# Class Assignment: Query Processing: Intermediate Submission

[A]

Look at all the queries beforehand as a collective batch. Process them accordingly. For example, change the order of execution, cache or save intermediate results which can be reused, build temporary indices on certain fields within relations and on fields in relations created as a result of an intermediate join. We can pre-compute the cost of joins as well and pre-compute a few joins needed for later queries. Some common pre-processing for joins can also be done. Other statistics can also be maintained. The optimization would collectively involve: reordering, saving some essential statistics, saving temporary indices after figuring the most optimal index patterns, saving necessary intermediate outputs during the actual execution.

[B]

Set 1:

1. SELECT SSN FROM EMPLOYEE
2. SELECT FNAME, LNAME FROM EMPLOYEE

Set 2:

1. SELECT SSN FROM EMPLOYEE
2. SELECT FNAME, LNAME FROM EMPLOYEE
3. SELECT FNAME FROM EMPLOYEE
4. SELECT LNAME FROM EMPLOYEE
5. SELECT FNAME, LNAME FROM EMPLOYEE WHERE DNO=4

Set 3:

1. SELECT SSN FROM EMPLOYEE
2. SELECT FNAME, LNAME FROM EMPLOYEE
3. SELECT FNAME FROM EMPLOYEE
4. SELECT LNAME FROM EMPLOYEE
5. SELECT FNAME, LNAME FROM EMPLOYEE WHERE DNO=4
6. SELECT FNAME, LNAME FROM EMPLOYEE WHERE DNO=1
7. SELECT DNAME FROM DEPARTMENT
8. SELECT PNAME FROM PROJECT WHERE DNUM = 5
9. SELECT ESSN FROM DEPENDENT WHERE SEX=”M”
10. SELECT ESSN FROM WORKS_ON WHERE HOURS>25.0

[C]

In the proposed case, we would want to execute the intersection of the given queries not more than once.

In general, for queries that act on the same set of rows (might be different projections), these three ideas could be used:

1. Create an index on the Key or the unique value in those rows.
2. Create indices on the most common fields used for access, e.g: if the selection is based on HOURS, have an index on hours, the index providing the relation key which can be used for the row selection & projection (number of indices and depth depending on the space available)
3. Two way indices on key and commonly used values in the WHERE clause

    - If the rows are created as a result of a join, pre-compute the join, build index on the join intermediate.

[D]

Find out which particular field is being utilized the most for accessing rows. Let's say, if all queries are on 'Bdate', then it would be ideal to have the index built on that. Space is the biggest bottleneck. Suppose, we require indices on 10 different fields. Depending on how much memory we have, we'll need to compute the number of levels required for each index. It would be ideal to have multi-level indices here. If we had unlimited space, we could have B+ trees on everything: fields in a table, a list of possible joins and crosses. Since we don't have endless memory, we need to find out which fields (key, or non-key or composite attribute) should be used for building indices, and how many levels they should have. This is a pure strategy design and optimization problem. We need to compute the most optimal set of fields for the purpose of indexing and a max_depth for each of the proposed indices.

[E]

Best case: All the queries are exactly the same, on a key field of a particular relation, no joins involved.

Worst Case: Covers all relations, involves large joins to estimate correct answers - queries are on non-key fields (need to traverse the entire relation to make sure we are considering everything). And the queries involve different relations with nothing particular in common: i.e pre-computing or intermediate result saving does not really help.

[F]

In query processing, having good optimization schemes and execution strategies is of utmost importance. Query ordering for execution, building indices on intermediate results is also important for processing batch queries.