CHAPTER 15

Query Processing

Solutions for the Practice Exercises of Chapter 15

Practice Exercises

15.1

Answer:

We will refer to the tuples (kangaroo, 17) through (baboon, 12) using tuple numbers t_1 through t_{12}. We refer to the j^{th} run used by the i^{th} pass, as r_{ij}. The initial sorted runs have three blocks each. They are:

$$\begin{aligned} r_{11} &= \{t_3, t_1, t_2\} \\ r_{12} &= \{t_6, t_5, t_4\} \\ r_{13} &= \{t_9, t_7, t_8\} \\ r_{14} &= \{t_{12}, t_{11}, t_{10}\} \end{aligned}$$

Each pass merges three runs. Therefore the runs after the end of the first pass are:

$$\begin{aligned} r_{21} &= \{t_3, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\} \\ r_{22} &= \{t_{12}, t_{11}, t_{10}\} \end{aligned}$$

At the end of the second pass, the tuples are completely sorted into one run:

$$r_{31} = \{t_{12}, t_3, t_{11}, t_{10}, t_1, t_6, t_9, t_5, t_2, t_7, t_4, t_8\}$$

15.2

Answer:
Query:

$$\Pi_{T.branch\_name}((\Pi_{branch\_name,\ assets}(\rho_T(branch))) \bowtie_{T.assets > S.assets} (\Pi_{assets}\ (\sigma_{(branch\_city = \text{'Brooklyn'})}(\rho_S(branch)))))$$

This expression performs the theta join on the smallest amount of data possible. It does this by restricting the right-hand side operand of the join to only those branches in Brooklyn and also eliminating the unneeded attributes from both the operands.

15.3

Answer:
r_1 needs 800 blocks, and r_2 needs 1500 blocks. Let us assume M pages of memory. If $M > 800$, the join can easily be done in 1500 + 800 disk accesses, using even plain nested-loop join. So we consider only the case where $M \leq 800$ pages.

a. Nested-loop join:
Using r_1 as the outer relation, we need $20000 * 1500 + 800 = 30,000,800$ disk accesses. If r_2 is the outer relation, we need $45000 * 800 + 1500 = 36,001,500$ disk accesses.

b. Block nested-loop join:
If r_1 is the outer relation, we need $\lceil \frac{800}{M-1} \rceil * 1500 + 800$ disk accesses. If r_2 is the outer relation, we need $\lceil \frac{1500}{M-1} \rceil * 800 + 1500$ disk accesses.

c. Merge join:
Assuming that r_1 and r_2 are not initially sorted on the join key, the total sorting cost inclusive of the output is $B_s = 1500(2\lceil log_{M-1}(1500/M)\rceil + 2) + 800(2\lceil log_{M-1}(800/M)\rceil + 2)$ disk accesses. Assuming all tuples with the same value for the join attributes fit in memory, the total cost is $B_s + 1500 + 800$ disk accesses.

d. Hash join:
We assume no overflow occurs. Since r_1 is smaller, we use it as the build relation and r_2 as the probe relation. If $M > 800/M$, i.e., no need for recursive partitioning, then the cost is $3(1500 + 800) = 6900$ disk accesses, else the cost is $2(1500 + 800)\lceil log_{M-1}(800) - 1\rceil + 1500 + 800$ disk accesses.

15.4

Answer:

If there are multiple tuples in the inner relation with the same value for the join attributes, we may have to access that many blocks of the inner relation for each tuple of the outer relation. That is why it is inefficient. To reduce this cost we can perform a join of the outer relation tuples with just the secondary index leaf entries, postponing the inner relation tuple retrieval. The result file obtained is then sorted on the inner relation addresses, allowing an efficient physical order scan to complete the join.

Hybrid merge–join requires the outer relation to be sorted. The above algorithm does not have this requirement, but for each tuple in the outer relation it needs to perform an index lookup on the inner relation. If the outer relation is much larger than the inner relation, this index lookup cost will be less than the sorting cost, thus this algorithm will be more efficient.

15.5

Answer:

We can store the entire smaller relation in memory, read the larger relation block by block, and perform nested-loop join using the larger one as the outer relation. The number of I/O operations is equal to $b_r + b_s$, and the memory requirement is $min(b_r, b_s) + 2$ pages.

15.6

Answer:

a. Use the index to locate the first tuple whose *branch_city* field has value "Brooklyn". From this tuple, follow the pointer chains till the end, retrieving all the tuples.

b. For this query, the index serves no purpose. We can scan the file sequentially and select all tuples whose *branch_city* field is anything other than "Brooklyn".

c. This query is equivalent to the query

$$\sigma_{(branch\_city \geq \text{'Brooklyn'} \ \wedge \ assets < 5000)}(branch)$$

Using the *branch-city* index, we can retrieve all tuples with *branch-city* value greater than or equal to "Brooklyn" by following the pointer chains from the first "Brooklyn" tuple. We also apply the additional criteria of $assets < 5000$ on every tuple.

15.7

Answer:

Let *outer* be the iterator which returns successive tuples from the pipelined outer relation. Let *inner* be the iterator which returns successive tuples of the inner relation having a given value at the join attributes. The *inner* iterator returns these tuples by performing an index lookup. The functions **IndexedNLJoin::open**, **IndexedNLJoin::close** and **IndexedNLJoin::next** to implement the indexed nested-loop join iterator are given below. The two iterators *outer* and *inner*, the value of the last read outer relation tuple t_r and a flag $done_r$ indicating whether the end of the outer relation scan has been reached are the state information which need to be remembered by **IndexedNLJoin** between calls. Please see Figure 15.101

15.8

Answer:

Suppose $r(T \cup S)$ and $s(S)$ are two relations and $r \div s$ has to be computed.

For a sorting-based algorithm, sort relation s on S. Sort relation r on (T, S). Now, start scanning r and look at the T attribute values of the first tuple. Scan r till tuples have same value of T. Also scan s simultaneously and check whether every tuple of s also occurs as the S attribute of r, in a fashion similar to merge join. If this is the case, output that value of T and proceed with the next value of T. Relation s may have to be scanned multiple times, but r will only be scanned once. Total disk accesses, after sorting both the relations, will be $|r| + N * |s|$, where N is the number of distinct values of T in r.

We assume that for any value of T, all tuples in r with that T value fit in memory, and we consider the general case at the end. Partition the relation r on attributes in T such that each partition fits in memory (always possible because of our assumption). Consider partitions one at a time. Build a hash table on the tuples, at the same time collecting all distinct T values in a separate hash table. For each value of T, Now, for each value V_T of T, each value s of S, probe the hash table on (V_T, s). If any of the values is absent, discard the value V_T, else output the value V_T.

In the case that not all r tuples with one value for T fit in memory, partition r and s on the S attributes such that the condition is satisfied, and run the algorithm on each corresponding pair of partitions r_i and s_i. Output the intersection of the T values generated in each partition.

15.9

Answer:

Seek overhead is reduced, but the the number of runs that can be merged in a pass decreases, potentially leading to more passes. A value of b_b that minimizes overall cost should be chosen.

```
IndexedNLJoin::open()
begin
    $outer$.open();
    $inner$.open();
    $done_r$ := $false$;
    if($outer$.next() $\neq$ $false$)
        move tuple from $outer$'s output buffer to $t_r$;
    else
        $done_r$ := $true$;
end

IndexedNLJoin::close()
begin
    $outer$.close();
    $inner$.close();
end

boolean IndexedNLJoin::next()
begin
    while($\neg done_r$)
    begin
        if($inner$.next($t_r[JoinAttrs]$) $\neq$ $false$)
        begin
            move tuple from $inner$'s output buffer to $t_s$;
            compute $t_r \bowtie t_s$ and place it in output buffer;
            return $true$;
        end
        else
            if($outer$.next() $\neq$ $false$)
            begin
                move tuple from $outer$'s output buffer to $t_r$;
                rewind $inner$ to first tuple of $s$;
            end
            else
                $done_r$ := $true$;
    end
    return $false$;
end
```

Figure 15.101 Answer for Exercise 15.7.

15.10

Answer:
As in the case of join algorithms, semijoin and anti-semijoin can be done efficiently if the join conditions are equijoin conditions. We describe below how to efficiently handle the case of equijoin conditions using sorting and hashing. With arbitrary join conditions, sorting and hashing cannot be used; (block) nested loops join needs to be used instead.

a. **Semijoin:**

- **Semijoin using sorting:** Sort both r and s on the join attributes in θ. Perform a scan of both r and s similar to the merge algorithm and add tuples of r to the result whenever the join attributes of the current tuples of r and s match.
- **Semijoin using hashing:** Create a hash index in s on the join attributes in θ. Iterate over r, and for each distinct value of the join attributes, perform a hash lookup in s. If the hash lookup returns a value, add the current tuple of r to the result.

  Note that if r and s are large, they can be partitioned on the join attributes first and the above procedure applied on each partition. If r is small but s is large, a hash index can be built on r and probed using s; and if an s tuple matches an r tuple, the r tuple can be output and deleted from the hash index.

b. **Anti-semijoin:**

- **Anti-semijoin using sorting**: Sort both r and s on the join attributes in θ. Perform a scan of both r and s similar to the merge algorithm and add tuples of r to the result if no tuple of s satisfies the join predicate for the corresponding tuple of r.
- **Anti-semijoin using hashing**: Create a hash index in s on the join attributes in θ. Iterate over r, and for each distinct value of the join attributes, perform a hash lookup in s. If the hash lookup returns a null value, add the current tuple of r to the result.
  As for semijoin, partitioning can be used if r and s are large. An index on r can be used instead of an index on s, but then when an s tuple matches an r tuple, the r tuple is deleted from the index. After processing all s tuples, all remaining r tuples in the index are output as the result of the anti-semijoin operation.

15.11

Answer:
Demand driven is better, since it will only generate the top K results. Producer driven may generate a lot more answers, many of which would not get used.

```
initialize the list $L$ to the empty list;
for (each keyword $c$ in $S$) do
begin
   $D$ := the list of documents identifiers corresponding to $c$;
   for (each document identifier $d$ in $D$) do
      if (a record $R$ with document identifier as $d$ is on list $L$) then
            $R.reference\_count$ := $R.reference\_count + 1$;
      else begin
            make a new record $R$;
            $R.document\_id$ := $d$;
            $R.reference\_count$ := 1;
            add $R$ to $L$;
      end;
end;
for (each record $R$ in $L$) do
   if ($R.reference\_count >= k$) then
      output $R$;
```

Figure 15.102 Answer for Exercise 15.13.

15.12

Answer:
Producer-driven pipelining executes the same set of instructions to generate multiple tuples by consuming already generated tuples from the inputs. Thus instruction cache hits will be more. In comparison, demand-driven pipelining switches from the instructions of one function to another for each tuple, resulting in more misses.

By generating multiple results at one go, a *next*(() function would receive multiple tuples in its inputs and have a loop that generates multiple tuples for its output without switching execution to another function. Thus, the instruction cache hit rate can be expected to improve.

15.13

Answer:
Let S be a set of n keywords. An algorithm to find all documents that contain at least k of these keywords is given in Figure 15.102

This algorithm calculates a reference count for each document identifier. A reference count of i for a document identifier d means that at least i of the keywords in S occur in the document identified by d. The algorithm maintains a list of records, each having two fields – a document identifier, and the reference count for this identifier. This list is maintained sorted on the document identifier field.

Note that execution of the second *for* statement causes the list D to “merge” with the list L. Since the lists L and D are sorted, the time taken for this merge is proportional to the sum of the lengths of the two lists. Thus the algorithm runs in time (at most) proportional to n times the sum total of the number of document identifiers corresponding to each keyword in S.

15.14

Answer:
Add doc to index lists for more general concepts also.

15.15

Answer:
If the nested-loops join algorithm is used as is, it would require tuples for each of the relations to be assembled before they are joined. Assembling tuples can be expensive in a column store, since each attribute may come from a separate area of the disk; the overhead of assembly would be particularly wasteful if many tuples do not satisfy the join condition and would be discarded. In such a situation it would be better to first find which tuples match by accessing only the join columns of the relations. Sort-merge join, hash join, or indexed nested loops join can be used for this task. After the join is performed, only tuples that get output by the join need to be assembled; assembly can be done by sorting the join result on the record identifier of one of the relations and accessing the corresponding attributes, then resorting on record identifiers of the other relation to access its attributes.

15.16

Answer:
FILL IN