
EVALUATION OF RELATIONAL OPERATORS

Exercise 14.1 Briefly answer the following questions:

1. Consider the three basic techniques, *iteration*, *indexing*, and *partitioning*, and the relational algebra operators *selection*, *projection*, and *join*. For each technique-operator pair, describe an algorithm based on the technique for evaluating the operator.
2. Define the term *most selective access path for a query*.
3. Describe *conjunctive normal form*, and explain why it is important in the context of relational query evaluation.
4. When does a general selection condition *match* an index? What is a *primary term* in a selection condition with respect to a given index?
5. How does hybrid hash join improve on the basic hash join algorithm?
6. Discuss the pros and cons of hash join, sort-merge join, and block nested loops join.
7. If the join condition is not equality, can you use sort-merge join? Can you use hash join? Can you use index nested loops join? Can you use block nested loops join?
8. Describe how to evaluate a grouping query with aggregation operator **MAX** using a sorting-based approach.
9. Suppose that you are building a DBMS and want to add a new aggregate operator called **SECOND LARGEST**, which is a variation of the **MAX** operator. Describe how you would implement it.
10. Give an example of how buffer replacement policies can affect the performance of a join algorithm.

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 simply scan the collection starting at the first tuple we found.
 - (d) *iteration-projection* Scan the entire relation, and eliminate unwanted attributes in the result.
 - (e) *indexing-projection* If a multiattribute 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 in 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 lookup the join attribute in the index over relation A to see if there is a match. If there is a match, we store the tuple, otherwise we move to the next tuple in relation B.
 - (i) *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 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).

5. Hybrid hash join improves performance by comparing the first hash buckets during the partitioning phase rather than saving it for the probing phase. This saves us the cost of writing and reading the first partition to disk.
6. Hash join provides excellent performance for equality joins, and can be tuned to require very few extra disk accesses beyond a one-time scan (provided enough memory is available). However, hash join is worthless for non-equality joins.

Sort-merge joins are suitable when there is either an equality or non-equality based join condition. Sort-merge also leaves the results sorted which is often a desired property. Sort-merge join has extra costs when you have to use external sorting (there is not enough memory to do the sort in-memory).

Block nested loops is efficient when one of the relations will fit in memory and you are using an MRU replacement strategy. However, if an index is available, there are better strategies available (but often indexes are not available).

7. If the join condition is not equality, you can use sort-merge join, index nested loops (if you have a range style index such as a B+ tree index or ISAM index), or block nested loops join. Hash joining works best for equality joins and is not suitable otherwise.
8. First we sort all of the tuples based on the `GROUP BY` attribute. Next we re-sort each group by sorting all elements on the `MAX` attribute, taking care not to re-sort beyond the group boundaries.
9. The operator `SECOND LARGEST` can be implemented using sorting. For each group (if there is a `GROUP BY` clause), we sort the tuples and return the second largest value for the desired attribute. The cost here is the cost of sorting.
10. One example where the buffer replacement strategy affects join performance is the use of LRU and MRU in a simple nested loops join. If the relations don't fit in main memory, then the buffer strategy is critical. Say there are M buffer pages and N are filled by the first relation, and the second relation is of size $M-N+P$, meaning all of the second relation will fit in the buffer except P pages. Since we must do repeated scans of the second relation, the replacement policy comes into play. With LRU, whenever we need to find a page it will have been paged out so every page request requires a disk IO. On the other hand, with MRU, we will only need to reread $P-1$ of the pages in the second relation, since the others will remain in memory.

Exercise 14.2 Consider a relation $R(a,b,c,d,e)$ containing 5,000,000 records, where each data page of the relation holds 10 records. R is organized as a sorted file with secondary indexes. Assume that $R.a$ is a candidate key for R , with values lying in the range 0 to 4,999,999, and that R is stored in $R.a$ order. For each of the following relational algebra queries, state which of the following approaches (or combination thereof) is most likely to be the cheapest:

- Access the sorted file for R directly.
 - Use a clustered B+ tree index on attribute $R.a$.
 - Use a linear hashed index on attribute $R.a$.
 - Use a clustered B+ tree index on attributes $(R.a, R.b)$.
 - Use a linear hashed index on attributes $(R.a, R.b)$.
 - Use an unclustered B+ tree index on attribute $R.b$.
1. $\sigma_{a < 50,000 \wedge b < 50,000}(R)$
 2. $\sigma_{a = 50,000 \wedge b < 50,000}(R)$
 3. $\sigma_{a > 50,000 \wedge b = 50,000}(R)$
 4. $\sigma_{a = 50,000 \wedge a = 50,010}(R)$
 5. $\sigma_{a \neq 50,000 \wedge b = 50,000}(R)$
 6. $\sigma_{a < 50,000 \vee b = 50,000}(R)$

Answer 14.2 Answer omitted.

Exercise 14.3 Consider processing the following SQL projection query:

```
SELECT DISTINCT E.title, E.ename FROM Executives E
```

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 *ename* 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).

1. How many sorted runs are produced in the first pass? What is the average length of these runs? (Assume that memory is utilized well and any available optimization to increase run size is used.) What is the I/O cost of this sorting pass?

2. 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?
3. (a) 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?
- (b) Suppose that a clustered B+ tree index on *ename* 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?
- (c) Suppose that a clustered B+ tree index on $\langle ename, title \rangle$ 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?
4. Suppose that the query is as follows:

```
SELECT E.title, E.ename FROM Executives E
```

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

Answer 14.3 The answer to each question is given below.

1. The first pass will produce 250 sorted runs of 20 pages each, costing 15000 I/Os.
2. Using the ten buffer pages provided, on average we can write 2×10 internally sorted pages per pass, instead of 10. Then, three more passes are required to merge the $5000/20$ runs, costing $2 \times 3 \times 5000 = 30000$ I/Os.
3. (a) Using a clustered B+ tree index on *title* would reduce the cost to single scan, or 12,500 I/Os. An unclustered index could potentially cost more than $2500 + 100,000$ (2500 from scanning the B+ tree, and $10000 \times$ tuples per page, which I just assumed to be 10). Thus, an unclustered index would not be cheaper. Whether or not to use a hash index would depend on whether the index is clustered. If so, the hash index would probably 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 12,500 I/Os. Since *ename* is a candidate key, no duplicate checking need be done for $\langle title, ename \rangle$ pairs. An unclustered index would require 2500 (scan of index) + $10000 \times$ tuples per page I/Os and thus probably be more expensive than sorting.
- (c) Using a clustered B+ tree index on $\langle ename, title \rangle$ would also be more cost-effective than sorting. An unclustered B+ tree over the same attributes

would allow an index-only scan, and would thus be just as economical as the clustered index. This method (both by clustered and unclustered) would cost 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 $\langle \textit{ename}, \textit{title} \rangle$ 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, a typical optimizer is not likely to recognize this and omit the duplicate elimination step.)

Exercise 14.4 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.

1. 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?
2. 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?
3. What is the cost of joining R and S using a sort-merge join? What is the minimum number of buffer pages required for this cost to remain unchanged?
4. What is the cost of joining R and S using a hash join? What is the minimum number of buffer pages required for this cost to remain unchanged?
5. 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.
6. 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?
7. 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*?

Answer 14.4 Answer omitted.

Exercise 14.5 Consider the join of R and S described in Exercise 14.1.

1. With 52 buffer pages, if unclustered B+ 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.
 - (a) Would your answer change if only five buffer pages were available?
 - (b) Would your answer change if S contained only 10 tuples instead of 2000 tuples?
2. With 52 buffer pages, if *clustered* B+ 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.
 - (a) Would your answer change if only five buffer pages were available?
 - (b) Would your answer change if S contained only 10 tuples instead of 2000 tuples?
3. If only 15 buffers were available, what would be the cost of a sort-merge join? What would be the cost of a hash join?
4. If the size of S were increased to also be 10,000 tuples, but only 15 buffer pages were available, what would be the cost of a sort-merge join? What would be the cost of a hash join?
5. If the size of S were increased to also be 10,000 tuples, and 52 buffer pages were available, what would be the cost of sort-merge join? What would be the cost of hash join?

Answer 14.5 Assume that it takes 3 I/Os to access a leaf in R, and 2 I/Os to access a leaf in S. And since S.b is a primary key, we will assume that every tuple in S matches 5 tuples in R.

1. The Index Nested Loops join involves probing an index on the inner relation for each tuple in the outer relation. The cost of the probe is the cost of accessing a leaf page plus the cost of retrieving any matching data records. The cost of retrieving data records could be as high as one I/O per record for an unclustered index.

With R as the outer relation, the cost of the Index Nested Loops join will be the cost of reading R plus the cost of 10,000 probes on S.

$$TotalCost = 1,000 + 10,000 * (2 + 1) = 31,000$$

With S as the outer relation, the cost of the Index Nested Loops join will be the cost of reading S plus the cost of 2000 probes on R.

$$TotalCost = 200 + 2,000 * (3 + 5) = 16,200$$

Neither of these solutions is cheaper than Block Nested Loops join which required 4,200 I/Os.

- (a) With 5 buffer pages, the cost of the Index Nested Loops joins remains the same, but the cost of the Block Nested Loops join will increase. The new cost of the Block Nested Loops join is

$$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 67,200$$

And now the cheapest solution is the Index Nested Loops join with S as the outer relation.

- (b) If S contains 10 tuples then we'll need to change some of our initial assumptions. Now all of the S tuples fit on a single page, and it will only require a single I/O to access the (single) leaf in the index. Also, each tuple in S will match 1,000 tuples in R.

Block Nested Loops:

$$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 1,001$$

Index Nested Loops with R as the outer relation:

$$TotalCost = 1,000 + 10,000 * (1 + 1) = 21,000$$

Index Nested Loops with S as the outer relation:

$$TotalCost = 1 + 10 * (3 + 1,000) = 10,031$$

Block Nested Loops is still the best solution.

2. With a clustered index the cost of accessing data records becomes one I/O for every 10 data records.

With R as the outer relation, the cost of the Index Nested Loops join will be the cost of reading R plus the cost of 10,000 probes on S.

$$TotalCost = 1,000 + 10,000 * (2 + 1) = 31,000$$

With S as the outer relation, the cost of the Index Nested Loops join will be the cost of reading S plus the cost of 2000 probes on R.

$$TotalCost = 200 + 2,000 * (3 + 1) = 8,200$$

Neither of these solutions is cheaper than Block Nested Loops join which required 4,200 I/Os.

- (a) With 5 buffer pages, the cost of the Index Nested Loops joins remains the same, but the cost of the Block Nested Loops join will increase. The new cost of the Block Nested Loops join is

$$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 67,200$$

And now the cheapest solution is the Index Nested Loops join with S as the outer relation.

- (b) If S contains 10 tuples then we'll need to change some of our initial assumptions. Now all of the S tuples fit on a single page, and it will only require a single I/O to access the (single) leaf in the index. Also, each tuple in S will match 1,000 tuples in R.

Block Nested Loops:

$$TotalCost = N + M * \lceil \frac{N}{B-2} \rceil = 1,001$$

Index Nested Loops with R as the outer relation:

$$TotalCost = 1,000 + 10,000 * (1 + 1) = 21,000$$

Index Nested Loops with S as the outer relation:

$$TotalCost = 1 + 10 * (3 + 100) = 1,031$$

Block Nested Loops is still the best solution.

3. SORT-MERGE: With 15 buffer pages we can sort R in three passes and S in two passes. The cost of sorting R is $2 * 3 * M = 6,000$, the cost of sorting S is $2 * 2 * N = 800$, and the cost of the merging phase is $M + N = 1,200$.

$$TotalCost = 6,000 + 800 + 1,200 = 8,000$$

HASH JOIN: With 15 buffer pages the first scan of S (the smaller relation) splits it into 14 buckets, each containing about 15 pages. To store one of these buckets (and its hash table) in memory will require $f * 15$ pages, which is more than we have available. 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 partitioning phases plus the cost of one matching phase.

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

4. SORT-MERGE: With 15 buffer pages we can sort R in three passes and S in three passes. The cost of sorting R is $2 * 3 * M = 6,000$, the cost of sorting S is $2 * 3 * N = 6,000$, and the cost of the merging phase is $M + N = 2,000$.

$$TotalCost = 6,000 + 6,000 + 2,000 = 14,000$$

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 partitioning 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 partitioning phase plus the cost of one matching phase.

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

Exercise 14.6 Answer each of the questions—if some question is inapplicable, explain why—in Exercise 14.1 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 14.6 Answer omitted.

Exercise 14.7 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 14.7 Each join method is considered in turn.

1. Sailors (S) NATURAL LEFT OUTER JOIN Reserves (R)
 In this LEFT OUTER JOIN, Sailor rows without a matching Reserves row will appear in the result with a *null* value for the Reserves value.
 - (a) *block nested loops join*
 In the *block nested loops join* algorithm, we place as large a partition of the Sailors relation in memory as possible, leaving 2 extra buffer pages (one for input pages of R, the other for output pages plus enough pages for a single bit for each record of the block of S. These 'bit pages' are initially set to zero; when a tuple of R matches a tuple in S, the bit is set to 1 meaning that this page has already met the join condition. Once all of R has been compared to the block of S, any tuple with its bit still set to zero is added to the output with a *null* value for the R tuple. This process is then repeated for the remaining blocks of S.
 - (b) *index nested loops join*
 An *index nested loops join* requires an index for Reserves on all attributes that Sailors and Reserves have in common. For each tuple in Sailors, if it matches a tuple in the R index, it is added to the output, otherwise the S tuple is added to the output with a *null* value.
 - (c) *sort-merge join*
 When the two relations are merged, Sailors is scanned in sorted order and if there is no match in Reserves, the Sailors tuple is added to the output with a *null* value.
 - (d) *hash join*
 We hash so that partitions of Reserves will fit in memory with enough leftover space to hold a page of the corresponding Sailors partition. When we compare a Sailors tuple to all of the tuples in the Reserves partition, if there is a match it is added to the output, otherwise we add the S tuple and a *null* value to the output.
2. Sailors NATURAL RIGHT OUTER JOIN Reserves
 In this RIGHT OUTER JOIN, Reserves rows without a matching Sailors row will appear in the result with a *null* value for the Sailors value.
 - (a) *block nested loops join*
 In the *block nested loops join* algorithm, we place as large a partition of the

Reserves relation in memory as possibly, leaving 2 extra buffer pages (one for input pages of Sailors, the other for output pages plus enough pages for a single bit for each record of the block of R. These 'bit pages' are initially set to zero; when a tuple of S matches a tuple in R, the bit is set to 1 meaning that this page has already met the join condition. Once all of S has been compared to the block of R, any tuple with its bit still set to zero is added to the output with a *null* value for the S tuple. This process is then repeated for the remaining blocks of R.

(b) *index nested loops join*

An *index nested loops join* requires an index for Sailors on all attributes that Reserves and Sailors have in common. For each tuple in Reserves, if it matches a tuple in the S index, it is added to the output, otherwise the R tuple is added to the output with a *null* value.

(c) *sort-merge join*

When the two relations are merged, Reserves is scanned in sorted order and if there is no match in Sailors, the Reserves tuple is added to the output with a *null* value.

(d) *hash join*

We hash so that partitions of Sailors will fit in memory with enough leftover space to hold a page of the corresponding Reserves partition. When we compare a Reserves tuple to all of the tuples in the Sailors partition, if there is a match it is added to the output, otherwise we add the Reserves tuple and a *null* value to the output.

3. Sailors NATURAL FULL OUTER JOIN Reserves

In this FULL OUTER JOIN, Sailor rows without a matching Reserves row will appear in the result with a *null* value for the Reserves value, and Reserves rows without a matching Sailors row will appear in the result with a *null* value.

(a) *block nested loops join*

For this algorithm to work properly, we need a bit for each tuple in both relations. If after completing the join there are any bits still set to zero, these tuples are joined with *null* values.

(b) *index nested loops join*

If there is only an index on one relation, we can use that index to find half of the full outer join in a similar fashion as in the LEFT and RIGHT OUTER joins. To find the non-matches of the relation with the index, we can use the same trick as in the *block nested loops join* and keep bit flags for each block of scans.

(c) *sort-merge join*

During the merge phase, we scan both relations alternating to the relation with the lower value. If that tuple has no match, it is added to the output with a *null* value.

(d) *hash join*

When we hash both relations, we should choose a hash function that will hash the larger relation into partitions that will fit in half of memory. This way we can fit both relations' partitions into main memory and we can scan both relations for matches. If no match is found (we must scan for both relations), then we add that tuple to the output with a *null* value.