CS3223 Tutorial 4: Projection & Join

Wk 6, Sem 2, 2022/23

Questions to be discussed: 1, 2 & 3.

1. (Adapted from Exercise 14.3, R&G) Consider processing the following SQL projection query:

   SELECT DISTINCT title, dname FROM Executives

   You are given the following information:

   - `Executives` has attributes `ename, title, dname`, and `address`; all are string fields of the same length.
   - The `ename` attribute is a candidate key.
   - The relation contains 10,000 pages.
   - There are 10 buffer pages.

   Consider the optimized version of the sorting-based projection algorithm: The initial sorting pass reads the input relation and creates sorted runs of tuples containing only attributes `dname` and `title`. Subsequent merging passes eliminate duplicates while merging the initial runs to obtain a single sorted result (as opposed to doing a separate pass to eliminate duplicates from a sorted result containing duplicates).

   In this question, the cost metric is the number of page I/Os.

   (a) How many sorted runs are produced in the first pass? What is the average length of these runs? What is the I/O cost of this sorting pass?

   (b) How many additional merge passes are required to compute the final result of the projection query? What is the I/O cost of these additional passes?

   (c) (i) Suppose that a clustered B^+-tree index on `title` is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

   (ii) Suppose that a clustered B^+-tree index on `dname` is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

   (iii) Suppose that a clustered B^+-tree index on `(dname, title)` is available. Is this index likely to offer a cheaper alternative to sorting? Would your answer change if the index were unclustered? Would your answer change if the index were a hash index?

   (d) Suppose that the query is as follows:

   SELECT title, dname FROM Executives

   That is, you are not required to do duplicate elimination. How would your answers to the previous questions change?

Solution:

(a) Each sorted run will have at most $B = 10$ pages of records. Since the optimized algorithm performs projection as part of the creation of initial sorted runs, the 10,000-page relation can actually fit into 5,000 pages after projecting out the unwanted attributes. Therefore, we assume that the number of initial sorted runs is $\frac{5000}{10} = 500$ runs; the size of each run is at most 10 pages. I/O cost for creating the initial sorted runs $= 10,000 + 5,000 = 15,000$ I/Os.

(b) Number of merge passes $= \lceil \log_9(500) \rceil = 3$. Total cost of merge passes $= 3 \times 2 \times 5,000 = 30,000$ I/Os.

(c) (i) Since the index key does not include all the required attributes, the data pages are only partially sorted (wrt *title*). However, we can exploit the partially sorted data to reduce the sorting cost. Suppose that there are k distinct title values in the Executives relation. Since the data records are sorted on *title*, the data pages form a sequence of k partitions $< P_1, P_2, \cdots, P_k >$, with the records in each P_i having the same *title* value. Instead of applying the optimized algorithm to the entire relation, we apply the optimized algorithm to each partition P_i separately to perform the projection. The total I/O cost for performing these k projections on the partitions is lower than the total I/O cost for performing a single projection on the entire large relation.

If the index was an unclustered B$^+$-tree, retrieving records with the same *title* value using the index would incur many random disk I/Os given that there are only 10 buffer pages and the relation consists of 10,000 pages. Thus, using the unclustered B$^+$-tree would not be cheaper than using a table scan to access the records.

If the index was a clustered hash index, the data pages form a sequence of partitions with each partition consisting of pages belonging to a hash bucket. Applying the optimized algorithm to each partition would be cheaper than applying the optimized algorithm to the entire relation.

(ii) Similar comments as (i).

(iii) Using a B$^+$-tree index on `(dname,title)` (whether clustered or unclustered) would be more cost-effective than sorting as the projection can be performed using an index-only scan by sequentially scanning the leaf pages of the index to eliminate duplicates.

For a clustered hash index on $(dname, title)$, the comments are similar to the case of a clustered hash index on $(dname)$.

(d) Knowing that duplicate elimination is not required, we can simply scan the relation and discard unwanted fields for each tuple. This is the best strategy except in the case that a format 2 or 3 index on `(dname, title)` is available; in this case, we can do an index-only scan.

2. (Exercise 14.4 R&G) Consider the join $R \bowtie_{R.a=S.b} S$, given the following information about the relations to be joined. The cost metric is the number of page I/Os unless otherwise noted, and the cost of writing out the result should be uniformly ignored.

   - Relation R contains 10,000 tuples and has 10 tuples per page.
   - Relation S contains 2000 tuples and also has 10 tuples per page.
   - Attribute b of relation S is the primary key for S.
   - Both relations are stored as simple heap files.
   - Neither relation has any indexes built on it.
   - 52 buffer pages are available.

   (a) What is the cost of joining R and S using a **page-oriented simple nested loops join**? What is the minimum number of buffer pages required for this cost to remain unchanged?

   (b) What is the cost of joining R and S using a **block nested loops join**? What is the minimum number of buffer pages required for this cost to remain unchanged?

   (c) What would be the lowest possible I/O cost for joining R and S using **any join algorithm**, and how much buffer space would be needed to achieve this cost? Explain briefly.

   (d) How many tuples does the join of R and S produce, at most, and how many pages are required to store the result of the join back on disk?

   (e) Would your answers to any of the previous questions in this exercise change if you were told that R.a is a foreign key that refers to S.b?

Solution: We have $|R| = \frac{10000}{10} = 1000$ pages, $|S| = \frac{2000}{10} = 200$ pages, and B = 52.

(a) The basic idea is to read each page of the outer relation, and for each page scan the inner relation for matching tuples. Total cost would be #pages in outer + (#pages in outer $\times$ #pages in inner), which is minimized by having the smaller relation be the outer relation. TotalCost $= |S| + (|S| \times |R|) =$ 200,200. The minimum number of buffer pages for this cost is 3 (one input for each of R and S, and one output page for the join results).

(b) This time read the outer relation in blocks, and for each block scan the inner relation for matching tuples. So the outer relation is still read once, but the inner relation is scanned once for each outer block, of which there are $\lceil \frac{\#pages\ in\ outer}{B-2} \rceil = \lceil \frac{200}{50} \rceil = 4$. TotalCost $= |S| + |R| \times \lceil \frac{|S|}{B-2} \rceil =$ 4,200. If the number of buffer pages is less than 52, the number of scans of the inner would be more than 4 since $= \lceil 200/49 \rceil$ is 5. The minimum number of buffer pages for this cost is therefore 52.

(c) The optimal cost would be achieved if each relation was only read once. We could do such a join by storing the entire smaller relation in memory, reading in the larger relation page-by-page, and for each tuple in the larger relation we search the smaller relation (which exists entirely in memory) for matching tuples. The buffer pool would have to hold the entire smaller relation, one page for reading in the larger relation, and one page to serve as an output buffer. The total cost $= |R| + |S| =$ 1,200. The minimum number of buffer pages for this cost is $|S| + 1 + 1 = 202$.

(d) Any tuple in R can match at most one tuple in S because S.b is a primary key (which means the S.b field contains no duplicates). So the maximum number of tuples in the result is equal to the number of tuples in R, which is 10,000. The size of a tuple in the result could be as large as the size of an R tuple plus the size of an S tuple (minus the size of the shared attribute). This may allow only 5 tuples to be stored on a page. Storing 10,000 tuples at 5 per page would require 2000 pages in the result.

(e) The foreign key constraint tells us that for every R tuple, if $R.a$ is a non-null value, the R tuple must match exactly one S tuple. The cost of Nested Loops joins could be reduced if we make R the outer relation: for each tuple of R with a non-null $R.a$ value, we only have to scan S until a match is found.

3. (Exercise 14.5 R&G) Consider the join $R \bowtie_{R.a=S.b} S$, given the following information about the relations to be joined. The cost metric is the number of page I/Os unless otherwise noted, and the cost of writing out the result should be uniformly ignored.

   - Relation R contains 10,000 tuples and has 10 tuples per page.
   - Relation S contains 2000 tuples and also has 10 tuples per page.
   - Attribute b of relation S is the primary key for S.
   - Both relations are stored as simple heap files.
   - Neither relation has any indexes built on it.
   - 52 buffer pages are available.
   - Assume that all indexes are format 2, any B$^+$-tree index on R has two levels of internal nodes, and any B$^+$-tree index on S has one level of internal nodes.

   (a) With 52 buffer pages, if unclustered B$^+$-tree indexes existed on R.a and S.b,would either provide a cheaper alternative for performing the join (using an **index nested loops join**) than a **block nested loops join**? Explain.

   1. Would your answer change if only five buffer pages were available?
   2. Would your answer change if S contained only 10 tuples instead of 2000 tuples?

   (b) With 52 buffer pages, if clustered B$^+$-tree indexes existed on R.a and S.b, would either provide a cheaper alternative for performing the join (using the **index nested loops algorithm**) than a **block nested loops join**? Explain.

   1. Would your answer change if only five buffer pages were available?
   2. Would your answer change if S contained only 10 tuples instead of 2000 tuples?

Solution:

(a) Consider the join with S as the outer relation. Since attribute b is the primary key of S, we assume that each S-tuple joins with $\frac{10000}{2000} = 5$ R-tuples. Since each data page of relation R can contain 10 tuples, we assume that all the 5 data entries that match each S-tuple fit in one leaf page of R's index. Since the index on R is unclustered, we assume that the I/O cost of retrieving the 5 matching data records is 5 disk I/Os. Therefore, the I/O cost of Index Nested Loop Join with S as outer relation is $|S| + ||S||(2 + 1 + 5)$ $= 200 + 2000 \times (2 + 1 + 5) = 16,200$.

Consider the join with R as the outer relation. Since attribute b is the primary key of S, we each assume that R-tuple joins with one S-tuple. Therefore, the I/O cost of Index Nested Loop Join with R as outer relation is $|R|+||R||(1+1+1) = 1000+10000\times(1+1+1) = 31,000$.

Thus, Block Nested Loop Join, which requires 4,200 I/Os, is still the cheaper alternative.

1. With 5 buffer pages, the cost of the Index Nested Loop Join remains the same, but the cost of the Block Nested Loop join will increase. The new cost of the Block Nested Loop join is $|S| + |R| \times \lceil \frac{|S|}{B-2} \rceil = 67{,}200$. And now the cheapest solution is the Index Nested Loop Join with S as the outer relation.
2. If S contains 10 tuples then we'll need to change some of our previous assumptions. Now all of the S tuples fit on a single page, and each tuple in S will match 1,000 tuples in R. We have
   - Block Nested Loop: Total I/O cost $= |S| + |R| \times \lceil \frac{|S|}{B-2} \rceil = 1{,}001$.
   - Index Nested Loop with S as the outer relation: Total I/O cost $= 1 + 10 \times (2 + \lceil \frac{1000}{b_i} \rceil + 1,000)$, where b_i is the number of leaf node entries per leaf node in the B$^+$-tree on R. Thus, the total I/O cost is at least $10,031$ (with $\lceil \frac{1000}{b_i} \rceil = 1$).
   - Index Nested Loop with R as the outer relation: Due to the large value of $||R||$, the total I/O cost will be higher than that of Index Nested Loop with S as the outer relation.

Therefore, Block Nested Loop is still the best solution.

(b) Consider the join with S as the outer relation. Since the index on R is clustered, we assume that the 5 data tuples that match each S-tuple fit on a single data page in R. The total I/O cost of Index Nested Loop is $|S| + ||S||(2 + 1 + 1) = 200 + 2000 \times (2 + 1 + 1) = 8,200$. As before, using R as the outer relation for the Index Nested Loop will incur a higher I/O cost. Therefore, Block Nested Loop Join remains the cheapest solution with 4,200 I/Os.

1. With 5 buffer pages, the cost of the Index Nested Loop Join remains the same, but the cost of the Block Nested Loop Join will increase. The new cost of the Block Nested Loop Join is $|S| + |R| \times \lceil \frac{|S|}{B-2} \rceil$ = 67,200. And now the cheapest solution is the Index Nested Loop Join with S as the outer relation.
2. If S contains 10 tuples, then we'll need to change some of our previous assumptions. Now all of the S-tuples fit on a single page, and each tuple in S will match 1,000 tuples in R. We have
   - Block Nested Loop Join: Total I/O cost = $|S| + |R| \times \lceil \frac{|S|}{B-2} \rceil$ = 1,001
   - Index Nested Loop with S as the outer relation: Total I/O cost = $1 + 10 \times (2 + \lceil \frac{1000}{b_i} \rceil + \frac{1000}{10}) \geq 1,031$.

   Therefore, Block Nested Loop Join is still the best solution.