# Weekly exercises IN3020&4020 (2023)
# Week 6: Query Compilation and Optimisation

## 1 Review

1. What are the 4 main steps that a query processor must perform (as outlined in the lectures), and what is their input and output?

   **Solution:**

   1. Parsing: transforms an input query (as a string) to a parse tree, which organise tokens into groups and subgroups
   2. Compiling: transforms a parse tree to an initial (logical) query plan
   3. Optimisation: transforms the initial plan to an optimised plan
   4. Evaluation: evaluates the plan over the tables

2. What is a logical query plan?

   **Solution:** It is a relational algebra expression (in the context of query processing).

3. What is the ideal goal of query optimisation?

   **Solution:** To find another query plan that

   - Gives the same answer as the initial plan over the tables
   - Can be evaluated as quick as possible
   - Is quick to find

4. What is the pragmatic goal of query optimisation, and what is the key ingredients for this approach?

   **Solution:** The goal is to find another query plan that

   - is equivalent to the original plan (as a relational algebra expression)
   - is estimated to be quicker than other candidates (for the tables)

- is quick to find

The ingredients are a good but quick cost function, which reflects evaluation time, and a good heuristic that quickly constructs relevant candidate plans.

5. What is our cost function and what is its main properties?

   **Solution:** Our cost is the estimate of the sum of sizes of the intermediate results of the plan. It is likely to be proportional to the number of disk I/O (which is the most expensive for evaluation). It is compositional: we can compute the cost of a plan from costs of the subplans. It is quick to compute.

## 2 Parsing

Consider this grammar for a subset of SQL:

```
<Query> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>
<SelList> ::= <Attribute>, <SelList>
<SelList> ::= <Attribute>
<FromList> ::= <Relation>, <FromList>
<FromList> ::= <Relation>
<Condition> ::= <Condition> AND <Condition>
<Condition> ::= <Attribute> IN ( <Query> )
<Condition> ::= <Attribute> = <Attribute>
<Condition> ::= <Attribute> LIKE <Pattern>
```

1. Extend (or change) the rules from `<Query>` to include the following:
   (a) Sort the output using the `ORDER BY` a list of attributes
   (b) A query with no `WHERE`-clause

   **Solution:**

   (a) Can change the rules for `<Query>` to:

   ```
   <Query> := SELECT <SelList> FROM <FromList>
              WHERE <Condition> <OptOrdering>
   <OptOrdering> := emptyString
   <OptOrdering> := ORDER BY <OrderList>
   <OrderList> := <Attribute>, <OrderList>
   <OrderList> := <Attribute>
   ```

   (b) Can change rules for `<Query>` to:

   ```
   <Query> := SELECT <SelList> FROM <FromList> <OptCondition>
   ```

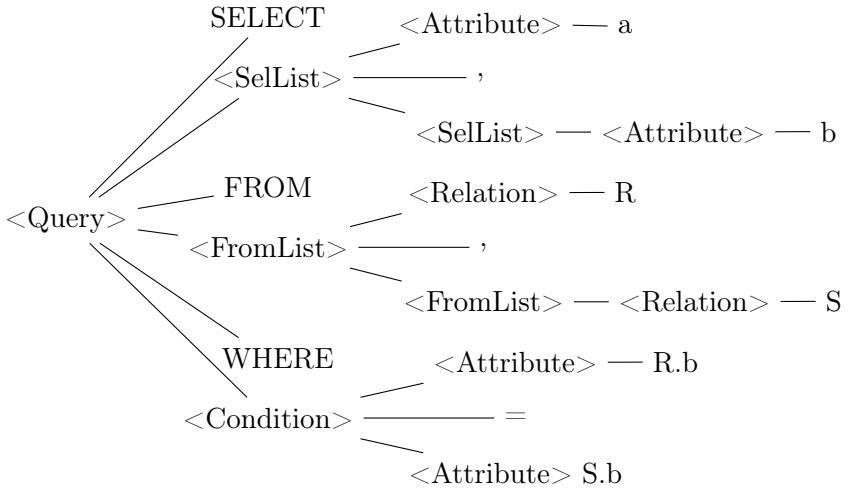
```
<OptCondition> := emptystring
<OptCondition> := WHERE <Condition>
```

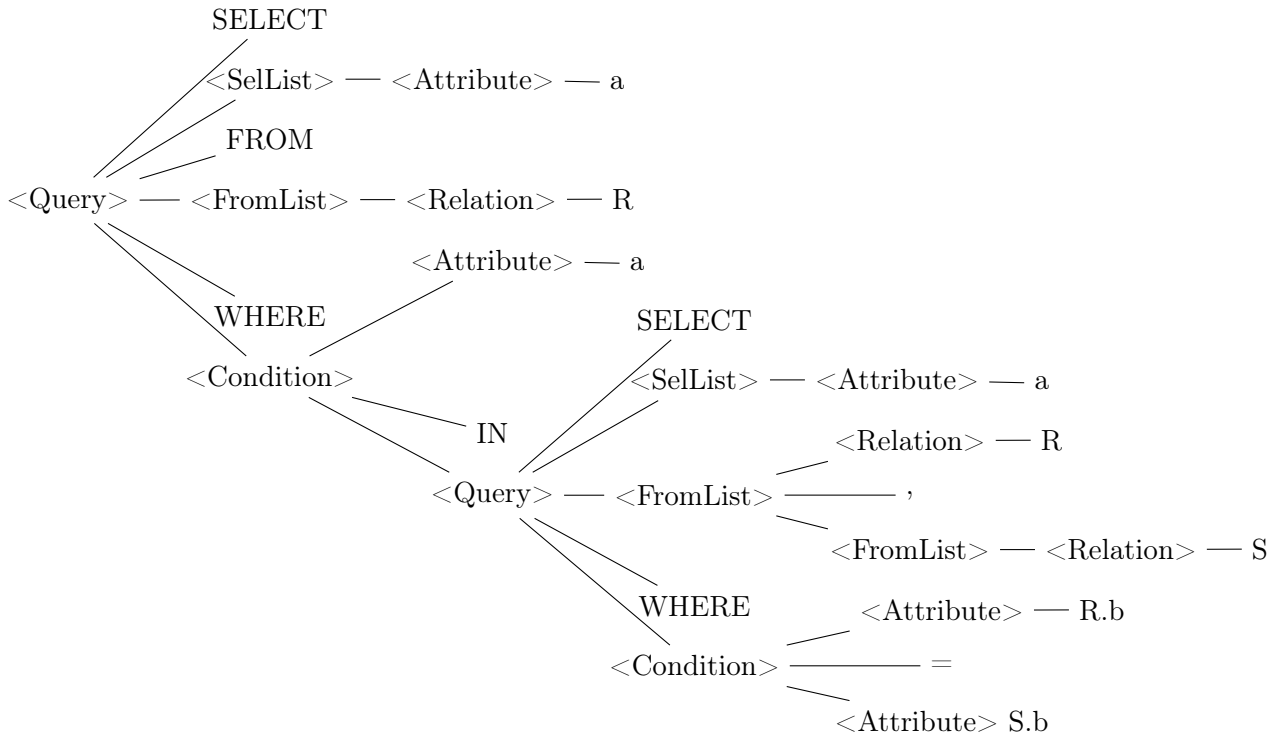
2. Using the subset of SQL grammar above, construct parse trees for the following queries over relations $R[a, b]$ and $S[b, c]$:

   (a) `SELECT a,c FROM R,S WHERE R.b = S.b;`

   (b)
   ```
   SELECT a FROM R WHERE a IN
     (SELECT a FROM R, S WHERE R.b = S.b);
   ```

**Solution:**






## 3 Compilation

Consider the two schemas $R[A, B, C]$, $S[A, D, F]$. Compile the following SQL queries into initial query plans (i.e., Relational Algebra expressions):

1. `SELECT A FROM R`

   **Solution:** $\pi^b_A(R)$

2. `SELECT DISTINCT A FROM R`

   **Solution:** $\delta(\pi^b_A(R))$

3. `SELECT * FROM R NATURAL JOIN S`

   **Solution:** $R \bowtie S$

4. `SELECT * FROM R, S`

   **Solution:** $\rho_{A \to R.A}(R) \times \rho_{A \to S.A}(S)$

5. `SELECT * FROM R INNER JOIN S ON R.B = S.D`

   **Solution:** $\rho_{A \to R.A}(R) \bowtie_{B=D} \rho_{A \to S.A}(S)$ (note the mismatch with the names of attributes; this is not essential and other variations of relational algebra may behave slightly differently with respect to renaming)

6.* `SELECT * FROM R LEFT OUTER JOIN S ON R.B = S.D`

**Solution:** $\rho_{A \to R.A}(R) \rtimes \pi^b_{A \to S.A,\, D \to B,\, D,\, F}(S)$ (Note the second copy of $D$ of $R$; as the result, $B$ and $D$ may have different values in the result despite that they are the join attributes, which is the semantics of `LEFT OUTER JOIN` operator in SQL)

7. `SELECT SUM(B) FROM R GROUP BY A`

   **Solution:** $\pi^b_{sumB}(\gamma_{A,\, \mathsf{SUM}(B) \to sumB}(R))$

8. `SELECT R.C FROM R WHERE R.A IN (SELECT S.A FROM S WHERE S.D > 0)`

   **Solution:** $\pi^b_C(R \ltimes (\sigma_{D>0}(S)))$

9. `SELECT R.C FROM R WHERE NOT EXISTS (SELECT S.A FROM S WHERE R.A = S.A)`

   **Solution:** $\pi^b_C(R \triangleright S)$

# 4 Optimization

## 4.1

When it is possible to push a selection to both arguments of a binary operator, we need to decide whether or not to do so. How would the existence of indexes on one of the arguments affect our choice? Consider for instance an expression $\sigma_C(R \cap S)$, where there is an index on $S$.

**Solution:** If the index on $S$ helps us find tuples that satisfy the condition $C$, then surely we should push the selection to $S$. Let the result of the selection on $S$ be a relation $S'$, which is a subset of $S$. Assuming there is no index on $R$ that helps with condition $C$, then if we push the selection to $R$ as well, we still have to intersect that result with $S'$. However, if we do not push the selection to $R$, then all that remains is intersecting $R$ with $S'$. Performing either the selection or the intersection on $R$ requires that we read all of $R$, so we are better off performing just the intersection rather than the selection plus a (smaller) intersection.

## 4.2

Consider the statistics for the four relations $W$, $X$, $Y$ and $Z$ in Table 1, additionally assuming that each attribute takes 5 bytes. Estimate the size (in bytes) of the relations that are the results of the following expressions:

1. $W \bowtie X \bowtie Y \bowtie Z$

Table 1: Statistics for $W, X, Y$ and $Z$

| $W\,[A, B]$ | $X\,[B, C]$ | $Y\,[C, D]$ | $Z\,[D, E]$ |
|---|---|---|---|
| $Tups(W) = 100$ | $Tups(X) = 200$ | $Tups(Y) = 300$ | $Tups(Z) = 400$ |
| $Vals(W.A) = 20$ | $Vals(X.B) = 50$ | $Vals(Y.C) = 50$ | $Vals(Z.D) = 40$ |
| $Vals(W.B) = 60$ | $Vals(X.C) = 100$ | $Vals(Y.D) = 50$ | $Vals(Z.E) = 100$ |

2. $\sigma_{A=10}(W)$
3. $W \times Y$
4. $\sigma_{D>10}(Z)$
5. $\sigma_{A=1\,\&\,B=2}(W)$
6. $\sigma_{C=20}(Y) \bowtie Z$
7. $\sigma_{C=20}(Y \bowtie Z)$

**Solution:**

1. Start by taking the product of the sizes of the relations: $100 * 200 * 300 * 400 = 2,400,000,000$. Since attribute $B$ appears in two relations, divide by the larger of $Vals(W.B)$ and $Vals(X.B)$, which is 60. Similarly, for $C$, divide by 100, and, for $D$, divide by 50. The result is 8000 tuples with 5 attributes each, which ends up with the final estimation $8000 * (5 * 5) = 20000$ bytes.
2. The estimate is
$$(Tups(W)/Vals(W.A)) * TupSize(W) = (100/20) * (5 + 5) = 50.$$
3. We have $Tups(W) * Tups(Y) * (TupSize(W) + TupSize(Y)) = 100 * 300 * 20 = 600,000$ bytes.
4. We have $Size(Z)/3 \approx 1333$.
5. $(Tups(W)/(Vals(W.A) * Vals(W.B))) * TupSize(W) = (100/(20 * 60)) * 10 = 10/12 \approx 1$ byte.
6. We have $Tups(\sigma_{C=20}(Y)) = Tups(Y)/Vals(Y.C)) = 300/50 = 6$ and so $Vals(\sigma_{C=20}(Y.D)) \leq 6$. Therefore, $Size(\sigma_{C=20}(Y) \bowtie Z) = ((6 * 400)/\max(40, 6)) * (5 + 5 + 5) = 900$.
7. We have
$$Tups(Y \bowtie Z) = (Tups(Y) * Tups(Z))/\max(Vals(Y.D), Vals(Z.D))) = (300 * 400)/50 = 2400$$

and $Vals((Y \bowtie Z).C) = Vals(Y.C) = 50$. Therefore, $Size(\sigma_{C=20}(Y \bowtie Z)) = (Tups(Y \bowtie Z)/Vals((Y \bowtie Z).C)) * TupSize(Y \bowtie Z) = (2400/50) * 15 = 720$.

### 4.3

Observe that the last two expressions in Question 4.2 are equivalent—that is, can be seen as two different logical query plans of the same SQL query. Compute the costs of these two plans (under our definition of cost); compare the relationship between the costs with the relationship between the sizes of these relations, which you computed in Question 4.2. Justify our definition of the cost function on the base of this comparison.

**Solution:** Our cost is the sum of estimated sizes of the intermediate subexpressions of the plan. There is only one intermediate result for both plans: $Size(\sigma_{C=20}(Y)) = Tups(\sigma_{C=20}(Y)) * TupSize(\sigma_{C=20}(Y)) = 6 * 10 = 60$ and $Size(Y \bowtie Z) = Tups(Y \bowtie Z) * TupSize(Y \bowtie Z) = 2400 * 15 = 36000$. So, the cost of $\sigma_{C=20}(Y) \bowtie Z$ is 60 while the cost of $\sigma_{C=20}(Y \bowtie Z)$ is 36000, and based on this we expect $\sigma_{C=20}(Y) \bowtie Z$ to be a better (quicker) plan than $\sigma_{C=20}(Y \bowtie Z)$. Note that the size of the first plan is expected to be greater than of the second. However, we know that, since the plans are equivalent, the real sizes of their results will be the same. This is why our cost does not take the estimations of the final results into account. In this example, pushing selection through the join seems to be a good idea even intuitively.

### 4.4

How would you estimate the size of a semijoin, assuming, for simplicity, that it is applied to set relations?

**Solution:** A (direct) idea is to use the definition of (set) semijoin ($R \ltimes S = \pi_{RAtts}(R \bowtie S)$) and calculate the estimation using our usual technique for this expression. However, we can use what we know about semijoin, in particular, that the number of tuples in the result of the semijoin can not be larger than the number of tuples in the first argument. Using this information, we can estimate the size of $R \ltimes S$ as follows:

$$TupSize(R) * Tups(R) * \min\left(1, \frac{Tups(S)}{Tups(R)}\right)$$

### 4.5 (*)

Consider relations $R[A, B]$ and $S[B, C]$ with $Vals(R.B) = Vals(S.B) = 20$ and (combined) histograms for the values of $B$ as in Table 2. Estimate the number of tuples in the join $R \bowtie S$ using an assumption that the values of $B$ are evenly distributed in the interval (in the histogram).

Table 2: Frequency of most common values

| Values: | 1 | 2 | 3 | 4 | 5 | 6...20 |
|---|---|---|---|---|---|---|
| $R.B$: | 5 | 6 | 4 | 6 | 0 | 30 |
| $S.B$: | 10 | 8 | 5 | 0 | 7 | 30 |

**Solution:** We have 5 values with exact number of tuples and 15 values in the interval for each relation. For each of the 5 values, we can calculate the exact number of tuples: $5 * 10 + 6 * 8 + 4 * 5 + 6 * 0 + 0 * 7 = 118$ tuples. Under the assumption, we have $30/15 = 2$ tuples per each of the 15 values in each relation and so $2 * 2 * 15 = 60$ tuples. In total we get an estimate of $118 + 60 = 178$ tuples.