirstName}
{LastName}
{Age}
{NumSeats}
{DriverId}
{GroupId}
{FromDate}
{ToDate}
    FileScan : null
    FileScan : null
DriverId   FirstName           LastName            Age        NumSeats
    ↪ DriverId   GroupId    FromDate   ToDate
------------------------------------------------------------------------------------------------------
    ↪
1          Ahmed               Elmagarmid          25.0       5          1
    ↪          2          2/12/2006  2/14/2006
1          Ahmed               Elmagarmid          25.0       5          1
    ↪          3          2/15/2006  2/16/2006
2          Walid               Aref                27.0       13         2
    ↪          6          2/17/2006  2/20/2006
2          Walid               Aref                27.0       13         2
    ↪          7          2/18/2006  2/23/2006
3          Christopher         Clifton             18.0       4          3
    ↪          5          2/10/2006  2/13/2006
3          Christopher         Clifton             18.0       4          3
    ↪          4          2/18/2006  2/19/2006
3          Christopher         Clifton             18.0       4          3
    ↪          2          2/24/2006  2/26/2006
4          Sunil               Prabhakar           22.0       7          4
    ↪          1          2/19/2006  2/19/2006
5          Elisa               Bertino             26.0       5          5
    ↪          7          2/14/2006  2/18/2006
6          Susanne             Hambrusch           23.0       3          6
    ↪          6          2/25/2006  2/26/2006
```

```
8           Arif                    Ghafoor                 20.0       5          8
    ↪           5             2/20/2006  2/22/2006
9           Jeff                    Vitter                  19.0       10         9
    ↪           1             2/15/2006  2/15/2006


Test 3 completed without exception.
-----------------------------------------------
          Reads    Writes   Allocs   Pinned
-----------------------------------------------
driver2   0        3        2        0
rides2    0        3        2        0
m_join    0        0        0        0
-----------------------------------------------


All basic relational operator tests completed; verify output for
    ↪ correctness.

Process finished with exit code 0
```