1. Query Runtime

- Query1 : Completed in 0.20 seconds
- Query2: Got 10 tuples in around 20 minutes, ran till one hour but didn't complete.
- Query3: Ran till an hour but didn't complete.

2. Components

Predicate: Predicate encapsulates the comparison logic that is applied to the tuple fields. It is used in the selection operation to check whether the tuple meets specific criteria.

JoinPredicate: It is similar to the 'Predicate,' but this is created to check conditions containing two tuples (from different relations). When used in join operations, it helps determine if two tuples from the respective relation satisfy the join condition.

Filter: The filter takes a predicate and a child operator as inputs, then processes the tuples from the child and passes only those tuples that satisfy the predicate's conditions.

Join: The join operator takes two child operators and a JoinPredicate. I implemented it using a nested for loop, which returns a pair of tuples that satisfy the constraint. So, when the fetchNext() is called on the operators, it returns a tuple at a time OR a pair of tuples for the join operator.

Aggregate, Integer Aggregator, String Aggregator: The Aggregate operator manages aggregation (Sum, Min, Max, Group By, etc..) on the tuples from its child operator. It consists of IntegerAggregator and StringAggregator. The IntegerAggregator handles numerical aggregation on integer fields. The group by is also taken into consideration to ensure if there is a grouping, then the aggregate is computed with grouping. The String Aggregator is specifically for counting operations on the string field, as the string does not support numerical aggregations like average and sum. The Aggregate class takes care of which aggregator to call; it either leverages String or Integer Aggregator and returns the results accordingly.

BufferPool, HeapPage, HeapFile, Insert, Delete: When users want to delete a tuple, they call the deleteTuple method in the BufferPool. Then, the BufferPool calls the deleteTuple method on the heap file that the tuple belongs to. Then, HeapFile calls the deleteTuple method on the HeapPage. HeapPage deletes the specified tuple from the page, and the header bit gets updated to reflect that the slot in which the tuple was stored is now available. When users want to insert a tuple, they call the insertTuple method in the BufferPool. Then, the BufferPool calls the insertTuple method on the respective heapFile. Then, HeapFile calls the deleteTuple method on the respective HeapPage. HeapPage inserts the tuple in the page, marks the slot as used, and

updates the record ID of the tuple. If the page we are writing in is not available in the file, then we add a new page to the file so that we can add a tuple to that page. When the insert/delete is used on the page, it is added to the BufferPool so that it can be retrieved from the disk before the eviction.

Insert Operator is used to add the tuples to the tableid, which it reads from the child operators. When the fetchNext is called, it calls the insertTuple method in the BufferPool. And it calls the insertTuple method in HeapFile and so on.. (Explained in the above paragraph). Delete Operator is used to delete the tuples from the tableid, which it reads from the child operators. When the fetchNext is called, it calls the deleteTuple method in the BufferPool. It calls the deleteTuple method in respective HeapFile and so on. (Explained in the above paragraph).

When we are retrieving a specific page from the BufferPool, if that page is not available in the BufferPool, we add it to the BufferPool and return it. But if there is insufficient space in the BufferPool to add a page, we evict a random page by calling evict(). Evict calls flushPage() on the ID of the random page and makes space for the page to be stored. FlushPage checks if the page has been modified during the existence of that page in BufferPool; if it is dirty, then it is written into the disk with all the content in the place where the previous page existed. Then, we remove the old page from the cache. This is how eviction works.

3. Design Decisions

The page evection policy we implemented removed a random page from BufferPool. We know it's less efficient than other strategies like "Least Recently Used (LRU)." However, implementing, understanding, and computing costs is easier than with different strategies. As stated in question 2, we have used nested loop join; we have also made a design decision in the join operator: we generate the next tuple on-demand rather than precomputing and storing all the possible joined tuples, which gives better performance for queries that do not require all join results.

4. Unit Test

A unit test that could we add to improve the JoinTest.java, would be to "test join with empty inputs" This test will verify that the behaviour of the Join operator when one or both child operators have no tuples. By adding this unit test in the JoinTest.java, would make the join operator more robust, by covering every edge cases.

5. API Changes

No changes have been made to API

6. Feedback

No Feedback.