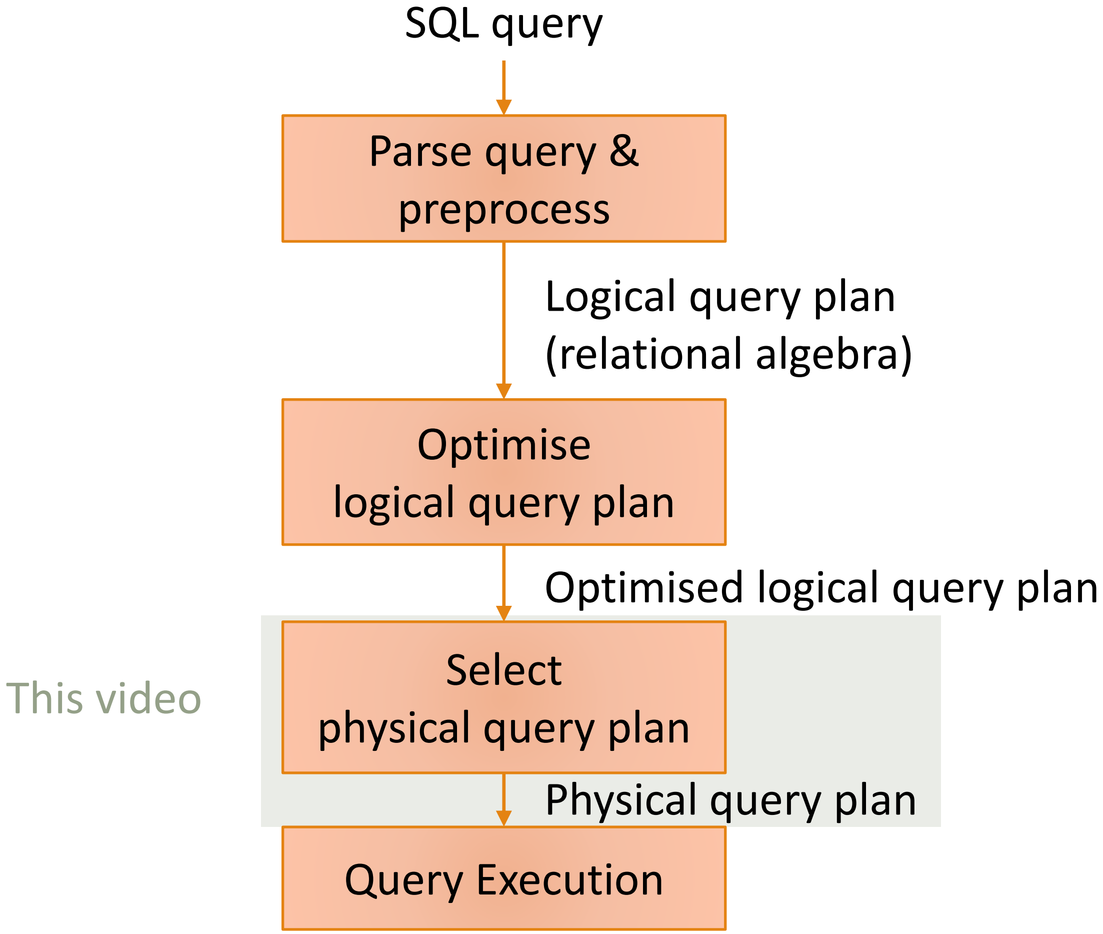
Physical query plans

Overview of this video

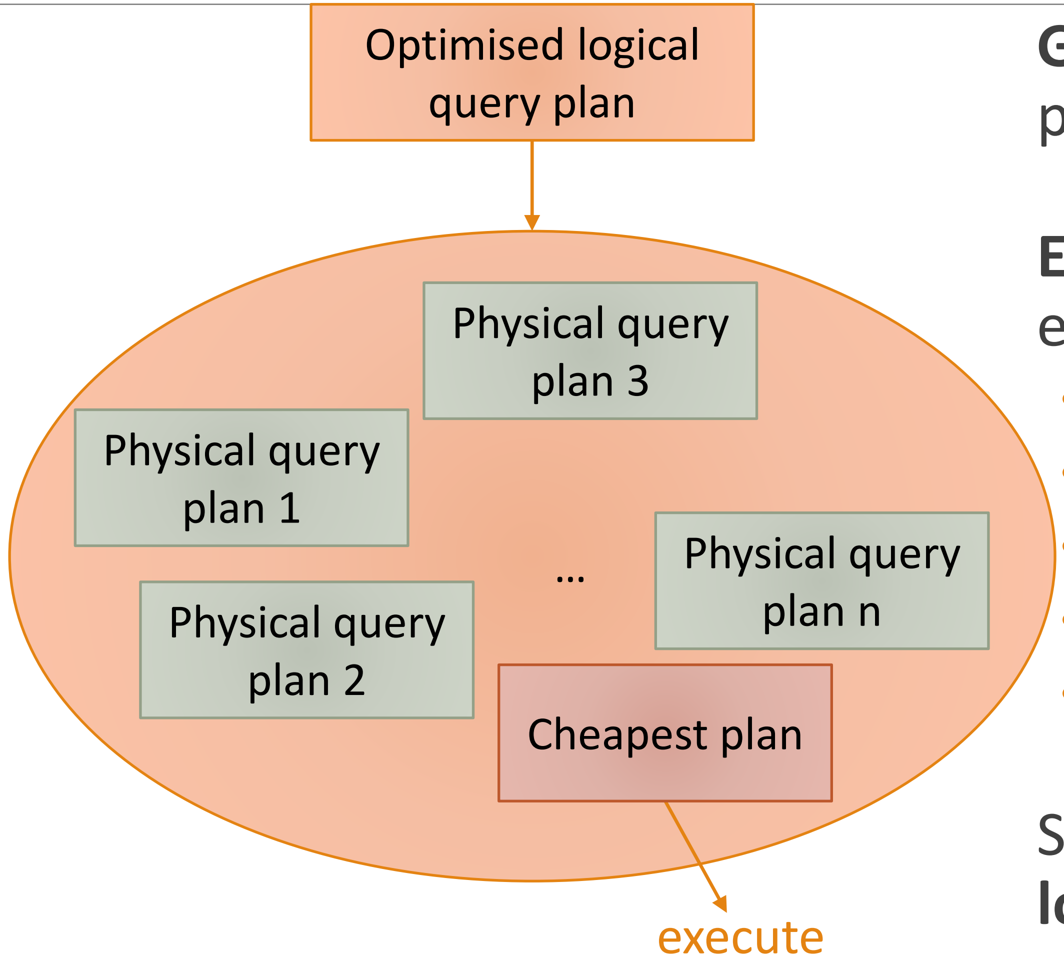


What is the purpose of a physical query plan?

A **physical query plan** adds information required to execute the optimised query plan

- Which **algorithm** to use for execution of operators?
  - Naïve selection or selection with an index?
  - Nested Block Join or Sort Join or Hash Join etc.?
- How to **pass information** from one operator to the other?
  - Write to disk, keep in memory, pipelining operators, etc.?
- Good **order** for computing joins, unions, etc.?
- Additional operations such as sorting

From Logical Plans To Physical Plans



Generate many different physical query plans

Estimate cost of execution for each plan

- Time
- Disk accesses
- Memory
- Communication
- …

Select physical plan with **lowest cost estimate**

Estimating the Cost of Execution

Here: **number of disk access operations**

Number of disk accesses influenced by many factors:

- Selection of algorithms for the individual operators
- Method for passing information
- **Size of intermediate results**

One of the most critical factors

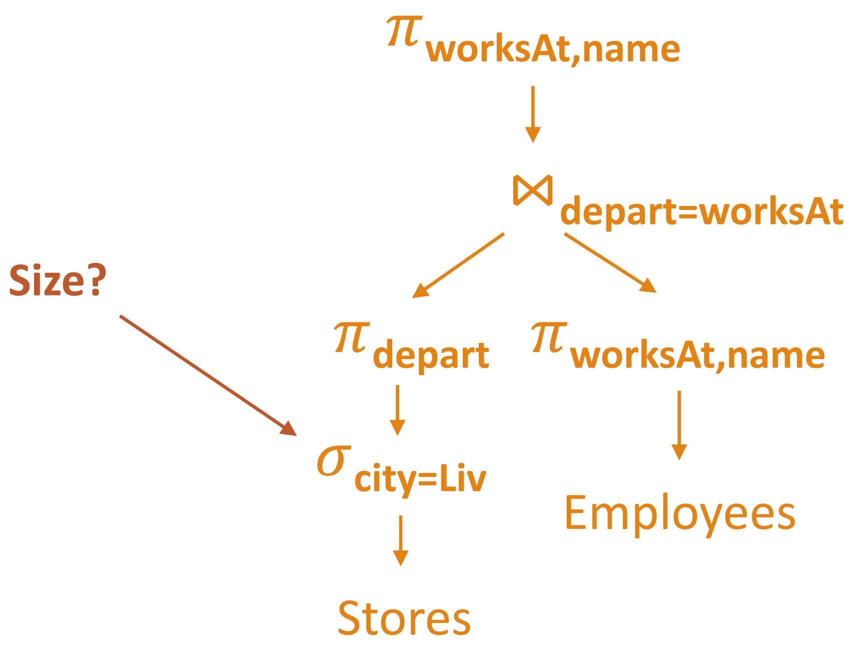
Estimated from parameters of the database

- Important paramters:
  - **Size of relations**
  - **Number of distinct items per attribute per relation**
- Computed exactly from the database or are estimated (“statistics gathering”)

Estimating Intermediate Result Sizes

One of the most challenging tasks of a DBMS

- Difficult even for nodes close to the leaves
- Cannot afford executing the query
- Rely on statistics gathered from data



Many different approaches

- Some easier than others
- With join size estimation, we enter active research…

Estimating the Size of a Selection

Estimate for the size of $\sigma_{A=a}(R)$:

Recall: |R| = number of tuples in R

$$\frac{|R|}{\text{number of distinct values in column } A \text{ of relation } R}$$

Estimate for the # of blocks required to store $\sigma_{A=a}(R)$:

$$\frac{\text{number of blocks for } R}{\text{number of distinct values in column } A \text{ of relation } R}$$

Good if values in column **A** of **R** occur equally often, but can be bad

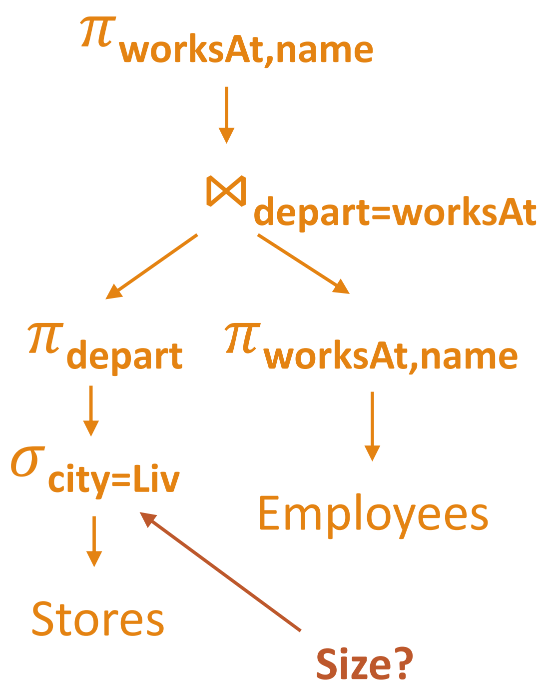
Example

Assume:

- Stores contains **80 tuples**, stored in **20 blocks**
- There are exactly **4 distinct values** for the city attribute of Stores

Estimate for size of $\sigma_{city=Liv}$(Stores):
80/4 = 20 tuples

Estimate for number of blocks that are required to store $\sigma_{city=Liv}$(Stores): 20/4 = 5 blocks



Joins

Assume A is the only common attribute.

How to estimate **R** ⋈ **S**?

Simple estimate based on size of **R** & **S** and number of distinct values in common attributes

$$\frac{|R| \times |S|}{\text{max. number of distinct values for } A \text{ in } R \text{ or } S}$$

As for selection, based on assumptions that might not always lead to good estimates

More sophisticated methods:

- Still a topic of active research
- See, e.g., SIGMOD/PODS/VLDB conferences

Other Issues

How to generate physical query plans?

- Explore all?
- More sensible approaches: top-down/bottom-up

Selection of a **suitable algorithm** for each operator

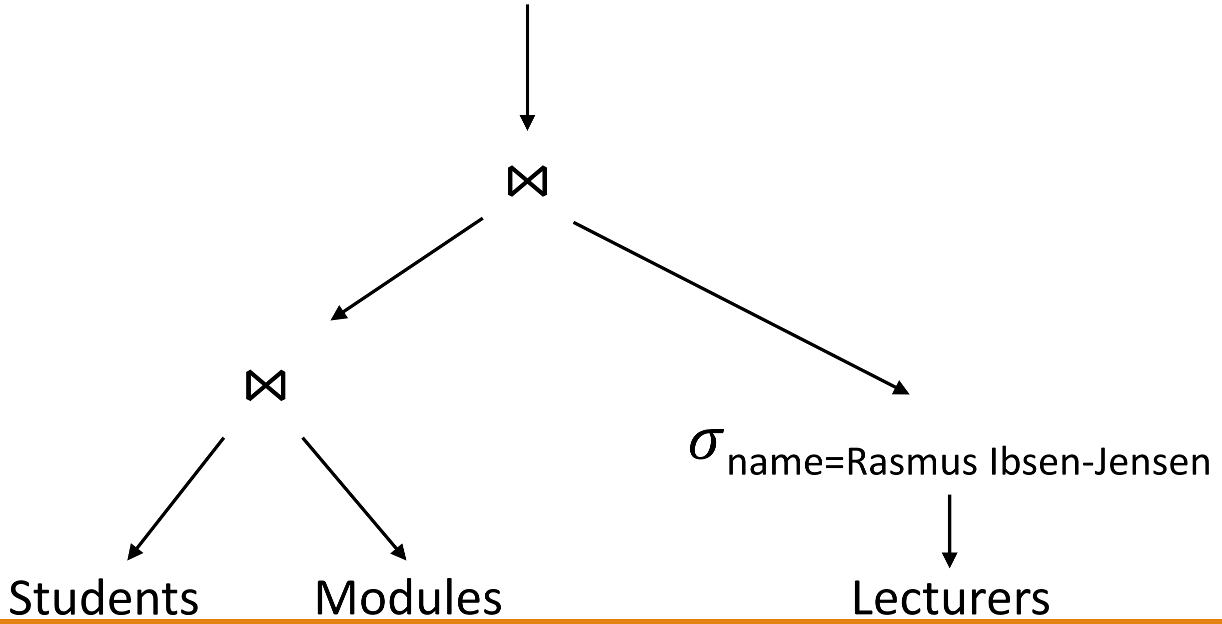
- based on size of intermediate result

Selection of a **good join order**

- also based on size of intermediate results

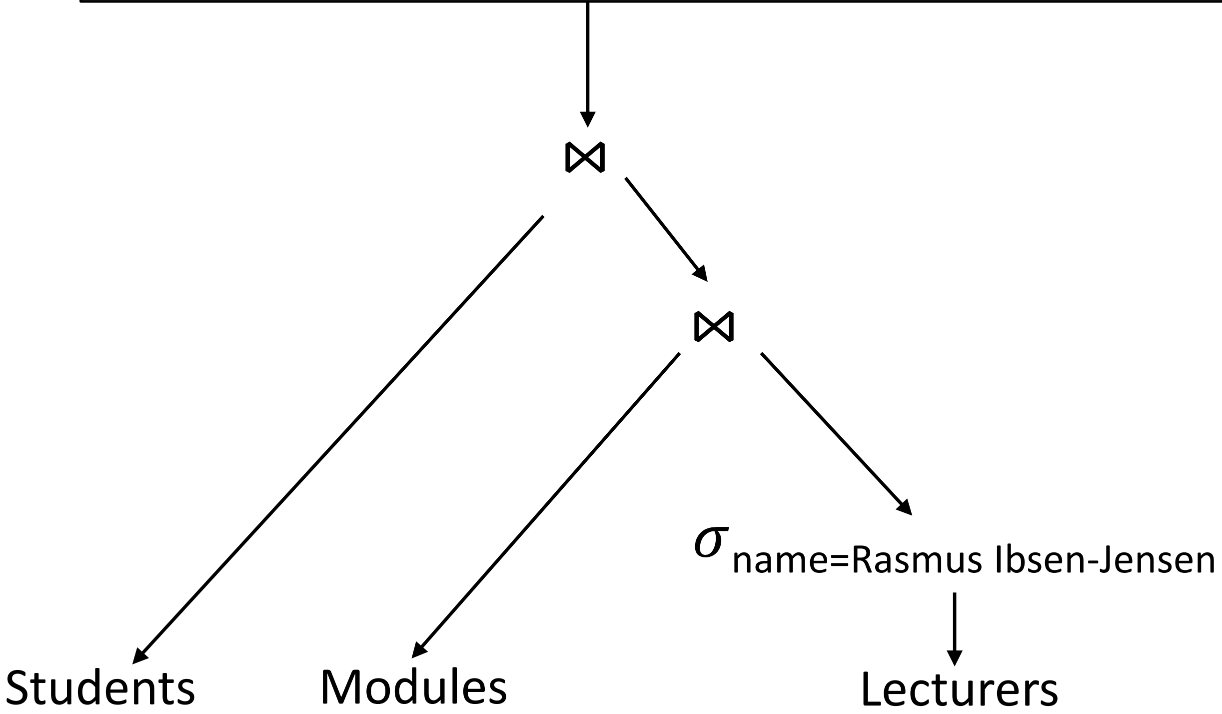
Example where join order matters

```
SELECT *
FROM Lecturers NATURAL JOIN Modules NATURAL JOIN Students
WHERE Lecturers.name = Rasmus Ibsen-Jensen
```



Example where join order matters

```
SELECT *
FROM Lecturers NATURAL JOIN Modules NATURAL JOIN Students
WHERE Lecturers.name = Rasmus Ibsen-Jensen
```



Other Issues

How to generate physical query plans?

- Explore all?
- More sensible approaches: top-down/bottom-up

Selection of a **suitable algorithm** for each operator

- based on size of intermediate result

Selection of a **good join order**

- also based on size of intermediate results

How to **pass information** from one operator to another?

Passing Information

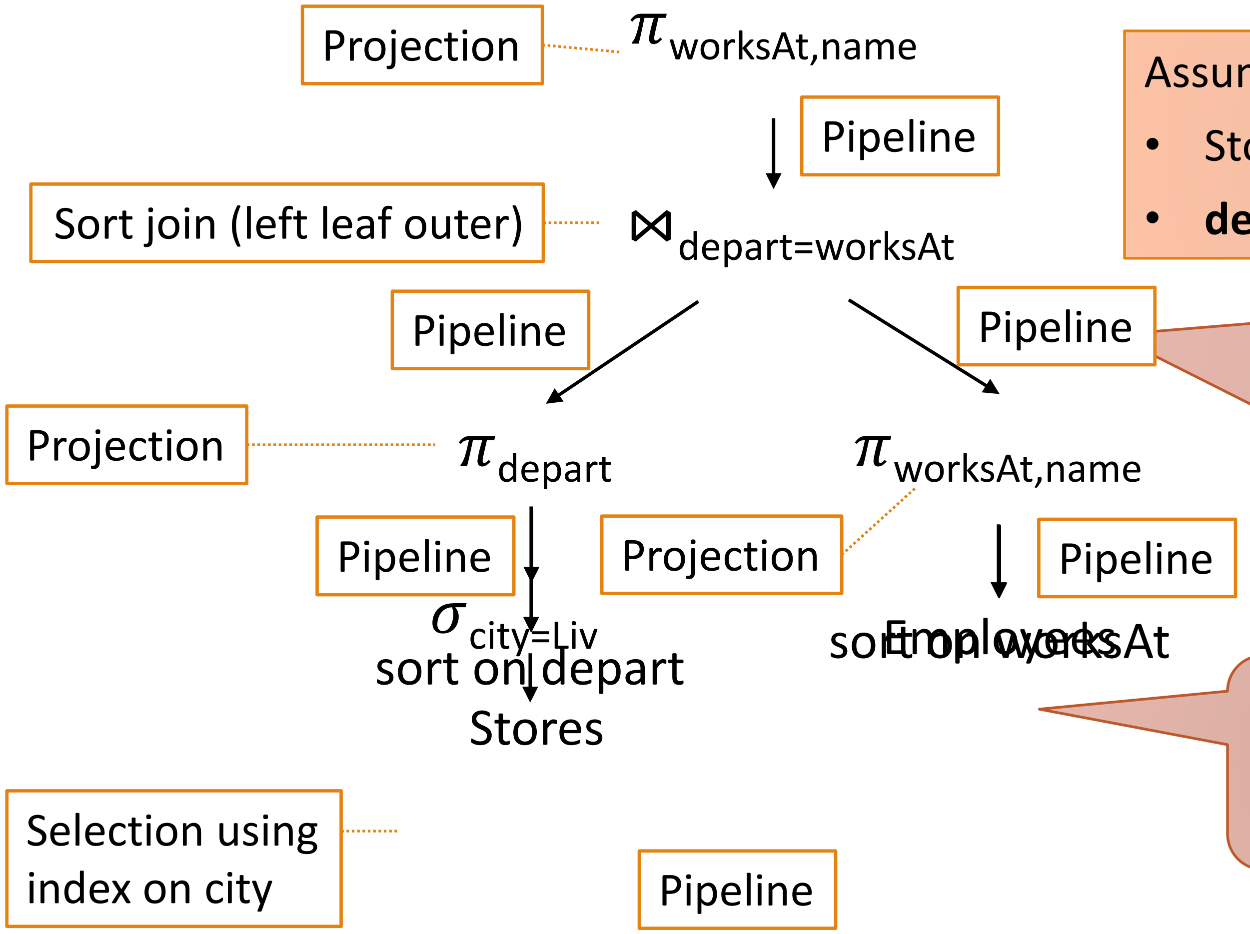
Materialisation: write intermediate results to disk

Pipelining ("stream-based processing")

- Passes the tuples of one operation directly to the next operation without using disk
- Extra buffer for each pair of adjacent operations to hold tuples passing from one relation to the other
- Example:
  - $\pi_{title,year}(\sigma_{length>=100 \text{ AND } studio='Fox'}(Film))$
  - With pipelining, the intermediate result of the selection will be written into a buffer in memory, from which the projection operator will read and process these tuples directly

Film $\rightarrow$ $\boxed{\text{Buffer}}$ $\rightarrow$ $\sigma_{length>=100 \text{ AND } studio='Fox'}$ $\rightarrow$ $\boxed{\text{Buffer}}$ $\rightarrow$ $\pi_{title,year}$

From Logical to Physical Query Plan



Assumption:

- Stores has index on **city**
- **depart** is a key of Stores

Can’t always do this pipeline here – depends on the amount of overlap Since depart is key of stores it is fine here (we can always do it on the left, due to it being the outer one here)

Can’t do pipelines on sort, due to requiring multiple scans

Summary

In this part, we have seen how to go from a SQL query to how exactly we are going to solve it!

Specifically, in this video, we saw how to go from a optimised logical query plan to a physical ditto