Jordan Fanapour and David Schwartz
Professor Stavros
CS 580
04/28/24

Project Writeup

Problem 1

Our algorithm is implemented in the function “naturalJoin” in relations.cpp. The pseudo-code for the algorithm is as follows:

```
naturalJoin(relation this, relation other):
      res := new relation to hold the results
      Add all attributes in this and other to res
      Build a hashmap H for relation other
            // H maps a value (from a shared attribute)
            // to tuples it is involved in for the “other” relation
      For each tuple t in this relation:
            For each tuple t’ in H(t):
                  If all shared attributes are equal:
                        Add t merged with t’ to res //the join
      return res
```

The key to implementation is the creation of the hash map H, which is a slightly more generalized version of the definition given in the problem statement. Our code finds all shared attributes between two relations and then uses the first shared attribute as the key for the hashmap. The definition for the contents of the map is the same as in the problem – values of the key attribute are mapped onto tuples the value occurs in.

This generalization leads to our code having an additional check. We check if all shared attributes are equal before adding a new tuple to the result. For this problem, checking the hashmap would have been sufficient as the relations only share one attribute. For generality, we did the check so our code could handle joins with multiple shared attributes/variables.

Our implementation is correct because it checks all possible pairs of tuples that can feasibly be joined together with the help of the hash map. Additionally, we will never merge two tuples where the values of their shared attributes are not equivalent. This means that our implementation finds all tuples that should be merged for the output of the query and never anymore. The following example is the full join between two relations with 10 tuples each and we can verify by hand that the full join is correct:

Output for experiment 1:

Relation R1:

A	B
1	2
4	5
7	8
10	9
1	3
2	2
11	3
6	7
11	5
12	15

Relation R2:

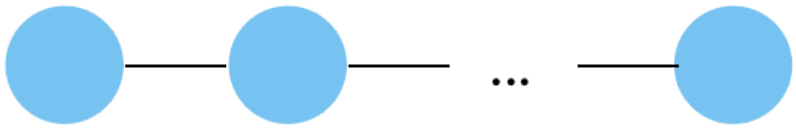
B	C
2	4
5	2
7	8
13	9
1	3
5	9
15	6
3	7
3	5
9	4

Join between relations R1 and R2:

A	B	C
1	2	4
4	5	2
4	5	9
10	9	4
1	3	7
1	3	5
2	2	4
11	3	7
11	3	5
6	7	8
11	5	2
11	5	9
12	15	6

Problem 2

Our code for the simplified implementation of Yannakakis algorithm is in the function “executeLineJoin” in relations.cpp. The algorithm involves creating and traversing a join tree. However, because we are focused on line joins, the tree is very simple:



Because the tree contains no branches, we can represent it as a list of the relations. This makes the traversal code a simple for loop. We use index 0 as the root. With that said, the pseudo code is as follows:

```
executeLineJoin(list of relations RS):
      pruned := a list of relations, with the same size as RS
      n := |RS|-2
      //remove dangling
      For i=n down to 0: //bottom up
            pruned[i] = RS[i] ⋉ RS[i+1]
      For i=0 up to n: //top down
            pruned[i] =  pruned[i] ⋉ pruned[i+1]

      //Compute final results bottom up
      For i=n down to 0:
            pruned[i] = pruned[i] ⋈ pruned[i+1]

      //results in “root node”, defined as 0
      return pruned[0]
```

The variable *pruned* is a list of relations, which eventually stores the relations without dangling tuples. We opt to assign relations to *pruned* in the first for loop to prevent wasting time copying all the relations before processing them.

The full join used in the algorithm is the “naturalJoin” implementation from problem 1. The semi-join implementation is a full join followed by a projection. This was done to reduce the amount of code we wrote – the principle was less code and fewer bugs. We could optimize this to go faster. This decision may have led to our Yannakakis implementation being slower (than other methods) in later problems.

Finally, as per the algorithm’s definition, the results accumulate in the root node, which we defined earlier as element 0 in the list. To check correctness, we compared the results of our Yanakakis implementation with the results obtained from a series of full joins over the relations.

Problem 3

The code for this algorithm is in the function “executeLineJoinByChaining” in relations.cpp. The pseudo-code is very simple:

```
executeLineJoinByChaining(list of relations RS):
      If |RS| == 0 //base case, empty instance
            Return empty relation
      If |RS| == 1 //base case, 1 relation
            Return RS[0]
      res := RS[0]
      For i = 1 up to |RS|
            res = res ⋈ RS[i]
      return res
```

It’s just a for loop over the list of input relations, with results accumulating in *res*. As before, the full join is our “naturalJoin” function.

Problem 4.

The following is the output of our program, with seed = 93483235:

```
Time taken for line join (Problem 2): 2002 microseconds.
Time taken for line join by chaining (Problem 3): 465 microseconds.

The two methods of executing the query produced equivalent results.

Results of the line join query.
A  |B    |C  |D
----------------
1  |4689 |97 |97
30 |171  |50 |50
57 |4095 |7  |7

— rest omitted because it reprints the results –
```

We added code to compare the results from both queries automatically and warn us if they ever differ. As the output states, the algorithms yielded the same solutions.

The more interesting facet is that the simple full join chaining algorithm ran 4.3x faster than our implementation of Yannakakis algorithm. Our sample program ran the experiment once, but rerunning it continued to yield similar results.

This result is surprising to us. There are very few tuples in the solution, meaning many of them are dangling. As such, the Yanakakis algorithm should have pruned many tuples away, especially from relations 1 and 2 because their common attribute had a random chance of matching. Nonetheless, it still had a much larger runtime.

We have come up with a few reasons as to why. First, our implementation of semi-join is not optimized. It performs a full join and then a projection, meaning that it takes longer to compute than a regular full join. As a result, the pruning phase of Yannakakis algorithm takes longer than it should. Second, our implementation does not operate on relations in place. The full join, semi join, and projection operations we implemented return a new relation with its copy of the data. Yanakakis algorithm makes use of more operations than the chained full joins, so we may be losing time due to memory allocation/frees.

Problem 5.

Part of the output from our program is provided below. All of the excluded results were of the same form: (i, 5, 8, y), for some integers i and y. To get a full listing, rerun our provided code. The snippet of output was generated with seed = 93483235.

```
Time taken for line join (Problem 2): 1.34554e+07 microseconds.
Time taken for line join by chaining (Problem 3): 1.16276e+07
microseconds.
The two methods of executing the query produced equivalent results.
Results of the line join query.
A    |B    |C |D
------------------
684  |5    |8 |30
546  |5    |8 |30
511  |5    |8 |30
97   |5    |8 |30
956  |5    |8 |30
…
2001 |2002 |8 |30
…
469  |5    |8 |30
944  |5    |8 |30
130  |5    |8 |30
420  |5    |8 |30
…
```

For this problem, our implementations performed more similarly – full join chaining was only 1.16x faster than our Yanakakis implementation. This problem shows that Yannakakis algorithm scales better as the size of the database instance increases. This is to be expected as the runtime of Yannakakis algorithm is O(N + OUT), whereas the runtime of the chained full joins is $O(N^3)$ when dealing with three relations. This is especially noticeable for this problem as the definitions for R_1 and R_2 are designed to yield many intermediate join results.

We listed a few reasons why our implementation of Yanakakis could be slower in problem 4. However, those issues may be insignificant concerning the scaling of the algorithm's performance, at least when compared to our simple alternative.

Once again, the output for the queries was equivalent.