CS 143: Summary of lab 2

Zhehao Wang 404380075

Feb 2015

1 Design and implementation decisions

The implementation is based on that of lab 1's, and this sections goes over the design and implementation decisions made specifically for lab 2.

- Page eviction policy: our implementation evicts a random page when the BufferPool is full. Note that *pin_count* is not explicitly in the implementation. JVM garbage collector's reference count can be considered as similar in the sense that objects being used will not get recollected.

- Join method: this implementation uses a basic nested loop join.

- Iterators, and when the operator action happens

  This lab includes the implementation of several operators, each as a subclass of the parent *Operator*. *Filter*, *Join*, *Insert*, *Delete*, and *Aggregate* are all such examples. Each of these provides an *Iterator* to the result of the operation, and decisions need to be made regarding when the action, for example, an actual *Join* should happen; Specifically: whether it should be done before the first call to a *fetchNext*, or it should happen in a *fetchNext* call, in which children are iterated as needed.

  For *Filter* and *Join*, our implementation uses the latter approach; for *Aggregate*, the former approach's used: the children are traversed upon an *open* call, which calls the *iterator* of an underlying *IntegerAggregator* or *StringAggregator* that generates the aggregated tuples, and its *fetchNext* merely brings back the next item from the aggregation, if it's already calculated. *Insert* and *Delete* are different, as *fetchNext* is supposed to be called only once, during which all results are calculated. The rationale of having the difference between the first two categories is that we want the evaluation of *Operator* to be as lazy as possible, in case the user does not need all the results (for example, having a LIMIT clause), however, in the case of *Aggregate* where each resulting tuple's only available after a scan through all the tuples (and GROUP BY can happen), it's likely more efficient to generate the results at once than traversing the children each time *fetchNext* is called.

2 Changes made to the API

This implementation does not make any changes to the API. Making *Operator*'s *opened* attribute was considered at first, so that classes inheriting from *Operator* could set the flag in their *open* methods, but a *super.open* call is used instead.

3 Summary

Approximately 24 hours were spent on this lab.

This update to lab 1 code fixed a *Tuple* constructor that uses a fixed length to initialize the fields array. The fixed length was an oversight, and will cause problems if tuples with a large amount of columns are used when testing.

The biggest confusion we had was the expected behavior of the *open*, *close*, and *rewind* methods of the child classes of *Operator*. Given that each has their own resulting iterator, and some children iterator, we are confused whether these calls are meant to operate on their own iterator, or their children's iterator as well. The unit test cases always pass in opened children iterators, so the former's assumed at first, based on the understanding that these methods want to mess as little as possible with the states of the children iterators, since they may be shared among other instances. However, system test cases pass in unopened children iterators, and expect the *Operator* to work without opening their children iterator separately. Thus we concluded that the latter's the expected approach, and our implementation has the caveat that the iterators passed in *Operator* constructors should not be shared between different instances, or otherwise modified during the *Operator*'s operation.

The second confusion lies with the example query interface, which we raised on Piazza. Given the example setup, projection by column name does not work because of the *TupleDesc.fieldNameToIndex(name)* implementation cannot match the given string with column names. Given string contains table aliases, but *TupleDesc*'s not aware of them. Either the caller of *fieldNameToIndex* or the function itself should strip the table aliases. We figured it's more reasonable for the caller in *LogicalPlan* to do it, since *fieldNameToIndex* cannot distinguish a column name with dots from a table alias. Since *LogicalPlan* is not part of this lab, its implementation is left as-is. Thus, the simple query interface will not work if column names are involved (for example in SELECT, GROUP BY, or WHERE clause).