Report of Lab 4

Yuxuan Duan 516030910573

1 Design Decisions

1.1 About HashEquiJoin

Since I implemented JoinOptimizer in Lab 4, I removed the special judge in the class Join which switched to HashEquiJoin if the operation type is EQUALS, as in Lab 3. Instead, in JoinOptimizer.instantiateJoin(), I made a special judge. If the operation type is EQUALS, a HashEquiJoin instance will be returned instead of a natual Join instance. I have done the query-parsing tests in Lab 3, and it indeed uses HashEquiJoins and has good performance.

Also, my JoinOptimizer.estimateJoinCost() considers using HashEquiJoin, and the cost is

$$cost1 + cost2 + card1 \times card2$$

because in HashEquiJoin, the two tables are read from disk only once, so the I/O cost is $cost1 + cost2$. Then, because memory cost is far less than I/O cost, so if $card$ is linearly showed up in the total cost (i.e. only $const \times card$ but not $card \times card$ or $cost \times card$), we can ignore it. So I took the memory cost as $card1 \times card2$.

1.2 About Join Cardinality

The estimated cardinalities are showed below.

is primary key	EQUALS	NOT_EQUALS	other operations
t1 & t2	$\min(card1, card2)$	$card1 \times card2 - min(card1, card2)$	$0.3 \times card1 \times card2$
t1	$card2$	$card1 \times card2 - card2$	$0.3 \times card1 \times card2$
t2	$card1$	$card1 \times card2 - card1$	$0.3 \times card1 \times card2$
neither	$\max(card1, card2)$	$card1 \times card2 - \max(card1, card2)$	$0.3 \times card1 \times card2$

Here is the explanation of the above tabular when the operation type is EQUALS.

- If both t1 and t2 are primary keys, the join will be a one-to-one matching between t1 and t2, so the number of tuples of the outcome is at most the smaller one of the cardinalities of the two tables.
- If t1 is primary key but t2 is not, a tuple of t2 can match at most one tuple of t1, so the outcome is card2. The situation of t2 is primary key is similar.
- If neither of the two is primary key, we heuristically suggest that the outcome is $\max(card1, card2)$.

If the operation type is NOT_EQUALS, the outcome is just $card1 \times card2$ minus the outcome of the corresponding primary key situations. And if the operation is a range operation, we take it as $0.3 \times card1 \times card2$.

1.3 About Selectivity Estimation

I did not do major special design on selectivity estimation in my code except the LIKE operation. LIKE of strings is a special one and I did not figure out a good way to estimate the LIKE selectivity. So heuristically, I take the selectivity of a LIKE operation is 0.1.

1.4 About Join Ordering

When implementing the join ordering algorithm, I changed the 5-th row of the pseudo-code of the document, which enumerates all length $|s|-1$ subsets s' of s, to enumerating the elements s_1 in s. Because we only need s and $s_1 = s - s'$ to do the following procedures, this could hopefully save time.

2 API Changes

2.1 DbFile.numPages()

I added this interface in DbFile to provide a more convenient access for the class TableStat. It should cause no error since numPages() had already been a method of both HeapFile and BTreeFile.

3 Missing or Incomplete Elements

3.1 Average Selectivity

I did not implement IntHistogram.avgSelectivity() because I did not see much significance in this method. I think that at least an operation type should be provided to make it significant statistically.

4 Time Spent and Difficulties

I roughly spent 12 hours on this Lab. Compared with the previous three Labs, Lab 4 has more things to explore since the methods we implements in this Lab have no fixed ways. The key difficulties in this Lab, I think, are how accurate our estimations should be. Although a wide range of estimation, from a strict one to a loose one, can all pass the tests, I think we should balance between accuracy and calculation complexity. Thus, I discussed much with my classmates, and it was fun.