Weekly exercises IN3020&4020 (2026)
Week 7: Query Evaluation

1 Review

1. What is the difference between logical query plan and physical query plan?

   Solution: A logical query plan is a relational algebra expression (this term is used in the context of query processing). A physical query plan is a logical plan where a dedicated algorithm is additionally assigned to each operation.

2. What is our cost for evaluation of an operation (not to be confused with the cost of a logical plan for optimisation)? What are our main assumptions?

   Solution: The cost is the number of data block reads and writes. The assumptions are: the input to the algorithm is on disk, but the output is written to the main memory; relation is clustered (i.e., tightly packed); other costs, including manipulation with indexes, is ignored.

3. Assuming you want to retrieve the entire relation, what is the IO cost of using a table scan? Same for the index scan?

   Solution:

   Table scan: blocks are retrieved one by one, so the cost is $Blocks(R)$ I/O (if clustered)

   Index scan: also $Blocks(R)$ (you still have to read the data), but is quicker if you need to select only the attributes in the index, in which case it is the same as reading the index.

4. Outline the main ideas of Two-Phase Multiway Merge Sort (TPMMS) and Hash Partitioning. What is their costs and conditions/requirements?

   Solution:

   TPMMS: sort parts of relation R of size *MMBuffers* individually in

the first phase and merge all sorted parts in the second phase (cost: $3 * Blocks(R)$, requirement: $Blocks(R) \leq \sqrt{MMBuffers}$).

Hash partitioning: partition a relation R to buckets according to a hash function h using one read buffer and $MMBuffers - 1$ buffers to accumulate blocks for each bucket (cost: $Blocks(R)$, condition: the number of hash values is no more than $MMBuffers - 1$).

5. What are the main advantages and disadvantages of one-pass algorithms, the two variants of two-pass algorithms (sort-based, hash-based), and index-based algorithms?

   Solution:

   One-pass: good, but applies only when at least one argument fits into the memory

   Two-pass: much lighter memory requirements, but higher cost (even in the size of input)

   Two-pass sort-based: always applicable, give sorted results, which can be utilised later

   Two-pass hash-based: usually requires less memory than sort-based (only the smaller argument counts), but relies on a good hash function (equal distribution over buckets), which may not exist

   Index-based: requires indexes, great for selection, often efficient for join

2 Unary operations

1. Assume $Tups(R) = 10,000$, $Blocks(R) = 500$ and $Vals(R.A) = 100$ for a relation R with an attribute A. What is the cost of our algorithm for $\sigma_{A=c}(R)$ if there are no indexes on A? If there exists an index on A (consider all types of indexes we know, but assume that accessing the index is of no cost)?

   Solution: Without an index, we do not know where the values are, so we need to read all blocks. So 500 disk I/Os. If R is not clustered, the cost can be as high as $10,000$.

   With a clustered index, we have $500/100 = 5$ disk I/O. If the index is dense (and hence no clustering), the cost can be up to $10000/100 = 100$ disk I/O if each tuple is in a different block.

2. Let $Tups(R) = 5,000,000$ and $Blocks(R) = 100,000$ for a relation R, and there are 1000 buffers (i.e., blocks in the main memory). What is the cost of $\delta(R)$?

Solution: Since $\sqrt{Blocks(R)} \leq MMBuffers$ but R does not fit in full into the main memory, we can use two pass sort-based duplicate elimination or two-pass hash-based duplicate elimination. Both algorithms have a cost $3 * Blocks(R) = 3 * 100,000 = 300,000$.

3 Blocking Operations

1. For each of the operators generalised bag projection, duplicate elimination, grouping and aggregation, set and bag union, intersection, difference, Cartesian product, and natural join, tell whether the operator is *blocking*, by which we mean that the first output cannot be produced by our evaluation algorithms until all the input has been read (assuming that each the argument relations occupy more than 1 block).

   Solution:

   - Generalised bag projection π^b_L is not blocking: we can start writing the output before all data is read.
   - Duplicate elimination δ is not blocking: we can start producing the output while the data structure of "seen" tuples is populated.
   - Grouping with aggregation γ_L is blocking: we keep one tuple for each group and update this tuple as needed, and so γ_L is blocking since we cannot know if the last tuple to read may update any of the group tuples.
   - Set union $\cup$ is not blocking by the same reasons as duplicate elimination, while the bag version $\cup^b$ by even easier reasons.
   - Intersection $\cap$ is not blocking, because every output tuple does not depend on the rest of the input.
   - Difference $\setminus$ can be blocking or not blocking depending on which relation is read first. For instance, when computing $R \setminus S$ if S is read into $MMBuffers - 1$ buffers and then each block of R is read block by block, then the operation is not blocking. However, if R is read into $MMBuffers - 1$ buffers and then S is read block by block, then the operation is blocking.
   - Natural join $\bowtie$ (including Cartesian product $\times$) is not blocking by the same reason as intersection.

4 One-pass algorithms for binary operations

1. Outline one-pass algorithms for $R \cup^b S$, $R \bowtie_\theta S$, and $R \times S$. Give the cost and space requirements for these algorithms.

Solution:

- Bag union: First retrieve and output all blocks in R one by one, then retrieve and output all blocks in S one by one. Cost: $Blocks(R) + Blocks(S)$, Space: essentially, none.
- Theta-join: keep the smaller relation in memory (assume R), use one buffer to process the bigger one by one (assume S); for each tuple t in S, search for tuples in R to join with, output all join results that satisfy the condition θ. Cost: $Blocks(R) + Blocks(S)$, Space: $\min(Blocks(R), Blocks(S)) + 1 \leq MMBuffers - 1$.
- Cartesian product: the same as theta-join (since it is a special case of).

2. Can nested-loop approach be applied to theta-join? If so, what will be the cost and space requirements?

   Solution: Yes, we can apply the same approach as for natural join without any significant changes. In particular, we can extend the condition in the nested join to include the condition. This will have the same cost and space requirements as a natural join (quadratic cost, but essentially no space requirements)

5 Sort-based algorithms

1. Let $Blocks(R) = Blocks(S) = 10,000$ for relations R and S, and $MMBuffers = 1000$. What is the cost of our algorithms for set union and two-pass sort-based natural join of R and S?

   Solution:

   (a) Set union (which boils down to duplicate elimination): $3*(Blocks(R)+Blocks(S)) = 3 * (10000 + 10000) = 60000$.

   (b) Natural join requires to sort by TPMMS and store result on disk (4 I/O's), and then to read again to join; so $5 * (Blocks(R) + Blocks(S)) = 5 * (10000 + 10000) = 100000$.

2. How many main-memory blocks do we need to use for a two-pass sort-based algorithm for relations of 10,000 blocks (per relation) if the operation is duplicate elimination, grouping with aggregation, set union, natural join?

   Solution:

   - Duplicate elimination of R: at least $\sqrt{Blocks(R)} = \sqrt{10,000} = 100$

- Grouping and aggregation: at least $\sqrt{10,000} = 100$
- Set union: at least $\sqrt{10,000 + 10,000} \approx 141$
- Natural join of R and S: at least $\sqrt{Max(Blocks(R), Blocks(S))} = \sqrt{10,000} = 100$

3. Design an algorithm that optimises the two-pass sort-based natural join $R \bowtie S$ that combines the second phase of the sorts with the join itself.

   Solution:

   a. Sort parts of R and S of size *MMBuffers* individually on the join attributes (same as the first phase of TPMMS)

   b. Bring the first block of each part of each relation into a buffer (assuming there are no more than *MMBuffers* parts in the both relations)

   c. Repeat until all tuples of S have been examined
      - Find the next tuple of S with the least join value (in S buffers)
      - For each R buffer, proceed through the R tuples one by one removing them, until the join value is at least the value of S; if the join values are equal, output the join; if the R buffer becomes empty, load a new block from the same part
      - Remove the S tuple; if the S buffer becomes empty, load a new block from the same part

4. (*) For the two-pass-sort-based natural join algorithm, we have a requirement that there should be at most *MMBuffers*/2 blocks with the same join value in each relation. Explain what should be changed in the algorithm if this requirement does not hold and how it influences the cost. Consider two cases: when each join value appears in at most *MMBuffers* blocks (in both relations together) or not. (Note: you do not need to describe the new algorithms in full, just discuss its essential differences)

   Solution:

   In the first case, we may decide how many buffers we allocate to each relation on the fly and fill in buffers with R or S blocks in a way so that all tuples in both relations with the current join value is fully in buffers. In that case, nothing changes in cost.

   In the second case, we have to essentially do nested-loop join for such 'popular' join values, and the cost increases to quadratic (for this particular values). In the worst case, when all join values are the same

(e.g., in the case of Cartesian product, where the only 'join value' is the empty tuple) we have to output all combinations of all tuples, for which we have to have the cost of the nested loop.

6 Hash-based algorithms

1. Come back to slide 41 of the slide set Lecture_13to15 and analise two-pass hybrid hash-based natural join algorithm. Try to understand why the improvement ensures the gain in the cost. If not sure, look at Section 15.5.6 in Garsia-Molina book (see slide 4 in the slide set for the complete reference).

2. The hybrid-hash-join idea of storing one bucket in main memory can also be applied to other operations. Show how to save the cost of storing and reading one bucket from each relation when implementing a two-pass hash-based algorithm for duplicate elimination, grouping and aggregation, and intersection.

   Solution:

   - For duplicate elimination, we assume that the number of buckets k is such that the entire selected bucket plus one block of each $k-1$ other buckets fit into *MMBlocks*. Then we keep all the distinct tuples of the first bucket in the main memory and avoid having to store and read blocks for one bucket out of k. Since the bucket size will be approximately *MMBlocks*, we can save about 2*MMBlocks* disk I/Os.
   - Grouping and aggregation is similar to duplicate elimination.
   - For intersection, we choose the smaller relation S and assume that the number of buckets k is such that the entire selected bucket plus one block of each $k-1$ buckets fits into *MMBlocks*. If a tuple of the other relation R hashes into this in-memory bucket, we perform the union operation right away. If the tuple of R does not hash into the in-memory bucket, we send it to the main memory block associated with that bucket.

3. (*) Let $Blocks(R) = Blocks(S) = 10,000$ for relations R and S, and $MMBuffers = 1000$. What is the number of disk I/O 's required for the two-pass hybrid hash join of R and S in the case when the number of hash values (i.e., buckets) is 11 (and the tuples of the relations are evenly distributed over the buckets)?

   Solution: Since the number of buckets is 11, the expected size of the in-memory bucket is 909 blocks, which leaves enough memory for the 10 blocks for the remaining buckets. The number of disk

I/O's we use for R for the first pass is 10,000 to read all of R, and $10,000 - 909 = 9,091$ to write ten buckets to disk. For S, we need the same. On the second pass, we read all buckets written to disk, or $9,091 + 9,091 = 18182$ I/O's. The total number of I/O's would be $10,000 + 9,091 + 10,000 + 9,091 + 18,182 = 56,364$.

7 Index based algorithms

1. Suppose there is an index on attribute A of relation R and another relation S. Describe how this index could be used to improve the execution of $\delta(R)$ and $R \cap S$. Under what circumstances would the index-based algorithm be more efficient than sort- or hash-based algorithms?

   Solution:

   - For duplicate elimination, we could use the index for A to process R in batches, where each batch is identified by a given index search key. If there is only one tuple for a given index key, output that tuple. Otherwise, perform duplicate elimination (e.g., using hash or sort) for the set of tuples. This method is efficient when $Val(R, A)$ is large.
   - For intersection, we can read S block by block and, for each tuple t of S, use index for A to retrieve all tuples of R that match t on A. If t is among those tuples, output t. This method is efficient when S is small and R is large.

2. Let $Blocks(R) = 10,000$ and $Tuples(R) = 500,000$ for a relation R. Let there be an index on attribute A of R and let $Val(R.A) = k$ for some number k. Give the cost of $\sigma_{A=0}(R)$ as a function of k, when the index is clustering, secondary, or not used at all (where you may neglect disk I/O's needed to access the index itself).

   Solution: If the index is clustering (dense or sparse), we need at most $\lceil 10,000/k \rceil + 1$ disk I/Os. If it is secondary, we need up to $\lceil 500,000/k \rceil$ I/Os (with the maximum when all tuples are in different blocks); however, we do not need to read more than 10,000 blocks, and so a better bound is $\max(500,000/k, 10,000)$. Without using the index we have to get all blocks: $10,000$ disk I/Os.