QUERY EXECUTION

We want to translate the SQL query into an algebraic query
to define the order of operations.
Then we do a logical transformation,
typically push the selection operation towards the leaves,
so that the join is computed later.
Finally, you generate a feasible access plan,
choosing the correct algorithm,
meaning that we annotate each node with
the algo that we are going to use.

The order of the join is the most important decision
in terms of speed of the query.

INITIAL LOGICAL PLAN $\xrightarrow[\text{TRANSFORMATION}]{\text{LOGICAL}}$ NEW LOGICAL PLAN $\longrightarrow$ ACCESS (OR PHYSICAL) PLAN

Physical operators

Each physical operator is a specific implementation of a logical operator.

- $R \to \text{TableScan}(R)$
- $\pi_X(E) \to \text{Project}(O_E, X)$ ⇝ argument O_E and parameter X
- $E_1 \bowtie_{cond} E_2$ → NestedLoop $(O_1, O_2, cond)$
  → IndexNestedLoop $(O_1, O_2, cond)$
  → MergeJoin $(O_1, O_2, cond)$
- ...

How do you implement a tree of operators?

One way is "one operator at a time"
or "materialization of the result of each operator.
Very simple to implement, but requires a lot of read/write.
Not how a DBMS work.

What is used is the iterator open/next/close,
called "iterator style" or "cursor interface".

For every node we use the iterator, for example

Filter(Province = PI) ⇝ it just calls next on the TableScan,
which remember the last cursor position
|
TableScan (Student)

The filter doesn't need to keep everything in memory.
The TableScan just stores the position of the cursor.

We write the access plan like a tree,
where the argument of an operator is its child.

Children of operators

Observe that each operator has a fixed number of children.

For example

- TableScan is always a leaf.
- IndexFilter (R, Idx, ψ) ⇝ it gets out of the index idx all the values that satisfy the condition ψ

⇝ IndexFilter (Students, IdAgeStud, age > 25)

open the index IdAgeStud, retrieve RIDS with age > 25, reads corresponding students from the table Students

⇝ this is ALWAYS a leaf, because it always reads from disk, cannot be combined a priori

Project and Filter

They are the simplest operators.

Logical Plan		Access Plan
π^b_A	$\rightarrow$	Project({A})
σ_ψ	$\rightarrow$	Filter(O, ψ)
R		TableScan(R)

π^b_A because SQL produces multisets (no duplicate elimination)

this is a basic
SELECT
FROM
WHERE

Distinct elimination

If we want to do duplicate elimination, it is more complicated

π_A	$\longrightarrow$	Distinct Sort({A}) Project({A})
σ_ψ	$\longrightarrow$	Filter(ψ)
R	$\longrightarrow$	TableScan(R)

$\rightsquigarrow$ This is needed because whenever you ask 'next' it remembers only the last element, so distinct can compare only with the last element

What is the cost of distinct? $\emptyset$ because it happens in the main memory.

And Sort? It is $2 \cdot N_{page}$, because it needs the disk,
so we try to reduce the use of sort,
and avoid it if data is already sorted

And Project? ϕ

And filter? ϕ

And TableScan? N_{pag}

Selection and Index Filter

SELECT *
FROM R
WHERE A BETWEEN 50 AND 100

with Idx an index on A

$$\pi^{b}_{*} \rightarrow \text{Project}(*)$$

$$\left.\begin{array}{c}\sigma_{\psi}\\ |\\ R\end{array}\right\} \rightarrow \text{IndexFilter}(R, Idx, \psi)$$

'IndexOnlyFilter' combines access to index with projection.
It just gives the values that it finds in the index,
without going to the data. Much more efficient.
Go to the index and project to the same attributes of the index.
It does not read the table.

What is the difference between Filter (O, ψ) and IndexFilter (R, Idx, ψ)?
The first filter the records on the given argument O,
while the second access the records of R
through the index, with NO argument.

all depends on the selectivity factor

About the cost,

filter (TableScan(R), ψ) $\rightsquigarrow N_{pag}(R)$ ✓

IndexFilter (R, Idx, ψ) $\rightsquigarrow CI + CD = CI + sf * NRec(R)$

In addition, IndexFilter (R, idx, ψ) gives dat already sort by A,
then it might be not need to sort again.

Group By

The canonical way is by sorting and then break when the attribute changes.

SELECT A, COUNT(*)
FROM R
WHERE
GROUP BY

$\sigma_{count(*)>1}$ → Filter(count(*)>1)

|

${}_{\{A\}}\gamma_{\{COUNT(*)\}}$ → GroupBy({A}, count(*)) / Sort

|

σ_{ψ} → Filter(ψ)

|

R → TableScan(R)

GROUPED means that when there is a break I will never get that value again.

SORTING implies GROUPED, it is stronger.

If you are not sure that data is grouped, then you need to sort.

But since grouped data is enough, it might not be required to sort.

Sort

The sort operator is $\tau_{\{A_i\}}$
and its cost is $2 * N_{pag}$

Nested Loop (JOIN)

This is the simplest and most expensive algo.

Algo:

for each record r in R do ← outer relation
  for each record in S do ← inner relation
    if $r_i = s_j$ then add $\langle r, s \rangle$ to the result

The cost is $NPag(R) + NRec(R) * NPag(S)$

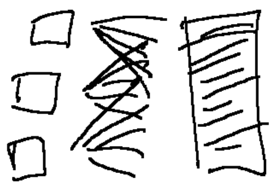
↑ read left side — ↑ for every record on the left side (outer rel.) — ↑ we scan the entire inner relation

So the cost is quadratic, and you will never use it.

Page Nested Loop (JOIN)

Instead of scanning the entire relation once for every record of the outer relation, you read an entire page of of the outer relation and join the entire page.

More difficult to implement and still quadratic, but less expensive since $N_{page} < N_{records}$.



The cost is $N_{pag}(R) + NPag(R) * N_{pag}(S)$

Index Nested Loop

For every tuple of R, read the tuple R.A,
then use the index of S.B.

IndexNested Loop requires two arguments,
on the left, something that allows "next", like TableScan,
on the right, IndexFilter (S, Idx, S.B = R.A)

observe that here S.B
opens the index, is a constant,
while R.A is the attribute

→ remember that for every
operator we do
open | next | close

→ the condition i

This is the only operator where
data flows also from left to right,
and not only bottom-up.
It is the 'open' operator that carries
this information.

77' and before

Few minutes missing here

MergeJoin (JOIN 4) (also SortMerge)

If both tables have many records.
The cost is just that of sort,
that is $2 * Npag$, linear in the number of pages.

- MergeJoin (R.A = S.B)
  - Sort (R.A)
    - TableScan (R)
  - Sort (S.B)
    - TableScan (S)

! Before MJ, be sure that the input is sorted exactly on the join attribute.

Should we use IndexNestedLoop or Mergejoin?
The cost is respectively $Npag(R) + K * NRec(R)$ and $2 * Npag$.

So if S has $NRec = 100.000$ and $NPag = 1.000$
and R has $NRec = 3$ and $NPag = 1$.
Then INL is very convenient, because the cost is $1 + K * 3$.
While MJ's cost is $2 * 1.000$.

But if R is as big as S, like
R with $NRec = 100.000$ and $NPag = 1.000$,
Then INL will be around 200.000
and MJ is $2 * 1.000 = 2.000$

In the transactional application, using few records, use INL,
while analytical applications use MJ.

Of course, to use INL, we need an INL.
But typically every DBMS put an index on every key and foreign key.

Operators of JOIN (review)

1. NestedLoop (O_E, O_I, ψ)
2. PageNestedLoop (O_E, O_I, ψ)

NOT USED

3. IndexNestedLoop (O_E, O_I, ψ)
4. SortMerge (O_E, O_I, ψ)

USED

Query Optimization

Typically means moving the restriction σ before the join, that is expensive. This is the logical optimization.

Then we have the physical plan generation of the plan with the optimal cost.

Actually we just avoid terrible plans.