COMP9315 19T2 DBMS Implementation

Exercises 08

Implementing Join: Nested-loop, Sort-merge, Hash

1. Does the buffer replacement policy matter for sort-merge join? Explain your answer.

   Answer:

   The sort phase requires multiple scans of progressively more sorted versions of the input files. On each scan, we are reading a new version of the file, and so buffering provides no real benefit. This will be the same regardless of the contents of the files. So, for the sorting phase, no buffer replacement strategy performs better than any other.

   For the merge phase, however, the performance is affected by the contents of the join attributes. We assume in the comments below that we are joining on a single attribute from each table, but the discussion applies equally if the join involves multiple attributes.

   If the "inner" table (the one whose file is scanned in the inner loop of the merge) contains no duplicate values in the join attribute, then the merge will be performed using a single scan through both files. In this case, the buffer replacement strategy does not matter (same rationale as for the sorting phase).

   On the other hand, if we merge two tables containing duplicated values in the join columns, then the buffer replacement strategy can affect the performance. In this case, we may need to visit pages of the inner relation several times if there are multiple occurrences of a given join value in the outer relation and if the matching tuples in the inner relation are held on separate pages. Under this scenario, we would not want to replace recently used pages, since they may need to be used again soon, and so an LRU replacement strategy would most likely perform better than an MRU replacement strategy.

2. Suppose that the relation *R* (with 150 pages) consists of one attribute *a* and *S* (with 90 pages) also consists of one attribute *a*. Determine the optimal join method for processing the following query:

   ```
   select * from R, S where R.a > S.a
   ```

   Assume there are 10 buffer pages available to process the query and there are no indexes available. Assume also that the DBMS only has available the following join methods: nested-loop, block nested loop and sort-merge. Determine the number of page I/Os required by each method to work out which is the cheapest.

   Answer:

   Simple Nested Loops:
   We use relation *S* as the outer loop. Total Cost = 90 + (90×150) = 13590

   Block Nested Loops:
   If *R* is outer: Total Cost = 150 + (90×ceil(150/(10-2))) = 1860
   If *S* is outer: Total Cost = 90 + (150×ceil(90/(10-2))) = 1890

   Sort-Merge:
   Denote B as the number of buffer pages, where B = 10; denote M as the number of pages in the larger relation, where M = 150. Since B < M , the cost on sort-merge is:

   - Sorting R: $2\times150\times(\text{ceil}(\log_{10-1}(150/10))+1) = 900$
   - Sorting S: $2\times90\times(\text{ceil}(\log_{10-1}(90/10))+1) = 360$
   - Merge: 150 + 90 = 240
     (This is the best case when only the maximum value in R.a is greater than the minimum value in S.a, otherwise, the worst case incurs 90*150 page I/Os)

Total Cost (best case) = 900 + 360 + 240 = 1500 (very unlikely)
Total Cost (worst case) = 900 + 360 + 13500 = 14760

Therefore, the optimal way to process the query is Block Nested Loop join.

3. [Ramakrishnan, exercise 12.4] Consider the join *Join[R.a=S.b](R,S)* and the following information about the relations to be joined:

   - Relation *R* contains 10,000 tuples and has 10 tuples per page
   - Relation *S* contains 2,000 tuples and also has 10 tuples per page
   - Attribute *b* of relation *S* is the primary key for *S*
   - Both of the relations are stored as simple heap files
   - Neither relation has any indexes built on it
   - There are 52 buffer pages available

   Unless otherwise noted, compute all costs as number of page I/Os, except that the cost of writing out the result should be uniformly ignored (it's the same for all methods).

   a. What is the cost of joining *R* and *S* using page-oriented simple nested loops join? What is the minimum number of buffer pages required for this cost to remain unchanged?

      Answer:

      The basic idea of nested-loop join is to do a page-by-page scan of the outer relation, and, for each outer page, do a page-by-page scan of the inner relation.

      The cost for joining *R* and *S* is minimised when the smaller relation *S* is used as the outer relation.

      Cost = $b_S + b_S * b_R$ = 200 + 200 * 1000 = 200,200.

      In this algorithm, no use is made of multiple per-relation buffers, so the minimum requirement is one input buffer page for each relation and one output buffer page i.e. 3 buffer pages.

   b. What is the cost of joining *R* and *S* using block nested loops join? What is the minimum number of buffer pages required for this cost to remain unchanged?

      Answer:

      The basic idea for block nested-loop join is to read the outer relation in *blocks* (groups of pages that will fit into whatever buffer pages are available), and, for each block, do a page-by-page scan of the inner relation.

      The outer relation is still scanned once, but the inner relation is scanned only once for each outer *block*. If we have *B* buffers, then the number of blocks is $ceil(b_{outer} / (B-2))$. As above, the total cost will be minimised when the smaller relation is used as the outer relation.

      Cost = $b_S + ceil(b_S/B-2) * b_R$ = 200 + ceil(200/50) * 1000 = 200 + 4 * 1000 = 4,200.

      If *B* is less than 52, then *B-2 < 50* and the cost will increase (e.g. *ceil(200/49)=5*), so 52 is the minimum number of pages for this cost.

   c. What is the cost of joining *R* and *S* using sort-merge join? What is the minimum number of buffer pages required for this cost to remain unchanged?

      Answer:

      The idea with sort-merge join is to sort both relations and then perform a single scan across each, merging into a single joined relation.

      Each relation has to first be sorted (assuming that it's not already sorted on the join attributes). This requires an initial pass, where chunks of the file of size *B* are read into memory and sorted. After this, a number of passes are done, where each pass does a *(B-*

1)-way merge of the file. Each pass reads and writes the file, which requires *2b* block reads/writes. The total number of passes is $1+ceil(\log_{B-1}ceil(b/B)))$.

Once the relations are sorted, the merge phase requires one pass over each relation, assuming that there are enough buffers to hold the longest run in either relation

Cost = $2b_S(1+ceil(\log_{B-1}ceil(b_S/B))) + 2b_R(1+ceil(\log_{B-1}ceil(b_R/B))) + b_S + b_R$
= $2\times200\times(1+ceil(\log_{51}ceil(200/52))) + 2\times1000\times(1+ceil(\log_{51}ceil(1000/52))) + 200 + 1000$
= $2.200.(1+ceil(\log_{51}4)) + 2.1000.(1+ceil(\log_{51}20)) + 200 + 1000$
= $2.200.2 + 2.1000.2 + 200 + 1000$ = 800 + 4000 + 200 + 1000 = 6,000.

The critical point comes when the value of $\log_{B-1}ceil(b_R/(B-1))$ exceeds 1. This occurs when *B* drops to 15 (and *B-1* becomes 14).

d. What is the cost of joining *R* and *S* using grace hash join? What is the minimum number of buffer pages required for this cost to remain unchanged?

Answer:

The basic idea with grace hash join is that we partition each relation and then perform the join by "matching" elements from the partitions. We need to assume that we have at least $sqrt(b_R)$ buffers or $sqrt(b_S)$ buffers and that the hash function gives a uniform distribution. Given that both *sqrt(200)* and *sqrt(1000)* are less than the number of buffers, the first condition is definitely satisfied.

Cost = $3.(b_S + b_R)$ = 3.(200 + 1000) = 3,600.

e. What would be the lowest possible I/O cost for joining *R* and *S* using *any* join algorithm? How much buffer space would be needed to achieve this cost?

Answer:

The minimal cost would be when each relation is read exactly once. We can perform such a join by storing the entire smaller relation in memory, reading in the larger relation page-by-page, and searching in the memory buffers for matching tuples for each tuple in the larger relation. The buffer pool would need to hold at least the same number of pages as the pages in the smaller relation, plus one input and one output buffer for the larger relation, giving

Pages = 200 + 1 + 1 = 202.

f. What is the maximum number of tuples that the join of *R* and *S* could produce, and how many pages would be required to store this result?

Answer:

Any tuple in *R* can match at most one tuple in *S* because *S.b* is a primary key. So the maximum number of tuples in the result is equal to the number of tuples in *R*, which is 10,000.

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 one copy of the join attribute). This size allows only 5 tuples to be stored per page, which means that 10,000/5 = 2,000 pages are required.

g. How would your answers to the above questions change if you are told that *R.a* is a foreign key that refers to *S.b*?

Answer:

The foreign key constraint tells us that for every *R* tuple there is exactly one matching *S* tuple. The sort-merge and hash joins would not be affected.

At first glance, it might seem that we can improve the cost of the nested-loop joins. If we make *R* the outer relation, then for each tuple of *R* we know that we only have to scan *S* until the single matching record is found, which would require scanning only 50% of *S* on

average. However, this is only true on a tuple-by-tuple basis. When we read in an entire block of *R* records and then look for matches for each of them, it's quite likely that we'll scan the entire *S* relation for every block of *R*, and thus would find no saving.

4. [Ramakrishnan, exercise 12.5] Consider the join of *R* and *S* described in the previous question:

   a. With 52 buffer pages, if *unclustered* B+ tree indexes existed on *R.a* and *S.b*, would either provide a cheaper alternative for performing the join (e.g. using index nested loop join) than a block nested loops join? Explain.

      i. Would your answer change if only 5 buffer pages were available?
      ii. Would your answer change if *S* contained only 10 tuples instead of 2,000 tuples?

   Answer:

   The idea is that we probe an index on the inner relation for each tuple from the outer relation. The cost of each probe is the cost of traversing the B-tree to a leaf node, plus the cost of retrieving any matching records. In the worst case for an unclustered index, the cost of reading data records could be one page read for each record. Assume that traversing the B-tree for the relation *R* takes 3 node accesses, while the cost for B-tree traversal for *S* is 2 node access. Since *S.b* is a primary key, assume that every tuple in *S* matches 5 tuples in *R*.

   If *R* is the outer relation, the cost will be the cost of reading *R* plus, for each tuple in *R*, the cost of retrieving the data

   Cost = $b_R + r_R * (2 + 1)$ = 1,000 + 10,000*3 = 31,000.

   If *S* is the outer relation, the cost will be the cost of reading *S* plus, for each tuple in *S*, the cost of retrieving the data

   Cost = $b_S + r_S * (3 + 5)$ = 200 + 2,000*8 = 16,200

   Neither of these is cheaper than the block nested-loops join which required 4,200 page I/Os.

   With 5 buffer pages, the cost of index nested-loops join remains the same, but the cost of the block nested-loops join increases. The new cost for block nested-loops join now becomes

   Cost = $b_S + b_R * ceil(b_S/B-2)$ = 200 + 1000 * ceil(200/3) = 200 + 1000 * 67 = 67,200.

   Now the cheapest solution is index nested-loops join.

   If *S* contains only 10 tuples, then we need to change some of our initial assumptions. All of the tuples of *S* now fit on a single page, and it requires only a single I/O to access the leaf node in the index. Also, each tuple in *S* matches 1,000 tuples in *R*.

   For block nested-loops join

   Cost = $b_S + b_R * ceil(b_S/B-2)$ = 1 + 1000 * ceil(1/50) = 1 + 1000 * 1 = 1,001.

   For index nested-loops join, with *R* as the outer relation

   Cost = $b_R + r_R * (1 + 1)$ = 1000 + 10000 * 2 = 21,000.

   For index nested-loops join, with *S* as the outer relation

   Cost = $b_S + r_S * (3 + 100)$ = 1 + 10 * (3 + 1000) = 10,031.

   Block nested-loops is still the best solution.

b. With 52 buffer pages, if *clustered* B+ tree indexes existed on *R.a* and *S.b*, would either provide a cheaper alternative for performing the join (e.g. using index nested loop join) than a block nested loops join? Explain.

   i. Would your answer change if only 5 buffer pages were available?
   ii. Would your answer change if *S* contained only 10 tuples instead of 2,000 tuples?

Answer:

With a clustered index, the cost of accessing data records becomes one page I/O for every 10 records. Based on this, we assume that for each type of index nested-loop join, that the data retrieval cost for each index probe is 1 page read.

Thus, with *R* as the outer relation

Cost = $b_R + r_R * (2 + 1)$ = 1000 + 10000 * 3 = 31,000.

With *S* as the outer relation

Cost = $b_S + r_S * (3 + 1)$ = 200 + 2000 * 4 = 8,200.

Neither of these solutions is cheaper than block nested-loop join.

With 5 buffer pages, the cost of index nested-loops join remains the same, but the cost of the block nested-loops join increases. The new cost for block nested-loops join now becomes

Cost = $b_S + b_R * ceil(b_S/B-2)$ = 200 + 1000 * ceil(200/3) = 200 + 1000 * 67 = 67,200.

Now the cheapest solution is index nested-loops join.

If *S* contains only 10 tuples, then we need to change some of our initial assumptions. All of the tuples of *S* now fit on a single page, and it requires only a single I/O to access the leaf node in the index. Also, each tuple in *S* matches 1,000 tuples in *R*.

For block nested-loops join

Cost = $b_S + b_R * ceil(b_S/B-2)$ = 1 + 1000 * ceil(1/50) = 1 + 1000 * 1 = 1,001.

For index nested-loops join, with *R* as the outer relation

Cost = $b_R + r_R * (1 + 1)$ = 1000 + 10000 * 2 = 21,000.

For index nested-loops join, with *S* as the outer relation

Cost = $b_S + r_S * (3 + 100)$ = 1 + 10 * (3 + 100) = 1,031.

Block nested-loops is still the best solution.

c. If only 15 buffers were available, what would be the cost of sort-merge join? What would be the cost of hash join?

Answer:

Sort-merge join:

With 15 buffers, we can sort *R* in 3 passes and *S* in 2 passes.

Cost = $2.b_R.3 + 2.b_S.2 + b_R + b_S$ = 2.1000.3 + 2.200.2 + 1000 + 200 = 8,000.

Hash join:

With 15 buffer pages the first scan of *S* (the smaller relation) splits it into 14 partitions, each containing (on average) 15 pages. Unfortunately, these partitions are too large to fit into the

memory buffers for the second pass, and so we must apply hash join again to all of the partitions produced in the first partitioning phase. Then we can fit an entire partition of *S* in memory. The total cost is the cost of two partitioning phases plus the cost of one matching phase.

Cost = $2.(2.(b_R + b_S)) + (b_R + b_S)$ = $2.(2.(200+1000)) + (200+1000)$ = 6,000.

d. 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 for sort-merge join? What would be the cost of hash join?

Answer:

Sort-merge join:

With 15 buffers, we can sort *R* in 3 passes and *S* in 3 passes.

Cost = $2.b_R.3 + 2.b_S.2 + b_R + b_S$ = $2.1000.3 + 2.1000.3 + 1000 + 1000$ = 14,000.

Hash join:

Now that both relations are the same size, we can treat either of them as the smaller relation. Let us choose *S* as before. With 15 buffer pages the first scan of *S* splits it into 14 partitions, each containing (on average) 72 pages. As above, these partitions are too large to fit into the memory buffers for the second pass, and so we must apply hash join again to all of the partitions produced in the first partitioning phase. Then we can fit an entire partition of *S* in memory. The total cost is the cost of two partitioning phases plus the cost of one matching phase.

Cost = $2.(2.(b_R + b_S)) + (b_R + b_S)$ = $2.(2.(1000+1000)) + (1000+1000)$ = 10,000.

e. If the size of *S* were increased to also be 10,000 tuples, and 52 buffer pages were available, what would be the cost for sort-merge join? What would be the cost of hash join?

Answer:

Sort-merge join:

With 52 buffers, we can sort both *R* and *S* in 2 passes.

Cost = $2.b_R.3 + 2.b_S.2 + b_R + b_S$ = $2.1000.2 + 2.1000.2 + 1000 + 1000$ = 10,000.

Hash join:

Both relations are the same size, so we arbitrarily choose *S* as the "smaller" relation. With 52 buffer pages the first scan of *S* splits it into 51 partitions, each containing (on average) 14 pages. This time we do not have to deal with partition overflow, and so only one partitioning phase is required before the matching phase.

Cost = $2.(b_R + b_S) + (b_R + b_S)$ = $2.(1000+1000) + (1000+1000)$ = 6,000.

5. Consider performing the join:

```
select * from R, S where R.x = S.y
```

where we have b_R = *1000*, b_S = *5000* and where *R* is used as the outer relation.

Compute the page I/O cost of performing this join using *hybrid hash join*:

a. if we have *N=100* memory buffers available

Answer:

The first step with hybrid hash join is to determine how many partitions we are going to use. Recall that one partition of the outer relation will be memory-resident, while all other partitions will be written/read to/from disk. The aim is to minimise the number of partitions, so that we have as large a partition as possible resident in memory.

To compute the number of partitions, choose the smallest number larger than b_R/N which ensures that both the memory-resident partition and enough buffers for the other partitions can fit into memory. This means choosing a value for the number of partitions k that satisfies the following: $k \approx ceil(b_R/N)$ and $ceil(b_R/k)+k \leq N$.

For $N=100$ buffers, $k \approx ceil(b_R/N)=10$, but $ceil(b_R/k)+k=110$, which is more than the number of available buffers. If we choose $k=11$, we then have $ceil(b_R/k)+k=91+11=102$, which is still more than the number of available buffers. If we choose $k=12$, we then have $ceil(b_R/k)+k=84+12=96$, which is satisfactory. (If we didn't want to waste any buffers, then we could allocate the extra 4 to the in-memory partition, but we'll ignore that for this exercise)

Once k is determined, the cost is easy to compute

Cost $= (3-2/k) \times (b_R+b_S) = (3-2/12) \times (1000+5000) = 17{,}000$

b. if we have $N=512$ memory buffers available

Answer:

For $N=512$ buffers, $k \approx ceil(b_R/N)=2$, and $ceil(b_R/k)+k=502$, which fits in the avaialable buffer space.

Cost $= (3-2/k) \times (b_R+b_S) = (3-2/2) \times (1000+5000) = 12{,}000$

c. if we have $N=1024$ memory buffers available

Answer:

With 1024 buffers, we can hold the whole of R in memory, so we can compute the minimum cost join using the simpler nested loop join method. In this case, the gain from using hybrid hash join is unclear. Nevertheless, we'll do the calculation anyway ...

For $N=1024$ buffers, $k \approx ceil(b_R/N)=1$, and $ceil(b_R/k)+k=1001$, which fits in the avaialable buffer space.

Cost $= (3-2/k) \times (b_R+b_S) = (3-2/1) \times (1000+5000) = 6{,}000$

Happily, the computation confirms that hybrid hash join gives the minimum possible cost for this scenario.