CS3223 Project

Project Team Members:

- Chua Wei Wen (A0156034M)
- Khor Shao Liang (A0160529E)
- Sim Kwan Tiong, Damien (A0155983N)

1. Introduction

This document provides information on the features that has been implemented into the query processing engine.

2. Implementation

The following features are implemented:

2.1. Block-Nested Loop Join

The Block-Nested-Loop Join is similar to the existing Nested-Loop Join algorithm. The main difference between these two algorithm is how the left pages are loaded. The Block-Nested-Loop Join uses `numBuff - 2` number of buffers, where `numBuff` is the total number of buffers available, to load the left pages instead of just using a single buffer. The following method `loadLeftBatches` loads in the left pages:

```
private void loadLeftBatches() {
    for(int m=0; m< (numBuff-2); m++) {
        Batch batch = left.next(); // get next batch of data
    if(batch != null) leftbatches.add(batch);
    }
}
```

After loading the left pages, the tuples will be extracted from the pages and added to `leftTuples` list.

```
private void loadTuplesFromBatch() {
    for(int m=0; m<leftbatches.size(); m++) {
        Batch batch = leftbatches.get(m);
        for(int n=0; n < batch.size(); n++) //for each of the batch read in the tuples
            leftTuples.add(batch.elementAt(n));
    }
}
```

This list is used to compare the tuples against the right tuple to perform the join. If there is a match, the tuples will be joined together and added to `outbatch`. When the `outbatch` is full, it will be returned.

```
for(i=lcurs; i< leftTuples.size(); i++){
   for(j=rcurs;j<rightbatch.size();j++){
       Tuple lefttuple = leftTuples.get(i);
       Tuple righttuple = rightbatch.elementAt(j);
       if(lefttuple.checkJoin(righttuple,leftindex,rightindex)){
           Tuple outtuple = lefttuple.joinWith(righttuple);
           outbatch.add(outtuple);
           if (outbatchFull(i, j, outbatch)) return outbatch;
       }
   }
   rcurs =0;
}
```

2.2. External Sort

The external sort algorithm is written in `ExternalSort` as an `Operator` object. The `table` holds the data, the `joinAttribute` is the `Attribute` that is used to sort the data in `table`. `numBuff` is the number of buffers and `flag` indicates whether to eliminate duplicates for distinct operation (see Section 2.4).

```
public ExternalSort(Operator table, Attribute joinAttribute, int numBuff, boolean flag) {
    super(OpType.SORT);
    this.table = table;
    this.numBuff = numBuff;
    this.batchSize = Batch.getPageSize()/table.getSchema().getTupleSize();
    this.joinAttribute = joinAttribute;
    this.fileNum = instanceNum++;
    this.isDistinct = flag;
}
```

Since external sort is required to generate temporary files to store intermediate results, a unique variable is assigned to each instance of `ExternalSort` created. This is achieved by creating a static variable `instanceNum`, which is assigned to `fileNum` in the constructor as the unique number. The `instanceNum` is incremented which will be used as the next unique number when another `ExternalSort` object is created.

The name of the temporary files will contain the integers `fileNum`, `passNum` and `runNum` to differentiate the files.

The external sort is done in the `open` method. There are two phases in external sort:

1. Generate sorted runs - implemented in the method `generateRuns`.
2. Merge the sorted runs - implemented in the method `executeMerge`.

A `Vector` object `sortedRunFiles` holds the file pointer to the files that are generated during the first phase.

2.2.1. Generate Sorted Runs

In the `generateRuns` method, a `Vector` object `batchList` that holds `Batch` object simulates the buffers that are available to hold the pages. Pages are added into the `batchList` until it is "full", in which the `batchList` will be passed to another method `generateSortedRun`.

In `generateSortedRun`, all the tuples are loaded from the pages and stored in a temporary `Vector` object `tupleList`. The `tupleList` is sorted using `SortComparator` which sorts the list based on the join attribute. After the list is sorted, the tuples will be added to the pages. If the page is full, the page `currentBatch` will be added to `sortedRun`. Pages will be added to `sortedRun` until it is full, in which it returns the completed sorted run.

The `sortedRun` will be passed to `writeRun` method, which writes to a temporary file to simulate writing of data to disk with a unique file name. The `File` object that holds the file pointer to the generated temporary files will be returned, and it will be stored in the `Vector` object `sortedRunFiles`. This allows easier access to the files during the second phase.

The `generateRuns` will run until all the tuples have been read, and all the sorted runs are generated.

2.2.2. Merge Sorted Runs

The second phase is executed in the `executeMerge` method. The number of buffers available `numUsableBuff` simulates the merging phase, in which one buffer is used for output, while the rest of the buffers are used for merging.

The while loop keeps executing until all the runs have been merged into a single sorted run. A `Vector` object `newSortedRuns` keeps track of the new files containing the new sorted runs that are merged. The `startIndex` and `endIndex` essentially picks the indexes of the file pointers stored in `sortedRunFiles` that can be placed into the available buffer space `numUsableBuff`. These files are picked and stored in a `List` object `runsToSort`. This list is passed to another method `mergeSortedRuns`.

In the `mergeSortedRuns`, `inputBuffers` holds the content of the pages that will be loaded from the file. The `inputStreams` is a list of `ObjectInputStream` object that reads in the respective `File` object from the `sortedRuns`. The `for` loop below reads in each page that are stored in the `File` and place it into the `inputBuffers`.

```
for (ObjectInputStream ois : inputStreams) {
    Batch batch = readBatch(ois);
        inputBuffers.add(batch);
}
```

The `int[]` array `batchTrackers` keeps track of the number of tuples that has been added to the `outputBuffer` for each `inputBuffers`. The `Tuple` object `smallest` keeps track of the smallest tuple encountered so far in the next `for` loop, and the `indexOfSmallest` keeps track of the page index of the tuple.

After the smallest tuple has been found, the `backTrackers` for the index of the `inputBuffers` where the smallest tuple is found will be incremented. This indicates that one additional tuple will be added to the `outputBuffer` and will then point to the next tuple in that buffer. If all the tuples in that `inputBuffer` has been added to the `outputBuffer`, that input buffer will be replaced with a new page, and the `batchTrackers` will be reset to 0 to point to the first tuple.

Next it will check whether `isDistinct` is true. This will be described in Section 2.4. Otherwise, the tuple will be added to the `outputBuffer`.

When the `outputBuffer` is full, it will be written to a temporary file. The whole process is repeated until a sorted run is generated and a `File` object holding the pointer to the file will be returned. This file will be added to `newSortedRuns` that contains the newly merged sorted runs. This process is repeated until the sorted runs are merged. The old `sortedRunFiles` will be deleted by the `clearTempFiles` method, and the `sortedRunFiles` will point to the `newSortedRuns`.

The whole process repeats until there is only one sorted run left.

2.2.3. Passing of data

In `next`, `ExternalSort` will pass the sorted table data in pages to the `Operator` object that calls `ExternalSort`.

2.2.4. Closing

In `close`, `ExternalSort` will close the `table` and clear the single sorted run file data.

2.3. Sort-Merge Join

The Sort-Merge Join algorithm is written in `SortMergeJoin` as an `Operator` object. The sort-merge join has two phases:

1. Sorting phase - the two tables are sorted using `ExternalSort`
2. Merging phase - merge the two tables based on the join attributes

2.3.1. Sorting phase

Two `ExternalSort` objects are created to sort both the left and right table.

```
leftSort = new ExternalSort(left, leftattr, numBuff, false);
rightSort = new ExternalSort(right, rightattr, numBuff, false);
```

After the two tables were sorted, the data will be written to temporary files using the `writeSortedFiles` method, where a unique file name is assigned to each page that are written. The `File` object that points to the temporary files are added to a list of files and is returned from the method. These list of files are stored in `leftSortedFiles` and `rightSortedFiles` for each table.

2.3.2. Merging phase

In the merging phase, one buffer `outbatch` is allocated for output, one buffer `rightbatch` for the right table, while the rest of the buffers `leftbatches` (or a block) are allocated for the left table. There are several pointer variables used to point to the correct position of the tuple:

- `lcurs` - points to a left tuple in the current page
- `rcurs` - points to a right tuple in the current page
- `leftBatchIndex` - points to the current left page where the left tuple is located
- `leftBlockIndex` - points to the current left block where the left page is located
- `rightBatchIndex` - points to the current right page where the right tuple is located
- `rightFirstMatchIndex` - points to very first right tuple that contains the same value. This is used to backtrack the `rightBatchIndex` pointer if the next left tuple reads in the same value again.
- `rightFirstMatchBatchIndex` - points to the page containing the very first right tuple that contains the same value.

The boolean variable `hasMatch` is set to `true` if the join results matched. This is used to handle the case where the left tuple has duplicate values.

In the `next` method, a `while` loop is executed while the `outbatch` is not full. Inside this loop, the first segment of code loads the left buffers, while the next segment loads the right buffer. If either all the left or right pages has been read, the execution will call `close`. However, if the `hasMatch` remains true after all the right pages has been read, pointer will be set to the first right tuple that contains the same value using the `rightFirstMatchIndex`, and the page that contain thats tuple be will loaded to the right buffer using `rightFirstBatchIndex`. This is to handle the case where there may be another duplicate value in the next left tuple.

The `while` loop reads each tuple from the left table and right table, and their join attributes are compared. There are three cases:

1. Left tuple is smaller - `lcurs` pointer will point to the next left tuple.
    a. If `hasMatch` is true, this means there is a duplicate value in the left tuple, and the right pointer will point back to the very first right tuple containing the value using `rightFirstMatchIndex` and `rightBatchIndex`.
    b. `hasMatch` will set to `false` regardless of whether `hasMatch` is initially true or false.
2. Left tuple is bigger - `rcurs` pointer will point to the next right tuple. `hasMatch` will be set to false.
3. Left tuple matches right tuple.
    a. If `hasMatch` is false, this means this is the first match that is encountered after some iterations. The pointer to the right tuple will be saved using `rightFirstMatchIndex` and `rightBatchIndex`. As explained in 1a, this is to keep track of the first right tuple containing the value which is necessary if there is a dulicate value in the left tuple.
    b. Regardless of whether `hasMatch` is true or false, a new tuple will be created by joining the two tuples and added to `outbatch`. `rcurs` pointer will point to the next right tuple.

The `outbatch` will be returned. This process is repeated until one of the tables has been fully read. In the `close` method, `SortMergeJoin` will clear all the temporary files that were generated earlier.

2.4. Distinct

The elimination of duplicated is implemented using a variant of optimized sort-based approach. Given a relation `R`, the attributes of `R` are passed to `ExternalSort`. Sorted runs are generated with the extracted attributes. During the merging phase, the duplicates are removed with the following algorithm:

```
lastTupleAdded;
if (isDistinct) {
    if (current smallest tuple != lastTupleAdded) {
        outputBuffer.add(current smallest tuple);
        lastTupleAdded = current smallest tuple;
    } else {
        // Duplicates detected, ignore
    }
} else {
    outputBuffer.add(current smallest tuple);
}
```

Comparison of the tuples are based on the extracted attributes. The comparator is modified such that if the SQL query contains `DISTINCT` the comparator will take in the flag and the extracted attributes that appear in the `DISTINCT` clause. Hence, when it compares the tuples it will compare the tuple based on the attributes. The modification of the override `compare()` method is as follows:

```
if (isDistinct) {
    boolean hasSameAttr = true;
    int finalComparisonResult = 0;
    Vector attList = joinAttributes;
    for (int i = 0; i < attList.size(); i++) {
        int index = schema.indexOf((Attribute) attList.get(i));
        int result = Tuple.compareTuples(t1, t2, index);
        finalComparisonResult = result;

        if (result != 0) {
            hasSameAttr = false;
            break;
        }
    }
    return hasSameAttr ? 0 : finalComparisonResult;
}
```

For example, given a relation `R(firstname, lastname, age, allowance)` and three tuples, with the extracted attributes `firstname`, `lastname` and `age`;

```
- Tuple A(John, Doe, 18, 500),
- Tuple B(John, Toh, 18, 500) and
- Tuple C(John, Doe, 18, 600)
```

Based on the three extracted attributes `A` is equal to `C`, `A` is not equal to `B` and `B` is not equal to `C`.

2.5. Greedy Optimizer

The optimizer `GreedyOptimizer` uses the greedy heuristics to determine the plan to be executed. The optimizer first prepares the plan through the `preparePlan` method. In the method, the code is similar to the `prepareInitialPlan` method in `RandomInitialPlan`, but the major difference lies in the `createJoinOp` method.

The `joinSelected` array keeps track of the joins that are chosen by the optimizer. The first loop of the method runs through each join in the `joinList` that is generated from the `SQLQuery` object. The current join that has already been selected by the optimizer will be ignored. The `Join` operator will be created for each join, and then it will enter another `for` loop that sets the join type and calculates the plan cost of all the different join types of that `Join`. These two loops will execute and updates the `minCost` and keeps track of the join index, `tempJoinIndex` and join type index, `tempJoinMethodIndex` that computes the `minCost`. Essentially, the algorithm selects the join and join method with the lowest cost.

At the end of the two loops, `tempJoinIndex` and `tempJoinMethodIndex` will be passed to `modifyJoinOp` method where it creates a `Join` object with the minimum cost in the current iteration. The hashtable `tab_op_hash` is modified to reflect the changes. `joinSelected[tempJoinIndex]` is set to 1 so that in the next iteration, this index in the `joinList` will be ignored.

This process is repeated until all the `Condition` object in `joinList` has been selected. The `root` will be set to the final `Join` operator.

After the plan has been generated, the optimizer returns the plan to `QueryMain`.