Assignment 4 - Solutions
CS44800 Staff
April 8, 2021

1 Pipelined Execution

1. \[
\text{relation\_it} = \text{relation\_open} ("\text{instructor}\") \\
\text{select\_it} = \text{select\_open} (\text{relation\_it} , \text{name} == "\text{Clifton}\") \\
\text{project\_it} = \text{project\_multi} (\text{select\_it} , ("\text{name}\" , "\text{salary}\")} \\
\text{formatted\_tuples} = \text{array[]} \text{ or list}[] \\
\text{while (not project\_it\_.done())}{ \\
\quad \text{formatted\_tuples\_.push(project\_it\_.next())} \\
\text{return formatted\_tuples}
\]

2. use \text{project\_dedup} instead of \text{project\_multi}. Or change \text{array[]} or \text{list}[] to \text{map}{} or \text{set}{}.

3. key: same results. not key: allow duplication which means you may have different results. Also, discussion on unique index (primary key) vs Index is also acceptable.

4. There is no I/Os difference since both queries need to read the same amount of data. There may have small differences in memory usage since \text{Project\_dedup} does not have duplicated records while \text{Project\_multi} tends to consume more memory.

2 Join Processing

The smaller relation can be stored in memory while the larger relation is read block by block. Perform nested-loop join using the larger one as the outer relation. The number of I/Os is \(b_r + b_s\) and the memory requirement is \(\min(b_r, b_s) + 2\) pages.

3 Hash Outerjoin

We can modify the hash join algorithm shown in Figure 15.10 of the book as follows:

1. For each tuple \(t_r \in H_r\), if there is no matching tuple \(t_s \in H_s\) then add \(t_r\) and pad with null values.
2. For each matching tuple \(t_s \in H_s\), set a flag to True that indicates it was matched.
3. Do a final pass on tuples \(t_s \in H_s\), if the flag set to False then add \(t_s\) and pad with null values.

4 Merge Join

With three memory blocks, two are used to hold input and the third one is used for output.

We name the tuples (kangaroo, 17) through (baboon, 12) as \(t_1\) to \(t_{12}\). First we create sorted runs: \(r_{11} = \{t_3, t_1, t_2\}\)
5 Merge Join

5.1

Algorithm 1: MergeSortJoin.open()

```plaintext
1 begin
2   r.open()
3   s.open()
// sort relations
4   sort(r)
5   sort(s)
6   if r.next() ≠ False and s.next() ≠ False then
7       move tuple r buffer to t_r
8       move tuple s buffer to t_s
9       mark_s ← null
```

Algorithm 2: MergeSortJoin.close()

```plaintext
1 begin
2   r.close()
3   s.close()
```
Algorithm 3: MergeSortJoin.next()

1 begin
2     while s.next() ≠ False and r.next() ≠ False do
3         if markₕ = null then
4             while ts[attr] < tr[attr] and s.next() ≠ False do
5                 move tuple s buffer to ts // advance s
6             while tr[attr] < ts[attr] and r.next() ≠ False do
7                 move tuple r buffer to tr // advance r
8                 markₕ ← ts
9             if ts[attr] = tr[attr] then
10                move ts ⊲ tr to output buffer
11                if s.next() ≠ False then
12                   move tuple s buffer to ts
13                return True
14         else
15             if r.next() ≠ False then
16                ts ← markₕ // rewind s
17                move tuple r buffer to tr
18                if ts[attr] < tr[attr] then
19                   markₕ ← null
20             return False

5.2

In open(), only sort r and skip the final merge. In next(), we pipeline s and compare each tuple ts with the smallest tr in the runs of r. We only have to pass through s once and we can start returning joined tuples earlier.

6 Join Pipelining

With indexed nested loop join, it will almost always get you the next tuple fastest (although it might be faster to first look in whatever blocks you have in memory to see if there is a match first). But it isn’t the fastest to the last tuple (it may need more disk I/Os in the end if the index is unclustered as same disk page may be visited more than once), even though at any given point, it is the fastest way to get to the next tuple.

7 Query Equivalence

Solution courtesy a student who wishes to remain anonymous.
7. \( \Pi_{\text{branch\_name}}((\Pi_{\text{branch\_name}, \text{assets}}(\rho_T(\text{branch}\_T))) \bowtie_{\text{assets}} S.\text{assets}
(\Pi_{\text{assets}}(\sigma_{\text{branch\_city} = \text{"Brooklyn"}}(\rho_S(\text{branch}\_S)))))) \)

By restricting the amount of data in the relation tuples that are being joined, this is an efficient relational algebra expression. Since the select is done on branch S before the join, there is no need to select after the join is performed. Additionally, all unused attributes on both relations are excluded before the join, making it as efficient as possible.