School of Computer Science, McGill University

COMP-421 Database Systems, Winter 2018

Written Assignment 3: Query Evaluation

Solution Outline

This is an individual assignment. You are required to work on your own to create the solution.

In this assignment, we evaluate queries for a manufacturer database. We look at three relations `Suppliers`, `Parts`, and `Catalog`, listed below.

Suppliers	(<u>sid</u>:CHAR(10), sname:CHAR(40), address:CHAR(160), country:CHAR(20))
	's3', 'BigSupp', '1180 Rue. St. Mathieu, Montreal', 'Canada'
Parts	(<u>pid</u>:INT, pname:CHAR(20), color:CHAR(20), memo:CHAR(200))
	2, 'bearing', 'brown', 'Memo for bearing'
Catalog	(<u>pid</u>:INT, <u>sid</u>:CHAR(10), productionYear:INT, price:FLOAT)
	⟶ pid references Parts, sid references Suppliers.
	2, 's3', 2003, 205

INT and FLOAT have 10 Bytes, a char has 1 Byte. The relation `Suppliers` has around 5,000 tuples on 350 data pages. The relation `Parts` has 20,000 tuples that are stored on 1500 pages. Each part has on average 10 different suppliers, i.e., 10 catalog entries. Hence, `Catalog` has around 200,000 entries on 3000 pages.

For indexes, a single data entry might be spread over more than one leaf page. Assume there are 2000 different prices. Each rid has 10 Bytes, each pointer (of internal index pages) has 6 Bytes. Leaf pages are filled on an average 75%. An index page has 4KBytes (use 4000 Bytes). The root might have any fill factor. Assume that the root and intermediate pages are in memory.

Ex. 1 — (24 Points)

One typical query checks the `Catalog` table for a given part `X` and a range of price `Y`. `X` and `Y` represent parameters that might differ for each execution.

```
SELECT *
FROM  Catalog
WHERE pid = X AND price <= Y
```

Assume the price is uniformly distributed between $1 to $2000.

(a) Indicate the access costs (in number of pages retrieved leading to I/O) for this query in each of the scenarios: (Give the access costs both in general (for `X` and `Y`), and for the concrete values: X=1123 and Y=500.)

- (i) (4 Points) you do not use any index.
- (ii) (6 Points) you use an unclustered index on price.
- (iii) (6 Points) you use an unclustered index on pid.
- (iv) (4 Points) you use both indexes.

(b) (4 Points) Would a clustered index on either `price` or `pid` increase performance? Give a short explanation.

Answer (Ex. 1) —

(a) (i) no Index: scan the relation: 3000 pages for all values of `X` and `Y`.

(ii) unclustered index on `price`:

200,000 / 2000 is 100 rids. Each data entry has around 10 + 10*100 = 1010 Bytes. There are 2000 different data entries. Also 1010 * 2000 / (0.75 * 4000) = 674 leaves.
According to the distribution of price, the reduction factor is R = Y / 2000 (R=0.25 if Y=500). Root and inner nodes are assumed to be in main memory. We need to read 674 * R index leaf pages. Hence, the total I/O is 674 * R + 200,000 * R data pages, in the worst case. For Y = 500, we need to read 169 leaves, and 50,000 data pages accessed. We could sort the entries in the 169 leave pages according to page-id and then retrieve every data page only once. However, given that each data page contains more than 60 tuples, R must be really small before there are a considerable number of data pages that do not contain a matching tuple. Hence, using the index will only be worth if R is really small.

(iii) unclustered index on `pid`:

Each part has around 10 entries and I have to find them with the index. To do so, I just need to read 1 leaf page. Hence, the total I/O is 1 leaf + 10 data pages (since access is random, I will access around 10 pages for 10 tuples). Independent of concrete value of X.

(iv) There are at most 10 target tuples. Get leaf of index on pid that has value of X. Keep the 10 rids in main memory. Get the leaves on index on price for price $<$ Y . For each rid, check whether among the 10 rids of pids. If yes, get data page. Hence we read 1 leaf of index on pid, R*674 leaves of the index on price, and less than 10 data pages (R*10 to be precise) . For Y=500, this accesses about 173 pages. It is larger than simply using the pid index, getting 10 tuples and then checking for price.

(b) Clustered indices would help. In case of price, all tuples with same price would be on adjacent data pages, requiring only to retrieve R*3000 data pages to retrieve (+ the index leave pages). For index on pid , all tuples with same pid are on one or at most 2 data pages.

Ex. 2 — (20 Points)

To make your computation easier, you can assume an extra one or two buffer page if you want.
Now assume there is an index on pid on Parts. Calculate the estimated I/O and give an estimate of the number of output tuples for an:

(a) (6 Points) index nested loop join between `Parts` and `Catalog`.

(b) (6 Points) block nested loop join between `Parts` and `Catalog` and `Parts` is outer relation (50 buffer pages available).

(c) (8 Points) sort merge join between `Parts` and `Catalog` (50 buffer pages available).

Answer (Ex. 2) —

We will have 200,000 tuples in the output, same as the cardinality of the `Catalog` table. This is because every catalog entry belongs to exactly one part.

(a) `Catalog` is outer: read 3000 pages from `Catalog`; for each tuple of `Catalog`, read leaf page + data page to get matching tuple from `Parts` (only one). 3,000 + 200,000 * 2 = 403,000.

(b) `Parts` is outer. Hence, we need 1500 I/Os to scan `Parts`. For each 50-page block of `Parts`, we have to scan `Catalog`, and hence the cost is: (1500 / 50) * 3000 = 90,000. Totally, the cost is 1,500 + 90,000 = 91,500.

(c) Sort `Parts` in 2 passes. pass 0 (makes 1500 pages/50 buffers = 30 runs) and pass 1 to merge it. Sort `Catalog` in 3 passes. pass 0 (makes 60 runs) , pass 1 (makes 2 runs) and pass 2 to merge it. This is followed by the actual merge join step that reads all the sorted pages of both the tables and performs the join. Cost is (2*2*1500 + 2*3*3000) + (1500 + 3000) = 28,500.
Optimization possible by combining merge join with the last pass of sorting; if done in solution, description of optimization must be included. In this case pass 1 of Parts and pass 2 of Catalog will be run simultaneously, their output pages being immediately pipelined to the join operator. This reduces 1 set of writes (from last pass of sorting) and 1 set of reads (read step of join) from the total cost from both tables.

In this case cost is 2*1500 + 2*2*3000 + 1500 + 3000 = 19,500.

Ex. 3 — **(36 Points)**

We will now have a look at the following query:

```
SELECT S.sname
FROM Suppliers S, Parts P, Catalogs C
WHERE  P.pname= 'bearing'
  AND  S.sid = C.sid
  AND  P.pid = C.pid
  AND  S.country = 'China'
```

A non-optimized relational expression for this query is:
$\pi_{sname}(\sigma_{country='China' \wedge pname='bearing'}(Suppliers \times Parts) \bowtie Catalog)$

(a)(6 Points) Perform an algebraic optimization of this expression according to the rules discussed in class. Do not consider any data cardinalities for this sub-question. Write down the resulting relational algebra expression (no need to draw any tree).

(b)(30 Points)

Now assume the only existing index is on sid of `Catalog`. Give an execution plan for your optimal expression and indicate how you are going to execute each of the operators. Draw the execution plan tree for this. You need not include number of rows and bytes passing through the operators in the tree that you are drawing.

Give rough estimations of I/O costs for your execution plan. Assume that you have 50 buffers available.

Note that our query optimizer, if it does not know how many different values exist for a given attribute, it assumes that there are 10 different values (uniformly distributed) by default. For instance, it assumes that suppliers come from 10 different countries.

Answer (Ex. 3) —

(a)$\pi_{sname}(\pi_{pid,sname}(\pi_{sid,sname}(\sigma_{country='China'}(Suppliers)) \bowtie \pi_{pid,sid}(Catalog)) \bowtie \pi_{pid}(\sigma_{pname} =' bearing'(Parts)))$

(b)Since there is an index on the foreign key sid of Catalog to Suppliers, this seems like a possible candidate for an index nested loop, with Catalog as inner relation.

Join between Suppliers and Catalog: Around 500 suppliers match China condition (as optimizer assumes that there are only 10 distinct countries). Also we have 200,000/5,000 = 40 parts per supplier. Index nested loop needs 350 supplier pages + 500 suppliers * (1 leaf + 40 data pages of catalog) = 20,850 page reads.

If we instead used block nested loop, we can read in 350 pages from Suppliers, find 500 suppliers, take only sid and sname and put it on 7 pages. (which fits into memory: 50 bytes per tuple * 500 tuples = 25,000 bytes.) Now read Catalog once (number of blocks = 1 as the outer relation is completely in the memory): Therefore total page reads is 350 + 1*3000 = 3350.

For sort-merge join, we would need to sort Suppliers and Catalog, with multiple passes and that is evidently more expensive than block nested loop. Hence, we can chose block nested loop as our join algorithm for this step.

For the join between the output "T" obtained from Suppliers $\bowtie$ Catalog (the previous step) and Parts, T has 20,000 tuples (40 Parts * 500 suppliers from China) each having 50 Bytes (after projection of pid, sname), they would need 250 pages to fit.

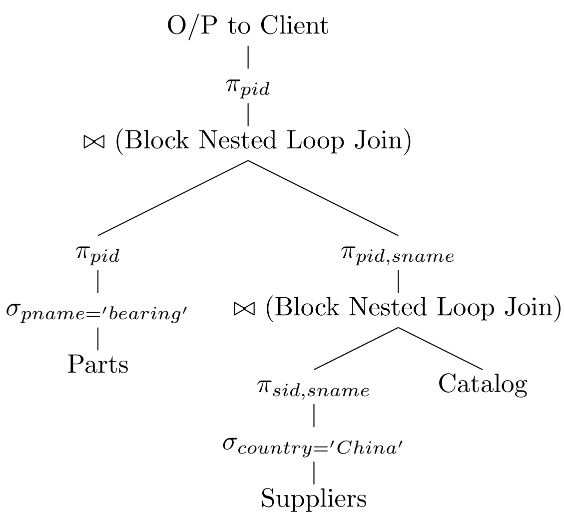
Parts, after the selection and projection has around 2000 tuples (optimizer assumes there are 10 different values for pname, hence 20,000/10 tuples qualify.), each only with 10 Bytes (projecting only pid), hence around 5 pages is enough.
Make a block nested loop join with the projected Parts as outer, and T as inner.

So the final approach and total costs are:
Read in Parts first, write result out on 5 pages (which fit into main memory) no need to write this to disk. That is 1500 I/O.

Read in Suppliers second, write result to 7 pages which also fits into main memory. That is 350 I/O.

Read in Catalog to join with Suppliers. That is 3000 I/O.

From here, everything else happens in main memory. (Each output buffer of Supplier $\bowtie$ Catalog can be immediately pipelined to the block nested join operator joining it with Parts and that operator's output itself can be immediately send out to the client. Hence none of those operations incurr any additonal I/O as we are not writing their output into disk).
(Of course, other calculations that look reasonable will be accepted).



Ex. 4 — (20 Points)

Consider the following query that calculates the distinct number of parts produced in each country.

```
SELECT S.country, COUNT(DISTINCT C.pid)
FROM Suppliers S, Catalog C
WHERE S.sid = C.sid
GROUP BY S.country
```

Find the optimal execution plan and its cost (you need not draw the algebraic optimization tree in the solution, but doing it might help you find the solution faster.).
Assume that both `Suppliers` and `Catalog` have indexes on their primary keys. For multi-attribute indexes, the index order is the same as the order in their table definition.
Assume you have 50 buffers. (Assume another extra buffer or two if it makes your computation easy).

Answer (Ex. 4) — For index nested join, if we used `Suppliers` as outer to use the index on `Catalog`, we will end up reading all the leaf pages of `Catalog`'s index on `(pid, sid)`. That is because the index is sorted first on `pid`. Hence a given `sid` could be in any leaf.

Since a data entry is 10+10+10 = 30 bytes, on an average there are 4000*0.75 / 30 = 100 data entries in a leaf page. So the index has 200000/100 = 2000 leaf pages.

Thus, the cost of this approach is 350 + 5000 * 2000 = 10,000,350.

That is a lot. Let us see what happens if we instead made `Catalog` the outer relation and used the index on `sid` of `Suppliers`.
As `sid` is unique in `Suppliers`, the cost of accessing records for a given `sid` is (1 leaf + 1 data page). So the cost of index nested join will be. 3000 + 200000 * 2 = 403,000

Seems to be better than the first option, but what about using block nested join ?
Here the cost is 350 + 350/50 * 3000 = 21,350

That is way better !! Lesson learned:
Multi-attribute indexes may not be always useful when the search is only on one of the attributes. Index nested joins can be terrible if there are no selection conditions on the outer table and we are reading a lot of records as result from the inner table.

Can we do even better ?
If we first project (sid, country) from Suppliers it will fit into = (30*5000 / 4000) = 38 pages. That fits in the memory.
Now the block nested join cost comes further down as the entire outer relation fits into the memory(i.e. 1 block) So the cost is 350 + 1*3000 = 3,350
This is the cheapest it will get (remember even for hash joins, the minimal cost is the cost of reading both the tables. So no need to compute it, as it will not be better than this).
As the output of the join can be projected for only the columns `(country, pid)`, that is 30 bytes, and has 200000 records. So we need 30*200000 / 4000 = 1500 pages to store it. Have to write it to the disk.

For grouping, we need to sort on `country, pid`. For 1500 pages, we need Pass 0 (30 runs) and Pass 1 (for merging). Cost of sorting = 2*2*1500 = 6,000

But we can remove the cost of the last pass of sort by pipelining the output of Pass 1 to the aggregation. Optimized cost would be 2*1500 + 1500 = 4,500

At the aggregation we keep track of the last `country` and `pid` that we saw. We increment a counter each time we see a different `pid` than we just saw immediately before it for the same `country` (remember the input to this operator is sorted first on `country` and then on `pid`). When we see a new `country` (or run out of input), we can output the current country's name and the value of the counter (which now means the distinct pids seen for that country) and reset it.
As this output is directly send to the client, we have no I/O cost for this.

So the total cost is.
3,350 for the block nested join, 1,500 to write its output, 4,500 to sort = 9,350.