HASH JOIN: Now both relations are the same size, so we can treat either one as the smaller relation. With 15 buffer pages the first scan of S splits it into 14 buckets, each containing about 72 pages, so again we have to deal with partition overflow. We must apply the Hash Join technique again to all partitions of R and S that were created by the first partitioning phase. Then we can fit an entire partition of S in memory. The total cost will be the cost of two partioning phases plus the cost of one matching phase.

$$TotalCost = 2 * (2 * (M + N)) + (M + N) = 10,000$$

5. SORT-MERGE: With 52 buffer pages we have $B > \sqrt{M}$ so we can use the "merge-on-the-fly" refinement which costs $3 * (M + N)$.

$$TotalCost = 3 * (1,000 + 1,000) = 6,000$$

HASH JOIN: Now both relations are the same size, so we can treat either one as the smaller relation. With 52 buffer pages the first scan of S splits it into 51 buckets, each containing about 20 pages. This time we do not have to deal with partition overflow. The total cost will be the cost of one partioning phase plus the cost of one matching phase.

$$TotalCost = (2 * (M + N)) + (M + N) = 6,000$$

Exercise 12.6 Answer each of the questions—if some question is inapplicable, explain why—in Exercise 12.1 again, but using the following information about R and S:

Relation R contains 200,000 tuples and has 20 tuples per page.
Relation S contains 4,000,000 tuples and also has 20 tuples per page.
Attribute a of relation R is the primary key for R.
Each tuple of R joins with exactly 20 tuples of S.
1,002 buffer pages are available.

Answer 12.6 Let M = 10,000 be the number of pages in R, N = 200,000 be the number of pages in S, and B = 1002 be the number of buffer pages available.

1. 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

$$\#pagesinouter + (\#pagesinouter * \#pagesininner)$$

which is minimized by having the smaller relation be the outer relation.

$$TotalCost = M + (M * N) = 2,000,010,000$$

The minimum number of buffer pages for this cost is 3.

2. 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 only once for each outer block, of which there are $\lceil \frac{\#pagesinouter}{B-2} \rceil$.

$$TotalCost = M + N * \lceil \frac{M}{B-2} \rceil = 2,010,000$$

   The minimum number of buffer pages for this cost is 1002.

3. Since $B > \sqrt{N} > \sqrt{M}$ we can use the refinement to Sort-Merge discussed on pages 254-255 in the text.

$$TotalCost = 3 * (M + N) = 630,000$$

   NOTE: if $R.a$ were not a key, then the merging phase could require more than one pass over one of the relations, making the cost of merging $M * N$ I/Os in the worst case.

   The minimum number of buffer pages required is 325. With 325 buffer pages, the initial sorting pass will split R into 16 runs of size 650 and split S into 308 runs of size 650 (approximately). These 324 runs can then be merged in one pass, with one page left over to be used as an output buffer. With fewer than 325 buffer pages the number of runs produced by the first pass over both relations would exceed the number of available pages, making a one-pass merge impossible.

4. The cost of Hash Join is $3*(M+N)$ if $B > \sqrt{f * M}$ where f is a 'fudge factor' used to capture the small increase in size involved in building a hash table, and M is the number of pages in the smaller relation, S (see page 258). Since $\sqrt{M} = 100$, we can assume that this condition is met. We will also assume uniform partitioning from our hash function.

$$TotalCost = 3 * (M + N) = 630,000$$

   Without knowing f we can only approximate the minimum number of buffer pages required, and a good guess is that we need $B > \sqrt{f * M}$.

5. 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.

$$TotalCost = M + N = 210,000$$

   The minimum number of buffer pages for this cost is $M + 1 + 1 = 10,002$.

6. Any tuple in S can match at most one tuple in R because *R.a* is a primary key (which means the *R.a* field contains no duplicates). So the maximum number of tuples in the result is equal to the number of tuples in S, which is 4,000,000.

   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 10 tuples to be stored on a page. Storing 4,000,000 tuples at 10 per page would require 400,000 pages in the result.

7. If R.b is a foreign key referring to S.a, this contradicts the statement that each R tuple joins with exactly 20 S tuples.

Exercise 12.7 We described variations of the join operation called *outer joins* in Section 5.6.4. One approach to implementing an outer join operation is to first evaluate the corresponding (inner) join and then add additional tuples padded with *null* values to the result in accordance with the semantics of the given outer join operator. However, this requires us to compare the result of the inner join with the input relations to determine the additional tuples to be added. The cost of this comparison can be avoided by modifying the join algorithm to add these extra tuples to the result while input tuples are processed during the join. Consider the following join algorithms: *block nested loops join, index nested loops join, sort-merge join,* and *hash join.* Describe how you would modify each of these algorithms to compute the following operations on the Sailors and Reserves tables discussed in this chapter:

1. Sailors `NATURAL LEFT OUTER JOIN` Reserves
2. Sailors `NATURAL RIGHT OUTER JOIN` Reserves
3. Sailors `NATURAL FULL OUTER JOIN` Reserves

Answer 12.7 Answer not available.

(b) $age = 25$ The most cost effective technique here employs sorting the clustered B+ index over $< age, sal >$, as the grouping requires that the output be sorted. An external merge sort with 11 buffer pages would require 16000. Totalled, the cost equals 16000 (sort) + 10000 * 0.4 = 20000.

(c) $age > 20$ This selection criterion works similarly to the previous one, in that an external merge over $< age, sal >$ is required, using the clustered index provided as the pages to sort. The final cost is the same, 20000.

(d) $eid = 1000$ Begin a candidate key, only one relation should match with a given *eid* value. Thus, the estimated cost should be 1.2 (hash lookup) + 1 (relation retrieval).

(e) $sal > 200 \wedge age > 30$ This case is similar to the $sal > 100$ case above, cost = 16412.

(f) $sal > 200 \wedge age = 20$ Again, this case is also similar to the $sal > 100$ case, cost = 16412.

(g) $sal > 200 \wedge title = "CFO"$ The solution to this case greatly depends of the number of tuples per page. Assuming a small number of tuples per page, the cheapest route is to use the B+ tree index over *sal*, getting each index. The total cost for this is 2 (lookup, $sal > 200$) + 10000 * .2 (smaller size) * .1 (selectivity) + 10000 * .1 (selectivity) * tuples per page. The solution to this case is similar to that of the other requiring sorts, but at a higher cost. Since the sort can't be preformed over the clustered B+ tree in this case, the sort costs 40000 I/Os. Thus, for tuples per page ¡ 40, the B+ index method is superior, otherwise, the sort solution is cheaper.

(h) $sal > 200 \wedge age > 30 \wedge title = "CFO"$ This solution is the same as the previous, since either the index over *sal* or an external sort must be used. The cost is the cheaper of 2 + 1000 * (.2 + tuples per page) [index method] and 40000 [sort method].

5. (a) $sal > 200 \vee age = 20$ In this case, a filescan would be the most cost effective, because the most cost effective method for satisfying $sal > 200$ alone is a filescan.

(b) $sal > 200 \vee title = "CFO"$ Again a filescan is the better alternative here, since no index at all exists for *title*.

(c) $title = "CFO" \wedge ename = "Joe"$ Even though this condition is a conjunction, the filescan is still the best method, since no indexes exist on either *title* or *ename*.

Exercise 14.3 For each of the following SQL queries, for each relation involved, list the attributes that must be examined in order to compute the answer. All queries refer to the following relations:

(b) An unclustered index would preclude the low cost of the previous plan and necessitate the choice of a simple filescan, cost = 10000, as the best.

(c) Due to the `WHERE` clause, the clustered B+ index on ename doesn't help at all. The best alternative is to use a filescan, cost = 10000.

(d) Again, as in the previous answer, the best choice is a filescan, cost = 10000.

(e) Although the order of the B+ index key makes the tree much less useful, the leaves can still be scanned in an index-only scan, and the increased number of tuples per page lowers the I/O cost. Cost = 10000 * .5 = 5000.

2. (a) A clustered index on *title* would allow scanning of only the 10% of the tuples desired. Thus the total cost is 2 (lookup) + 10000 * 10% + 2500 * 10%= 1252.

(b) A clustered index on *dname* works functionally in the same manner as that in the previous question, for a cost 1002 + 250 = 1252 . The *ename* field still must be retrieved from the relation data pages.

(c) In this case, using the index lowers the cost of the query slightly, due to the greater selectivity of the combined query and to the search key taking advantage of it. The total cost = 2 (look up) + 10000 * 5% + 5000 * 5% =752.

(d) Although this index does contain the output field, the *dname* still must be retrieved from the relational data pages, for a cost of 2 (lookup) + 10000 * 10% + 5000 * 10%= 1502.

(e) Since this index contains all three indexes needed for an index-only scan, the cost drops to 2 (look up) + 10000 * 5% * .75 (smaller size) = 402.

(f) Finally, in this case, the prefix cannot be matched with the equality information in the `WHERE` clause, and thus a scan would be the superior method of retrieval. However, as the clustered B+ tree's index contains all the indexes needed for the query and has a smaller tuple, scanning the leaves of the B+ tree is the best plan, costing 10000 * .75 = 7500 I/Os.

3. (a) Since *title* is the only attribute required, an index-only scan could be performed, with a running counter. This would cost 10000 * .25 (index-only scan, smaller tuples) = 2500.

(b) Again, as the index contains the only attribute of import, an index-only scan could again be performed, for a cost of 2500.

(c) This index is useless for the given query, and thus requires a sorting of the file, costing 10000 + 3 * 2 * (2500). Finally, a scan of this sorted result will allow us to answer the query, for a cost of 27500.

(d) This is similar to the previous part, except that the initial scan requires fewer I/Os if the leaves of the B+ tree are scanned instead of the data file. Cost = 5000 + 3 * 2 * (2500) = 22500.