Week 7 Exercises: Query Processing with Relational Operations Solutions

Answer 14.1 The answer to each question is given below.

1. (a) *iteration–selection* Scan the entire collection, checking the condition on each tuple, and adding the tuple to the result if the condition is satisfied.
   (b) *indexing–selection* If the selection is equality and a B+ or hash index exists on the field condition, we can retrieve relevant tuples by finding them in the index and then locating them on disk.
   (c) *partitioning–selection* Do a binary search on sorted data to find the first tuple that matches the condition. To retrieve the remaining entries, we simple scan the collection starting at the first tuple we found.
   (d) *iteration–projection* Scan the entire relation, and eliminate unwanted attributes from the result.
   (e) *indexing–projection* If a multi-attribute B+ tree index exists for all of the projection attributes, then one needs to only look at the leaves of the B+.
   (f) *partitioning–projection* To eliminate duplicates when doing a projection, one can simply project out the unwanted attributes and hash a combination of the remaining attributes so duplicates can be easily detected.
   (g) *iteration–join* To join two relations, one takes the first attribute of the first relation and scans the entire second relation to find tuples that match the join condition. Once the first attribute has compared to all tuples in the second relation, the second attribute from the first relation is compared to all tuples in the second relation, and so on.
   (h) *indexing–join* When an index is available, joining two relations can be more efficient. Say there are two relations A and B, and there is a secondary index on the join condition over relation A. The join works as follows: for each tuple in B, we do a lookup on the join attribute in the index over relation A to see if there is a match. If there is, we store the tuple, otherwise we move to the next tuple in relation B.
   (h) *partitioning–join* One can join using partitioning by using hash join variant or a sort-merge join. For example, if there is a sort merge join, we sort both relations on the join condition. Next, we scan both relations and identify matches. After sorting, this requires only a single scan over each relation.
2. The *most selective access path* is the query access path that retrieves the fewest pages during query evaluation. This is the most efficient way to gather the query's results.
3. *Conjunctive normal form* is important in query evaluation because often indexes exist over some subset of conjuncts in a CNF expression. Since conjunct order does not matter in CNF expressions, often indexes can be used to increase the selectivity of operators by doing a selection over two, three, or more conjuncts using a single multiattribute index.

4. An index *matches* a selection condition if the index can be used to retrieve just the tuples that satisfy the condition. A *primary term* in a selection condition is a conjunct that matches an index (i.e. can be used by the index).

Answer 14.3 The answer to each question is given below.

1. The first pass will produce N1 = ⌈N/B⌉ runs of B pages each, where B = 10 and N = 5000 (we project the half of the relation, 10000/2). Therefore, the first pass produces 5000/10=500 sorted runs of 10 pages each, costing 15000 I/Os (i.e., reading 10000 and writing 5000 pages).
2. The number of additional merge passes is $\lceil \log_{B-1} N1 \rceil$, where B = 10 and N1 = 500.

   Therefore, $\lceil \log_9 500 \rceil$ = 3 more passes are required to merge the 500 runs.

   The first two merge passes need to read&write the number of pages, costing 2*2*5000=20000 I/Os. The third merge pass needs to read the number of pages, costing 5000 I/Os (we omit writing in the last pass, as it produces the final result).
3. Since ename is a candidate key, it is not necessary to perform duplicate checking for <title, ename> pairs. We ignore the I/O cost of traversing B+ tree from root to leaf because it involves only a handful of pages due to typical low heights of the such trees. We also assume that the leaf nodes of an index with x (out of 4) attributes take approximately 10000 * x/4 pages in total. Let the tuples per page for the table be *tpp.*

   (a) Using the clustered B+ tree index on title would reduce the cost to single scan with 10,000 (scan of table) I/Os, which would be cheaper than sorting. (Note: for clustered indexes using Alternative (1), the leaf nodes and data records are the same, so the cost only includes the leaf node scan, but in such as case, the number of leaf pages is actually 10000. Essentially it would be same as file scan). Using an unclustered index would cost up to 2500 (scan of index) + 10,000 * *tpp* (scan of table). Thus, an unclustered index would not be cheaper.

   (b) Using the clustered B+ tree on ename would be cheaper than sorting, in that the cost of using the B+ tree would be 10,000 (scan of table) I/Os. (If Altenative (1) index, then the cost is 10,000 like in (a)). An unclustered index would require up to: 2500 (scan of index) + 10000 * *tpp* (scan of table) I/Os and thus be more expensive than sorting.

   (c) Index with two attributes: number of leaf pages = 10000 * 2/4 = 5000. This index contains all the attributes required by the query and we can use an index-only scan without fetching records from the table. Thus, for both a clustered and unclustered B+ tree, the cost includes only the scanning of the leaf nodes and thus would be around 5000 I/O's.
4. 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 an index (clustered or unclustered) on <ename, title> is available; in this case, we can do an index-only scan. (Note that even with DISTINCT specified, no duplicates are actually present in the answer because ename is a candidate key. However, in the general case, a query optimizer may fail to recognize this pattern and omit the duplicate elimination step.)