CHAPTER 12

Query Processing

This chapter describes the process by which queries are executed efficiently by a database system. The chapter starts off with measures of cost, then proceeds to algorithms for evaluation of relational algebra operators and expressions. This chapter applies concepts from Chapter 2, 6, 10, and 11.

Query processing algorithms can be covered without tedious and distracting details of size estimation. Although size estimation is covered later, in Chapter 13, the presentation there has been simplified by omitting some details. Instructors can choose to cover query processing but omit query optimization, without loss of continuity with later chapters.

Exercises

12.10 Suppose you need to sort a relation of 40 gigabytes, with 4 kilobyte blocks, using a memory size of 40 megabytes. Suppose the cost of a seek is 5 milliseconds, while the disk transfer rate is 40 megabytes per second.

a. Find the cost of sorting the relation, in seconds, with $b_b = 1$ and with $b_b = 100$.

b. In each case, how many merge passes are required?

c. Suppose a flash storage device is used instead of a disk, and it has a seek time of 1 microsecond, and a transfer rate of 40 megabytes per second. Recompute the cost of sorting the relation, in seconds, with $b_b = 1$ and with $b_b = 100$, in this setting.

Answer:

a. The number of blocks in the main memory buffer available for sorting(M) is $\frac{40\times10^6}{4\times10^3} = 10^4$. The number of blocks containing records of the given relation (b_r) is $\frac{40\times10^9}{4\times10^3} = 10^7$. Then the cost of sorting the relation is: (*Number of disk seeks* $\times$ *Disk seek cost*) + (*Number of block transfers* $\times$ *Block transfer time*). Here Disk seek cost is $5x10^{-3}$ seconds and Block transfer time is 10^{-4} seconds ($\frac{4\times10^3}{40\times10^6}$). The number of block transfers is independent of b_b and is equal to 25×10^6.

- **Case 1**: $b_b = 1$
  Using the equation in Section 12.4, the number of disk seeks is 5002×10^3. Therefore the cost of sorting the relation is: $(5002 \times 10^3) \times (5 \times 10^{-3}) + (25 \times 10^6) \times (10^{-4}) = 25 \times 10^3 + 2500 = 27500$ seconds.

- **Case 2:** $b_b = 100$
  The number of disk seeks is: 52×10^3. Therefore the cost of sorting the relation is: $(52 \times 10^3) \times (5 \times 10^{-3}) + (25 \times 10^6) \times (10^{-4})$ $= 260 + 2500 = 2760$ seconds.

b. Disk storage The number of merge passes required is given by $\lceil log_{M-1}(\frac{b_r}{M})\rceil$. This is independent of b_b. Substituting the values above, we get $\lceil log_{10^4-1}(\frac{10^7}{10^4})\rceil$ which evaluates to 1.

c. Flash storage:

- **Case 1**: $b_b = 1$
  The number of disk seeks is: 5002×10^3. Therefore the cost of sorting the relation is: $(5002 \times 10^3) \times (1 \times 10^{-6}) + (25 \times 10^6) \times (10^{-4}) = 5.002 + 2500$ 2506 seconds.

- **Case 2**: $b_b = 100$
  The number of disk seeks is: 52×10^3. Therefore the cost of sorting the relation is: $(52 \times 10^3) \times (1 \times 10^{-6}) + (25 \times 10^6) \times (10^{-4}) =$ $0.052 + 2500$, which is approx $= 2500$ seconds.

12.11 Consider the following extended relational-algebra operators. Describe how to implement each operation using sorting, and using hashing.

a. **Semijoin** ($\ltimes_\theta$): $r \ltimes_\theta s$ is defined as $\Pi_R(r \bowtie_\theta s)$, where R is the set of attributes in the schema of r; that it selects those tuples r_i in r for which there is a tuple s_j in s such that r_i and s_j satisfy predicate θ.

b. **Anti-semijoin** ($\bar{\ltimes}_\theta$): $r \bar{\ltimes}_\theta s$ is defined as $r - \Pi_R(r \bowtie_\theta s)$; that it it selects those tuples r_i in r for which there is no tuple s_j in s such that r_i and s_j satisfy predicate θ.

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.

12.12 Why is it not desirable to force users to make an explicit choice of a query-processing strategy? Are there cases in which it *is* desirable for users to be aware of the costs of competing query-processing strategies? Explain your answer.

Answer: In general it is not desirable to force users to choose a query processing strategy because naive users might select an inefficient strategy. The reason users would make poor choices about processing queries is that they would not know how a relation is stored, nor about its indices. It is unreasonable to force users to be aware of these details since ease of use is a major object of database query languages. If users are aware of the costs of different strategies they could write queries efficiently, thus helping performance. This could happen if experts were using the system.

12.13 Design a variant of the hybrid merge-join algorithm for the case where both relations are not physically sorted, but both have a sorted secondary index on the join attributes.

Answer: We merge the leaf entries of the first sorted secondary index with the leaf entries of the second sorted secondary index. The result file contains pairs of addresses, the first address in each pair pointing to a

tuple in the first relation, and the second address pointing to a tuple in the second relation.
This result file is first sorted on the first relation's addresses. The relation is then scanned in physical storage order, and addresses in the result file are replaced by the actual tuple values. Then the result file is sorted on the second relation's addresses, allowing a scan of the second relation in physical storage order to complete the join.

12.14 Estimate the number of block transfers and seeks required by your solution to Exercise 12.13 for $r_1 \bowtie r_2$, where r_1 and r_2 are as defined in Practice Exercise 12.3.

Answer: r_1 occupies 800 blocks, and r_2 occupies 1500 blocks. Let there be n pointers per index leaf block (we assume that both the indices have leaf blocks and pointers of equal sizes). Let us assume M pages of memory, $M < 800$. r_1's index will need $B_1 = \lceil \frac{20000}{n} \rceil$ leaf blocks, and r_2's index will need $B_2 = \lceil \frac{45000}{n} \rceil$ leaf blocks. Therefore the merge join will need $B_3 = B_1 + B_2$ accesses, without output. The number of output tuples is estimated as $n_o = \lceil \frac{20000 * 45000}{max(V(C,r_1), V(C,r_2))} \rceil$. Each output tuple will need two pointers, so the number of blocks of join output will be $B_{o1} = \lceil \frac{n_o}{n/2} \rceil$. Hence the join needs $B_j = B_3 + B_{o1}$ disk block accesses.
Now we have to replace the pointers by actual tuples. For the first sorting, $B_{s1} = B_{o1}(2\lceil log_{M-1}(B_{o1}/M) \rceil + 2)$ disk accesses are needed, including the writing of output to disk. The number of blocks of r_1 which have to be accessed in order to replace the pointers with tuple values is $min(800, n_o)$. Let n_1 pairs of the form (r_1 *tuple, pointer to* r_2) fit in one disk block. Therefore the intermediate result after replacing the r_1 pointers will occupy $B_{o2} = \lceil (n_o/n_1) \rceil$ blocks. Hence the first pass of replacing the r_1-pointers will cost $B_f = B_{s1} + B_{o1} + min(800, n_o) + B_{o2}$ disk accesses.
The second pass for replacing the r_2-pointers has a similar analysis. Let n_2 tuples of the final join fit in one block. Then the second pass of replacing the r_2-pointers will cost $B_s = B_{s2} + B_{o2} + min(1500, n_o)$ disk accesses, where $B_{s2} = B_{o2}(2\lceil log_{M-1}(B_{o2}/M) \rceil + 2)$.
Hence the total number of disk accesses for the join is $B_j + B_f + B_s$, and the number of pages of output is $\lceil n_o/n_2 \rceil$.

12.15 The hash-join algorithm as described in Section 12.5.5 computes the natural join of two relations. Describe how to extend the hash-join algorithm to compute the natural left outer join, the natural right outer join and the natural full outer join. (Hint: Keep extra information with each tuple in the hash index, to detect whether any tuple in the probe relation matches the tuple in the hash index.) Try out your algorithm on the *takes* and *student* relations.

Answer: For the probe relation tuple t_r under consideration, if no matching tuple is found in the build relation's hash partition, it is padded with nulls and included in the result. This will give us the natural left outer join $t_r ⟕ t_s$. To get the natural right outer join $t_r ⟖ t_s$, we can keep

a boolean flag with each tuple in the current build relation partition s_i residing in memory, and set it whenever any probe relation tuple matches with it. When we are finished with s_i, all the tuples in it which do not have their flag set, are padded with nulls and included in the result. To get the natural full outer join, we do both the above operations together.
To try out our algorithm, we use the sample *student* and *takes* relations of Figures A.9 and A.10. Let us assume that there is enough memory to hold three tuples of the build relation plus a hash index for those three tuples. We use *takes* as the build relation. We use the simple hashing function which returns the *student_ID* mod 10. Taking the partition corresponding to value 7, we get $r_1 = \{(\text{"Snow"})\}$, and $s_1 = \phi$. The tuple in the probe relation partition will have no matching tuple, so ("70557", "Snow", "Physics","0", *null*) is outputted. Proceeding in a similar way, we process all the partitions and complete the join.

12.16 Pipelining is used to avoid writing intermediate results to disk. Suppose you need to sort relation r using sort–merge and merge-join the result with an already sorted relation s.

a. Describe how the output of the sort of r can be pipelined to the merge join without being written back to disk.

b. The same idea is applicable even if both inputs to the merge join are the outputs of sort–merge operations. However, the available memory has to be shared between the two merge operations (the merge-join algorithm itself needs very little memory). What is the effect of having to share memory on the cost of each sort–merge operation?

Answer:

a. Using pipelining, output from the sorting operation on r is written to a buffer B. When B is full, the merge-join processes tuples from B, joining them with tuples from s until B is empty. At this point, the sorting operation is resumed and B is refilled. This process continues until the merge-join is complete.

b. If the sort–merge operations are run in parallel and memory is shared equally between the two, each operation will have only $M/2$ frames for its memory buffer. This may increase the number of runs required to merge the data.

12.17 Write pseudocode for an iterator that implements a version of the sort –merge algorithm where the result of the final merge is pipelined to its consumers. Your pseudocode must define the standard iterator functions *open*(), *next*(), and *close*(). Show what state information the iterator must maintain between calls.

Answer: Let M denote the number of blocks in the main memory buffer available for sorting. For simplicity we assume that there are less than M runs created in the run creation phase. The pseudocode for the iterator functions open, next and close are as follows:

```
SortMergeJoin::open()
begin
    repeat
        read M blocks of the relation;
        sort the in-memory part of the relation;
        write the sorted data to a run file $R_i$
    until the end of the relation
    read one block of each of the $N$ run files $R_i$, into a
        buffer block in memory
    $done_r$ := false;
end

SortMergeJoin::close()
begin
    clear all the N runs from main memory and disk;
end

boolean SortMergeJoin::next()
begin
    if the buffer block of any run $R_i$ is empty and not end-of-file($R_i$)
        begin
            read the next block of $R_i$ (if any) into the buffer block;
        end
    if all buffer blocks are empty
        return false;
    choose the first tuple (in sort order) among the buffer blocks;
    write the tuple to the output buffer;
    delete the tuple from the buffer block and increment its pointer;
    return true;
end
```

12.18 Suppose you have to compute ${}_A\mathcal{G}_{sum(C)}(r)$ as well as ${}_{A,B}\mathcal{G}_{sum(C)}(r)$. Describe how to compute these together using a single sorting of r.

Answer: Run the sorting operation on r, grouping by (A, B), as required for the second result. When evaluating the sum aggregate, keep running totals for both the (A, B) grouping as well as for just the A grouping.