CSC 370

Activity Worksheet:
Join Algorithms

Dr. Sean Chester

Fall 2022

Notes

In this worksheet, you will practice join algorithms within the I/O model, using the algorithmic cost to assess correctness. The questions are split into two sections: one focused on walking through the algorithms and the other on applying your knowledge of the join algorithms to determine which attributes would benefit from an index. The first question of each section has been answered already as an exemplar.

Questions

1. (Algorithm Logic and Analysis)

   Assume that you have two relations, R(y) and S(x,y), where x and y are both 4B integers, R contains 1000 tuples, S contains 250 tuples, you have 400B of memory available, and each block fits 100B. What is the I/O cost of using nested-loops join to answer the query below? Assume that the result will stay in main memory, i.e., do not include the output blocks for the final result.

   ```
   SELECT ‘x‘
   FROM ‘R‘
     NATURAL JOIN ‘S‘;
   ```

   Solution:

   We start by determining the number of blocks for the relevant operands:

   - Blocks of memory, i.e., the number of memory buffers, $B(M) = 400B/100B = 4$ blocks.
   - Blocks of R, $B(R) = (1 * 4B) * 1000/100B = 40$ blocks.
   - Blocks of S, $B(S) = (2 * 4B) * 250/100B = 10$ blocks

   Since S occupies fewer blocks than R, we set it as the outer loop, which then has $T(S) = 250$ iterations.

   The inner loop scans R contiguously, which has cost $B(R) = 40$. Therefore, the total inputs attributable to R are $T(S)B(R) = 250 * 40 = 10000$ blocks.

   The outer loop scans S contiguously, which has cost $B(S) = 10$. Therefore, the total inputs attributable to S are $B(S) = 10$ blocks.

   Therefore, in total, the nested loops algorithm has cost $B(S) + T(S)B(R) = 10 + 10000 = 10010$ blocks for this input configuration.

2. (Algorithm Logic and Analysis)

Assume that you have two relations, R(y) and S(x,y), where x and y are both 4B integers, R contains 1000 tuples, S contains 250 tuples, you have 400B of memory available, and each block fits 100B. How many I/O's are saved by using block-nested-loops join instead of nested loops join to answer the query below?

```
SELECT 'x'
FROM 'R'
   NATURAL JOIN 'S';
```

Solution:

3. (Algorithm Logic and Analysis)

Assume that you have two relations, R(y) and S(x,y), where x and y are both 4B integers, R contains 1000 tuples, S contains 250 tuples, you have 400B of memory available, and each block fits 100B. Moreover, you have a B+-tree index on R.y with 8B pointers. How many I/O's are saved by using an index join instead of block-nested loops join to answer the query below? (Hint: you need to use the information in the question to determine the fan-out of the tree; you can assume that each node contains as many keys and pointers as possible.)

```
SELECT 'x'
FROM 'R'
   NATURAL JOIN 'S';
```

Solution:

4. (Index Selection)

Assume that you have two relations, R(y) and S(x,y), where x and y are both 4B integers, R contains 200 tuples, S contains 1000 tuples, you have 400B of memory available, and each block fits 100B. Moreover, assume that there are no indexes on either R or S. If you want to accelerate the query below, what is the best secondary index to construct?

```
SELECT 'x'
FROM 'R'
  NATURAL JOIN 'S';
```

Solution:

We start by determining the number of blocks for the relevant operands:

- Blocks of memory, i.e., the number of memory buffers, $B(M) = 400B/100B = 4$ blocks.
- Blocks of R, $B(R) = (1 * 4B) * 200/100B = 8$ blocks.
- Blocks of S, $B(S) = (2 * 4B) * 1000/100B = 80$ blocks

We observe that there are five possible choices of index:

- ON (R.y)
- ON (S.x)
- ON (S.y)
- ON (S.x, S.y)
- ON (S.y, S.x)

We also observe the following conditions in our problem:

- Since R occupies fewer blocks, it would naturally be a nice outer loop relation.
- For any given R.y value, we want the corresponding S.y and vice versa.
- We eventually want attribute S.x.

Thus, we can evaluate each of the options:

- ON (R.y): this requires making S the outer loop relation, which could be up to 80/8 = 10x worse
- ON (S.x): this is not helpful for the join, because the join is on attribute S.y
- ON (S.x, S.y): this is not helpful for the join either, because it orders S.x first; it has the same efficacy as an index on S.x
- ON (S.y): this is more appealing, because it makes R the outer loop variable; however, it requires looking up S.x for each result tuple, which could add T(O) extra I/O's
- ON (S.y, S.x): this ensures that once we probe S.y, the value S.x is also available in the index; however, it decreases the fan-out of the B+-tree because the two-element key is more expensive to store. If it increases the height of the B+-tree, then it adds T(R) I/O's to the index probes.

In conclusion, we prefer an index ON (S.y) or ON (S.y, S.x), depending on T(O).

5. (Index Selection)

Assume that you have two relations, R(y) and S(x,y), where x and y are both 4B integers, R contains 1000 tuples, S contains 200 tuples, you have 400B of memory available, and each block fits 100B. Moreover, assume that there are no indexes on either R or S. If you want to accelerate the query below, what is the best secondary index to construct?

```
SELECT 'x'
FROM 'R'
  NATURAL JOIN 'S';
```

Solution:

Solutions

Question 1

We start by determining the number of blocks for the relevant operands:

- Blocks of memory, i.e., the number of memory buffers, $B(M) = 400B/100B = 4$ blocks.
- Blocks of R, $B(R) = (1 * 4B) * 1000/100B = 40$ blocks.
- Blocks of S, $B(S) = (2 * 4B) * 250/100B = 10$ blocks

Since S occupies fewer blocks than R, we set it as the outer loop, which then has $T(S) = 250$ iterations.

The inner loop scans R contiguously, which has cost $B(R) = 40$. Therefore, the total inputs attributable to R are $T(S)B(R) = 250 * 40 = 10000$ blocks.

The outer loop scans S contiguously, which has cost $B(S) = 10$. Therefore, the total inputs attributable to S are $B(S) = 10$ blocks.

Therefore, in total, the nested loops algorithm has cost $B(S) + T(S)B(R) = 10 + 10000 = 10010$ blocks for this input configuration.

Question 2

We know from Question 1 that nested-loops join uses 10010 blocks of I/O (excluding output, which is the same in both cases so cancels out of the comparison).

We start again by determining the number of blocks for the relevant operands:

- Blocks of memory, i.e., the number of memory buffers, $B(M) = 400B/100B = 4$ blocks.
- Blocks of R, $B(R) = (1 * 4B) * 1000/100B = 40$ blocks.
- Blocks of S, $B(S) = (2 * 4B) * 250/100B = 10$ blocks

We see that memory has four blocks available, one of which we need for the inner loop and one of which we need to buffer output. That leaves two buffers/blocks for the outer loop relation.

Since S occupies fewer blocks than R, we set it as the outer loop, which then has $B(S)/2 = 5$ iterations.

The inner loop scans R contiguously, which has cost $B(R) = 40$. Therefore, the total inputs attributable to R are $B(S)/2 * B(R) = 5 * 40 = 200$ blocks.

The outer loop scans S contiguously, which has cost $B(S) = 10$. Therefore, the total inputs attributable to S are $B(S) = 10$ blocks.

Therefore, in total, the block-nested-loops algorithm has cost $B(S) + B(S)/2 * B(R) = 10 + 200 = 210$ blocks for this input configuration.

Therefore, we save $\Delta = 10010 - 210 = 9800$ blocks of I/O, or, stated relatively, $9800/10010 = 97.9\%$ of I/O's.

Question 3

We know from Question 2 that block-nested-loops join uses 210 blocks of I/O (excluding output, which is the same in both cases so cancels out of the comparison).

We start by determining the height of the B+-tree:

- If we have k keys, then we have $k + 1$ pointers
- Keys here are 4B integers and pointers are 8B
- Therefore, we want to maximise k, subject to $4k + 8(k + 1) \leq 100$
- Hence $k = 7$
- Assuming, as suggested, that each node is full, the height of the tree is thus:

$$h(T) = \lceil \log_8(T(R)) \rceil = \lceil \log_8(1000) \rceil = 4$$

.

We see that memory has four blocks available, one of which we need for the inner loop and one of which we need to buffer output. That leaves two buffers/blocks. We can use one to cache the root of the tree, effectively reducing its height by 1. The other can scan blocks of S.

Our outer loop scans S contiguously and thus has a cost of $B(S) = 10$ blocks.

We incur one I/O to input the root of the B+-tree for $R.y = 1$ block.

Finally, in the inner loop, for each tuple of S, we probe the index on R.y to locate a matching tuple (if it exists). The cost is fixed at 3 I/O's for each probe, so the total cost is $3T(S) = 3(250) = 750$ blocks.

Therefore, the index-based join costs an additional $750 - 210 = 540$ blocks of I/O, relative to block-nested-loops.

(Note: that BNL performs better is an artefact of the very low fan-out in the B+-tree, limited memory, and the relatively equal sizes of R and S, which leads to a very high number of index probes. In reality, this entire B+-tree would fit in memory.)

Question 4

We start by determining the number of blocks for the relevant operands:

- Blocks of memory, i.e., the number of memory buffers, $B(M) = 400B/100B = 4$ blocks.
- Blocks of R, $B(R) = (1 * 4B) * 200/100B = 8$ blocks.
- Blocks of S, $B(S) = (2 * 4B) * 1000/100B = 80$ blocks

We observe that there are five possible choices of index:

- ON (R.y)
- ON (S.x)
- ON (S.y)
- ON (S.x, S.y)
- ON (S.y, S.x)

We also observe the following conditions in our problem:

- Since R occupies fewer blocks, it would naturally be a nice outer loop relation.
- For any given R.y value, we want the corresponding S.y and vice versa.
- We eventually want attribute S.x.

Thus, we can evaluate each of the options:

- ON (R.y): this requires making S the outer loop relation, which could be up to 80/8 = 10x worse
- ON (S.x): this is not helpful for the join, because the join is on attribute S.y
- ON (S.x, S.y): this is not helpful for the join either, because it orders S.x first; it has the same efficacy as an index on S.x
- ON (S.y): this is more appealing, because it makes R the outer loop variable; however, it requires looking up S.x for each result tuple, which could add T(O) extra I/O's
- ON (S.y, S.x): this ensures that once we probe S.y, the value S.x is also available in the index; however, it decreases the fan-out of the B+-tree because the two-element key is more expensive to store. If it increases the height of the B+-tree, then it adds T(R) I/O's to the index probes.

In conclusion, we prefer an index ON (S.y) or ON (S.y, S.x), depending on T(O).

Question 5

This question is much simpler than Question 4, because the projection is applied to the smaller relation. Therefore, we can scan S, obtaining both attributes, and then use the index just to probe R.y.

We prefer an index ON (R.y).