UNIVERSITAT DE BARCELONA

Bases de Datos Avanzadas

4º Año

Assignment 2

Entregado a
Prof. de teoría:
Jerónimo Hernández González

Entregado por:
Alberto Barragán Verdejo 20222370
David López Adell 20150524

Índice

1. Processing pipeline

The first question we have to answer is if DBxic is using an eager or a lazy processing pipeline. To answer this question we will analyze the code.

```java
/**
 * getNextProcessedTupleList: the inner method to access tuples one by one
 */
@Override
protected List<Tuple> getNextProcessedTupleList() throws DBxicException {
    try {
        returnList.clear();
        if (tuples.hasNext()) returnList.add(tuples.next());
        else returnList.add(new Tuple(Tuple.tupleType.CLOSING));
        return returnList;
    }
    catch (Exception sme) {
        throw new DBxicException("Error: couldn't read tuples
        from intermediate file.", sme);
    }
} // getNextProcessedTupleList()
```

As we can see in the code above, the method is pulling the tuples one by one. This is the typical scheme that a lazy processing pipeline would use.

```java
/**
 * getNextOutTupleList: Basic function that serves
 * (sets of) tuples after being processed and produced
 * according to the current operator. Set ups and
 * closes the tuple processing system, when needed.
 */
public List<Tuple> getNextOutTupleList() throws DBxicException {
    // we need to call setup the first time the operator is requested a tuple
    if (firstGetNext) {
        setup();
        firstGetNext = false;
    }

    List<Tuple> t = getNextProcessedTupleList();
    while (t.size() == 0) {
    // keep asking until we obtain a non-empty result
        t = getNextProcessedTupleList();
    }
    if (t.size() == 1 && t.get(0).isClosing()) {
        // clean up if we reach the end
```

```
            cleanup();
        }
        return t;
    } // getNext()
```

Also in this second function, where the prior one is called, we can see in the top comment that it's only being used when we need the tuple processing system.

If we explore more the previous class we can end up in PhysicalOperator, where it's stated that Ït masks through a buffer the possible internal multiple-tuple processing to be able to serve tuples one-by-one.". This is remarked by the use of the UnaryOperator class which states that "Models a unary engine". This means that each input, as there's no output method, is done one by one.

2. Query optimisation

In this second question we are told to search if we can find any query optimisation. In order to do so we have searched first in Query class, where we found the next piece of code.

```
    /**
     * execute: returns the plan built by the
     * PlanBuilder for this algebraic operations list
     */
    @Override
    public Pipe execute(DBMS engine) throws DBxicException {
        PlanBuilder pb = new PlanBuilder(engine.catalog, engine.storManager);
        String file = FileHandling.getTempFileName();
        return (Pipe) pb.buildPlan(file, algebra);
    }
```

In this method we can find a class called PlanBuilder, where we can find the query evaluation system. Here we find a method called buildPlan where, first, it decomposes the operators into scans, projections, selections and joins. Later on it builds the plan as shown in the next code fragment.

```
        // 2) Now, we can start building the plan
        // 2.1) Build the projections that can be carried out from the beginning
        List<Projection> init_projections = buildInitialProjections(operators, tables);
        // 2.2) Build the relation scans
        List<PhysicalOperator> curLOps = buildScans(tables);
        // 2.3) Place the initial projections right after the relation scans
        curLOps = pipeInitialProjectionsToScan(init_projections, curLOps);
        // 2.4) Append the relevant selections over each scan
        curLOps = pipeSelections(selections, curLOps);
        // 2.5) order the joins and cartesian products in a tree
        curLOps = cascadeJoins(joins, curLOps);
        // perform sanity check and impose the final projections
        if (curLOps.size() != 1) throw new DBxicException("Error: Multiple branches
        after join enumeration.");

        // 2.6) establish the root of the plan tree
        PhysicalOperator root_operator = curLOps.get(0);
        // 2.7) append the final projections, if any
        if (!projections.isEmpty()) root_operator =
        pipeFinalProjections(projections, root_operator);
        // 2.8) append the final sorts, if any
        if (!sorts.isEmpty()) root_operator = pipeSorts(sorts, root_operator);
        // 2.9) append the final groups, if any
        if (!groups.isEmpty()) root_operator = pipeGroups(groups, root_operator);
```

Here we can check that the buildInitialProjections method is the one taking most of the burden when optimising the queries.

```
    /**
     * buildInitialProjections: returns the list of projections that can be carried out f
     * given a list of algebraic operators, for a specific set of tables. That is, all th
     * won't be output and are not required for any operation
     */
    protected List<Projection> buildInitialProjections(List<AlgebraicOperator> operators,
                                                        Set<String> tables) throws DBxicEx

        List<Projection> initProjections = new ArrayList<Projection>();

        for (String table : tables) {
            Set<Variable> listVars = getAttributes(table, operators); // this are the var
            // We won't add an initial projection if, after all, it won't help to reduce
            if (listVars.size() < catalog.getTable(table).getNumberOfAttributes()) {
                initProjections.add(new Projection(new ArrayList<Variable>(listVars)));
            }
        }

        return initProjections;
    } // getInitialProjections()
```

3. Buffer replacement strategy

It uses Least Recently Used as stated in the next code:

```
/**
 * getBlock: returns a block given its identifiers, moving it to the back of the repl
 */
public Block getBlock(String name, int number) {
    Pair blockid = new Pair(name,number);
    int index = getIndex(blockid);
    if (index >= 0) {  // if it exists, we mark it as recently used by moving it to t
        lReplacement.remove(blockid);
        lReplacement.add(blockid);
        return blocks[index];
    }
    return null;
} // getBlock()
```

Here we can see that when a block has been used it's moved to the last position in the queue. And in the next code section we can see that the least recently used block, the one in the first position is the that's going to be deleted.

```
/**
 * indexToEvict: returns the block to be evicted from the buffer to make room.
 */
protected int indexToEvict() {
    Pair blockid = lReplacement.removeFirst();
    return idToIdx.get(blockid);
} // indexToEvict()
```

4. Using constraints

The first thing that we have to add is a new function in the parser so it's able to read the constraint that we want to introduce.

After we have the valid constraints that we want to be used in the parser we'll have to update the Attribute variables, adding a constraint variable.

```
/**
 * Constructor: creates an attribute with a name and a type
 */
 public Attribute(String name, String table, Class<? extends Comparable> type, Class<
     this.name = name;
     this.table = table;
     this.type = type;
     this.constraint = constraint;
 } // Attribute()
```

With that done now the Attribute will have the constraints, but in order to enforce them we have to check that the values that we add are compliant with the constraint that we set before. If the value added, or modified, is not within the expectations of the constraints it should send an error back to the user using the database.

To enforce the values, we'll have to modify the TupleInsertion and TupleModifier to make them check the constraints before adding a value.

This could be done too in the StorageManager class, but it would have to be hardcoded, giving us less liberty to add the constraints.

5. Extra questions

5.1. How could we access directly to the block containing the modified tuple given its position?

In the code we find the position with a while loop. Instead of that we could use a formula to get the position and the block number:

```
int blockNumber = TuplePosition / BlockSize
int inBlockPosition = TuplePosition % BlockSize
```

However, this equation assumes that we can now the size of the Blocks and that it is a static size. But the last block could be smaller because it can be partially empty.

5.2. What does it change if we do not assume that att_i is a key?

First, we're not assuming that with this question you want to change more than one tuple at the same time because all the tuples to be modified pass the proposition. If it was that, then it should have been specified that att_i is a primary key. This is done in our code as a proof of concept.

If the question is that att_i is not an ATTRIBUTE, then the proposition would be pretty absurd, as it only could be True or False, as in $4\bar{3}$ or $1\bar{1}$, for all the tuples that we check.