Database Management Systems Implementation

Report Lab2: SimpleDB

Massimiliano Pronesti
Federico Tiblias
Giulio Corallo

Instructor: **Paolo Papotti**

September 7, 2022

1 Design and Implementations

In this section, we're going to briefly describe our main design and implementation choices, especially focusing on the aggregations, the page eviction approach and the join policies.
No change was made to the provided APIs of SimpleDB.

1.1 Aggregation

As regards the aggregations, we employed four HashMaps (see class IntegerAgregator), in order to store the results of the operations: counts, sums, averages and minmax. The latter one is employed to store the minimum, if the operation is MIN or the maximum if the operation is MAX. Whenever the "mergeTupleIntoGroup" is called, given a previously defined operator, the corresponding map is updated, except for the AVG operator, where both counts and sums are updated. Besides, we implemented the methods "getTupleDesc", which creates a TupleDesc according to the type and the name of the field of the new tuple to merge on the aggregate, and "getTuples" which loops over the corresponding map and return a list of tuples.

The StringAggregator class follows the same approach of IntegerAggregator class, but it only supports the COUNT operator.

5Eventually, the class Aggregate is used as an interface: depending on the type of the aggregate, i.e. INT_TYPE vs STRING_TYPE, it constructs an IntegerAggregator or a StringAggregator rispectively.

1.2 Join

Our Join policy consists of a general case, where a nested-loop approach is employed, and the specific case of the equality join, where we perform an HashJoin to increase the efficiency of the process.

As regards the NLJ, we iterate over the different tuples using two nested loops for the two different children, and merging them in a single tuple which is returned. On the other hand, the HJ has been implemented on top of the HashEquiJoin class already provided by the author of SimpleDB. An initial check on the operator of the join predicate is employed as discriminant for the join approach.

1.3 BufferPool

A helper class has been implemented that acts a queue, where least-recently used pages get returned to main BufferPool class to perform eviction. PageBufferPool is based on the proposed solution for lab1 and implements the concept of memory: all pages contained in PageBufferPool are considered as loaded in memory. Flushing simply copies dirty pages to disk. If a page is clean, the actual copy is avoided. Eviction is performed by getting the last element from the PageBufferPool, copying it to disk and removing it from the queue.

1.4 Insert and Delete

The implementation is pretty straight-forward, the only significant design choice was to execute the insertion/deletion in the fetchNext method. Please notice that the returned tuple has null as column name as doing otherwise caused some assertion failures.

2 General comments

In this section, we're going to describe the time spent in this lab, our split of the work as well as our subjective evaluation of the effort in terms of difficult and/or confusing aspects of the requirements.
The work was split equally, following the suggested order of the implementations and employing a github workflow to make sure the provided tests actually passed on a gh-provided runner. The entire work took overall, approximately, 7h per person.
Broadly speaking, we encountered no particular setbacks even if it was definitely more challenging then the first one and it required discussing and revising every step from all the members of the group. All the

code developed for lab1 was maintained for this assignment, with the exception of the class PageBufferPool described in **1.3**.