CSE 444: Lab 2 Writeup

Linxing Preston Jiang Winter 2018

February 4, 2018

---

1. Query runtime

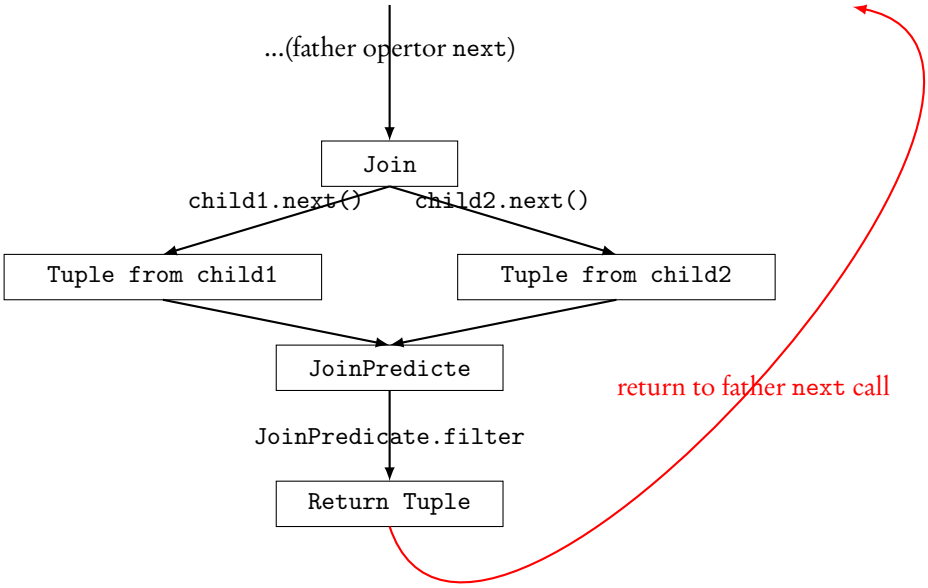
- Query 1: 1.53 seconds (`attu4`)
- Query 2: Because of nested-loop-join implementation, Query 2 takes forever to finish (did not finish after an eight hour tmux session on `attu4`)
- Query 3: Same as above

2. Lab 2 consists two major parts: operators & aggregates, and database mutability. For the first part, we implemented `Filter`, `Join` operator and common aggregates such as `MIN, MAX, AVG`. For the second part, we focused on the implementation of inserting/deleting tuples to and from the database, also the eviction of pages in `BufferPool` when it's full. Key components implemented are:

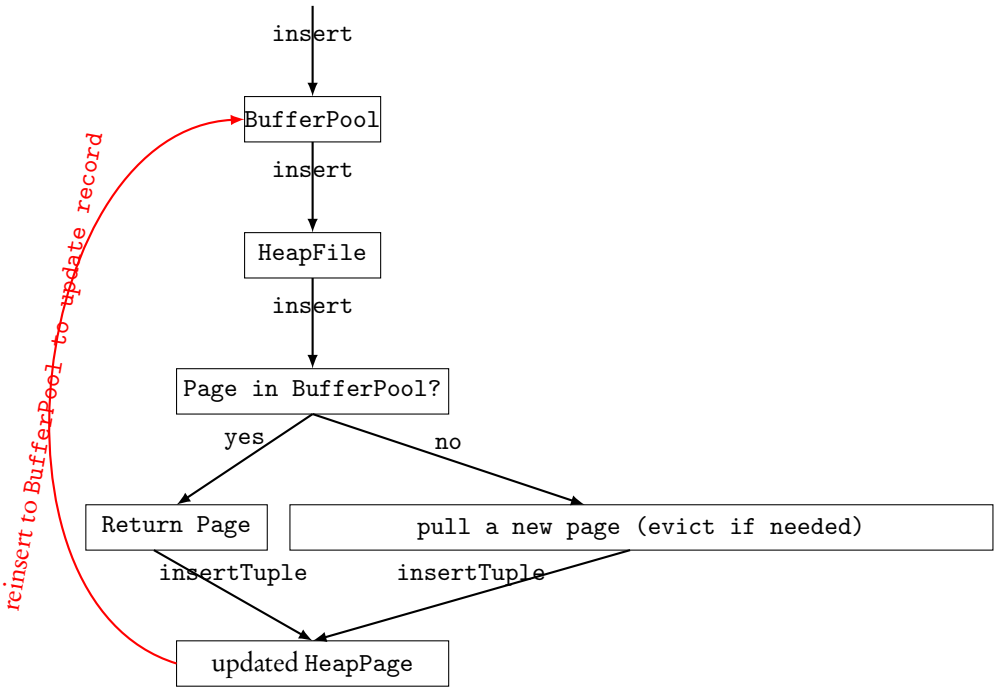
- `Predicate`: this defines a predicate (comparision) which is needed for operators such as `Filter` and `Join` for tuple and field comparisons. `JoinPredicate` is designed directly for `Join` operation.
- `Join`: The class represents the join operation for relational database. The `Join` operator takes in tuples from two child iterator and use `JoinPredicate` to filter results.
- `Aggregate`: this represents an aggregation clause, can be on `Integer` or `String` fields. `Aggregate` loads all tuples from child at `open()` and returns the aggregate results one group at a time in `next()`.
- `HeapPage`: We added `insert`, `delete` in it, which insert/delete tuples to/from the `tuple` array and mark the corresponding slots to used/free. `HeapPage` directly inserts/deletes tuples to/from the `Tuple` array which will be the data updated on disk.
- `HeapFile`: We added `insert`, `delete` in it, which obtains the page from `BufferPool` (Or add a new empty page when added) and calls the `insert/delete` from `HeapPage`. Also, we added `writePage` method which is needed later for `BufferPool` to write the data on dirty `HeapPages` to disk.
- `BufferPool`: We added `insert`, `delete` in it, which insert/delete tuples to/from the pages in BufferPool by calling the method from `HeapFile` to update the page, then `BufferPool` updates the records by re-inserting the pages into the `BufferPool`. When the `BufferPool` is full and a new page need adding, `writePage` from `HeapFile` will be called to write the dirty page to disk and add the new page into the `BufferPool`. `BufferPool` is in charge of updating the pages because `Insert/Delete` operators directly call `insert/delete` of `BufferPool`.

Examples:

Workflow (`join()`, showing `next()` call):



Workflow (`insertTuple()`):



3. Design decisions: I chose to use nested loop join. Nested-loop-join is the slowest but it works for all kinds of comparisons (Hash join only works well for equality comparison, sort-merge join only works well for inequaliy comparison). For eviction policy, I chose to evict the first page every time the `BufferPool` is full. This does not work well in terms of keeping most commonly used page in memory, which result in extra disk IOs, but the implementation is easy and does not require extra data structures in order to keep track of page usage.

4. Extra Unit tests: A unit test for the correctness of HeapFile insertTuple behavior would be great. The spec requires the update of `BufferPool` pages to happen in the insert/delete methods of `BufferPool` instead of in `HeapFile`. I did it the wrong way and caused the `handleManyDirty` test to fail, and it took me a while to realize that test uses a overriden `insert` method and `BufferPool` should be in charge of updating the record.

5. Changes to API: I did not make changes to the APIs given.

6. Missing elements: I believe I finished the entire Lab 2