Canonical Query Translation

Canonical translation of SQL queries into algebra expressions.
Structure:

select distinct $a_1, \ldots, a_n$
from $R_1, \ldots, R_k$
where p

Restrictions:

- only **select distinct** (sets instead of bags)
- no **group by**, **order by**, **union**, **intersect**, **except**
- only attributes in **select** clause (no computed values)
- no nested queries, no views
- not discussed here: NULL values

From Clause

1. Step: Translating the **from** clause

Let $R_1, \ldots, R_k$ be the relations in the **from** clause of the query. Construct the expression:

$$F = \begin{cases} R_1 & \text{if } k = 1 \\ ((\ldots(R_1 \times R_2) \times \ldots) \times R_k) & \text{else} \end{cases}$$

Where Clause

2. Step: Translating the **where** clause

Let p be the predicate in the **where** clause of the query (if a **where** clause exists).
Construct the expression:

$$W = \begin{cases} F & \text{if there is no } \textbf{where} \text{ clause} \\ \sigma_p(F) & \text{otherwise} \end{cases}$$

Select Clause

3. Step: Translating the **select** clause

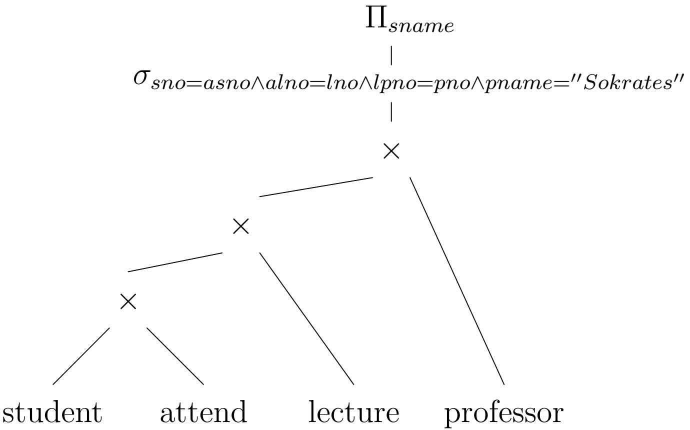
Let $a_1, \ldots, a_n$ (or "*") be the projection in the **select** clause of the query. Construct the expression:

$$S = \begin{cases} W & \text{if the projection is "*"} \\ \Pi_{a_1,\ldots,a_n}(W) & \text{otherwise} \end{cases}$$

4. Step: S is the canonical translation of the query.

Sample Query

select distinct $s.sname$
from $student\ s, attend\ a, lecture\ l, professor\ p$
where $s.sno = a.asno$ **and** $a.alno = l.lno$ **and**
$l.lpno = p.pno$ and $p.pname ='' Sokrates''$



Extension - Group By Clause

2.5. Step: Translating the **group by** clause. Not part of the "canonical" query translation!

Let $g_1, \ldots, g_m$ be the attributes in the **group by** clause and *agg* the aggregations in the **select** clause of the query (if a **group by** clause exists). Construct the expression:

$$G = \begin{cases} W & \text{if there is no \textbf{group by} clause} \\ \Gamma_{g_1,\ldots,g_m;agg}(W) & \text{otherwise} \end{cases}$$

use G instead of W in step 3.

Optimization Phases

Textbook query optimization steps:

1. translate the query into its canonical algebraic expression
2. perform logical query optimization
3. perform physical query optimization

we have already seen the translation, from now one assume that the algebraic expression is given.

Concept of Logical Query Optimization

- foundation: algebraic equivalences
- algebraic equivalences span the potential search space
- given an initial algebraic expression: apply algebraic equivalences to derive new (equivalent) algebraic expressions
- note: algebraic equivalences do not indicate a direction, they can be applied in both ways
- the conditions attached to the equivalences have to be checked

Algebraic equivalences are essential:

- new equivalences increase the potential search space
- better plans
- but search more expensive

Performing Logical Query Optimization

Which plans are better?

- plans can only be compared if there is a *cost function*
- cost functions need details that are not available when only considering logical algebra
- consequence: logical query optimization remains a heuristic

Performing Logical Query Optimization

Most algorithms for logical query optimization use the following strategies:

- organization of equivalences into groups
- directing equivalences

Directing means specifying a preferred side.
A *directed equivalences* is called a *rewrite rule*. The groups of rewrite rules are applied sequentially to the initial algebraic expression. Rough goal:

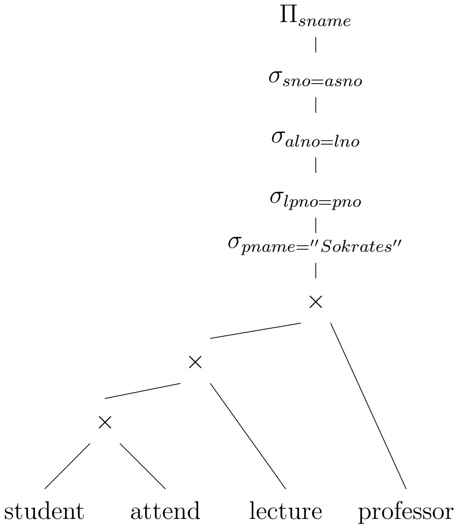
reduce the size of intermediate results

Phases of Logical Query Optimization

1. break up conjunctive selection predicates
   (equivalence (1) $\rightarrow$)
2. push selections down
   (equivalence (2) $\rightarrow$, (14) $\rightarrow$)
3. introduce joins
   (equivalence (13) $\rightarrow$)
4. determine join order
   (equivalence (9), (10), (11), (12))
5. introduce and push down projections
   (equivalence (3) $\leftarrow$, (4) $\leftarrow$, (16) $\rightarrow$)

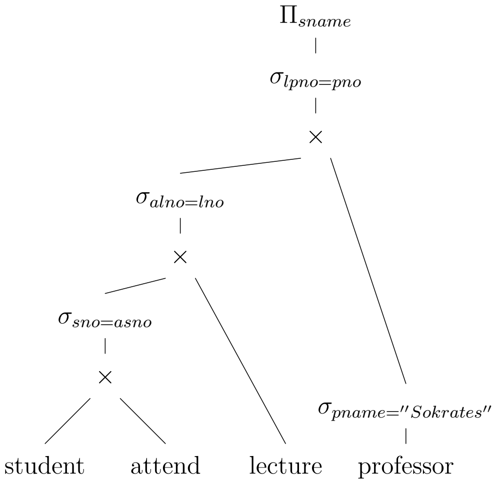
Step 1: Break up conjunctive selection predicates

- selection with simple predicates can be moved around easier



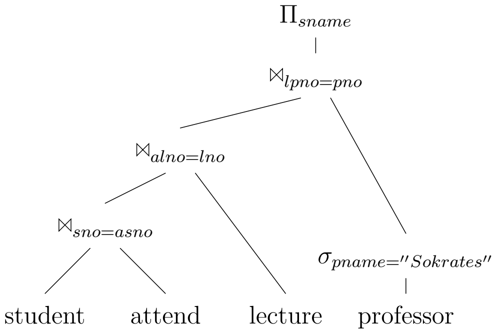
Step 2: Push Selections Down

- reduce the number of tuples early, reduces the work for later operators



Step 3: Introduce Joins

- joins are cheaper than cross products



Step 4: Determine Join Order

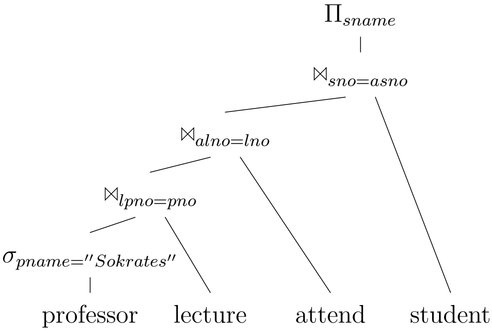
- costs differ vastly
- difficult problem, NP hard (next chapter discusses only join ordering)

Observations in the sample plan:

- bottom most expression is $student \bowtie_{sno=asno} attend$
- the result is huge, all students, all their lectures
- in the result only one professor relevant $\sigma_{name=''Sokrates''}(professor)$
- join this with lecture first, only lectures by him, much smaller

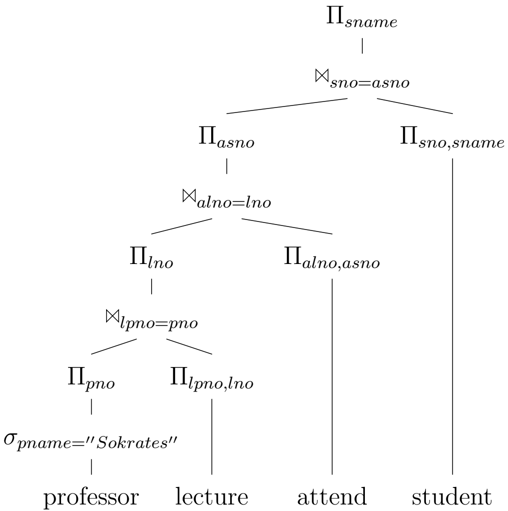
Step 4: Determine Join Order

- intermediate results much smaller



Step 5: Introduce and Push Down Projections

- eliminate redundant attributes
- only before pipeline breakers

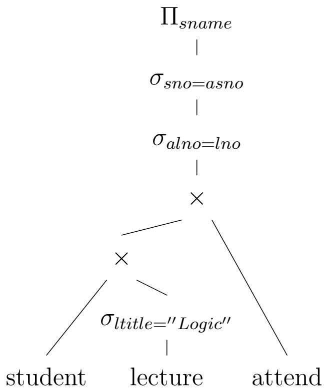


Limitations

Consider the following SQL query

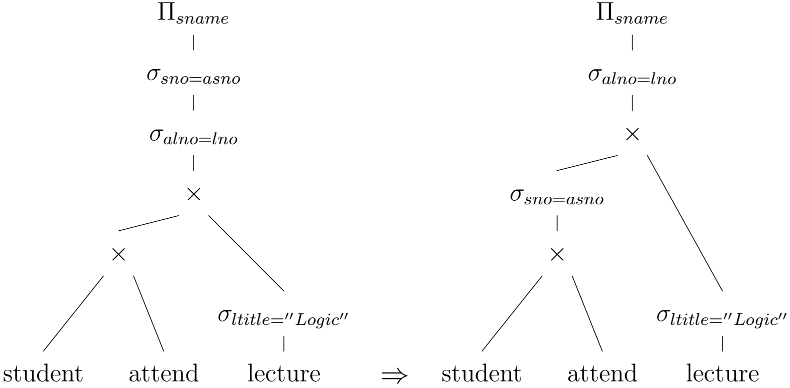
select distinct *s.sname*
from *student s, lecture l, attend a*
where $s.sno = a.asno$ **and** $a.alno = l.lno$ **and** $l.ltitle ='' Logic''$

Steps 1-2 could result in plan below. No further selection push down.



Limitations

However a different join order would allow further push down:



- the phases are interdependent
- the separation can loose the optimal solution

Physical Query Optimization

- add more execution information to the plan
- allow for cost calculations
- select index structures/access paths
- choose operator implementations
- add property enforcer
- choose when to materialize (temp/DAGs)

Access Paths Selection

- scan+selection could be done by an index lookup
- multiple indices to choose from
- table scan might be the best, even if an index is available
- depends on selectivity, rule of thumb: 10%
- detailed statistics and costs required
- related problem: materialized views
- even more complex, as more than one operator could be substitued

Operator Selection

- replace a logical operator (e.g. $\bowtie$) with a physical one (e.g. $\bowtie^{HH}$)
- semantic restrictions: e.g. most join operators require equi-conditions
- $\bowtie^{BNL}$ is better than $\bowtie^{NL}$
- $\bowtie^{SM}$ and $\bowtie^{HH}$ are usually better than both
- $\bowtie^{HH}$ is often the best if not reusing sorts
- decission must be cost based
- even $\bowtie^{NL}$ can be optimal!
- not only joins, has to be done for all operators

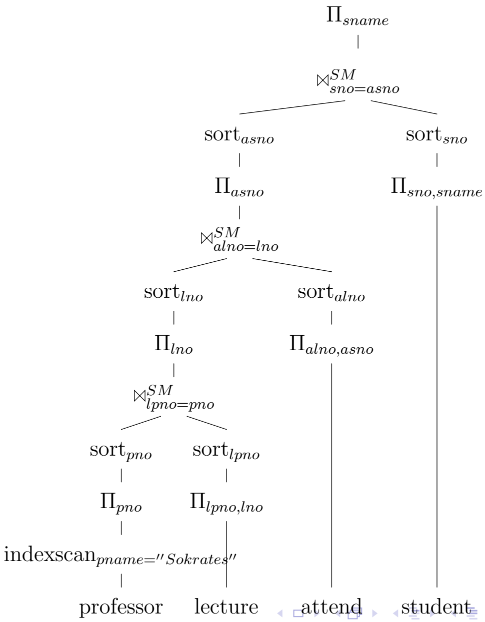
Property Enforcer

- certain physical operators need certain properties
- typical example: sort for $\bowtie^{SM}$
- other example: in a distributed database operators need the data locally to operate
- many operator requirements can be modeled as properties (hashing etc.)
- have to be guaranteed as needed

Materializing

- sometimes materializing is a good idea
- temp operator stores input on disk
- essential for multiple consumers (factorization, DAGs)
- also relevant for $\Join^{NL}$
- first pass expensive, further passes cheap

Physical Plan for Sample Query



Outlook

- separation in two phases looses optimality
- many decissions (e.g. view resolution) important for logical optimization
- textbook physical optimization is incomplete
- did not discuss cost calculations
- will look at this again in later chapters

3. Join Ordering

- Basics
- Search Space
- Greedy Heuristics
- IKKBZ
- MVP
- Dynamic Programming
- Generating Permutations
- Transformative Approaches
- Randomized Approaches
- Metaheuristics
- Iterative Dynamic Programming
- Order Preserving Joins

Queries Considered

Concentrate on join ordering, that is:

- conjunctive queries
- simple predicates
- predicates have the form $a_1 = a_2$ where a_1 is an attribute and a_2 is either an attribute or a constant
- even ignore constants in some algorithms

We join relations $R_1, \ldots, R_n$, where R_i can be

- a base relation
- a base relation including selections
- a more complex building block or access path

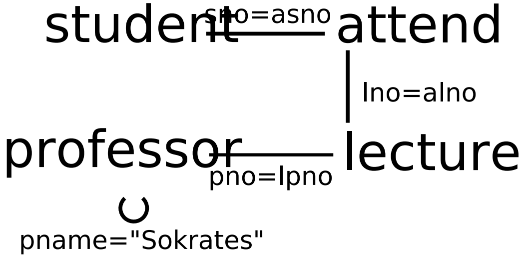
Pretending to have a base relation is ok for now.

Query Graph

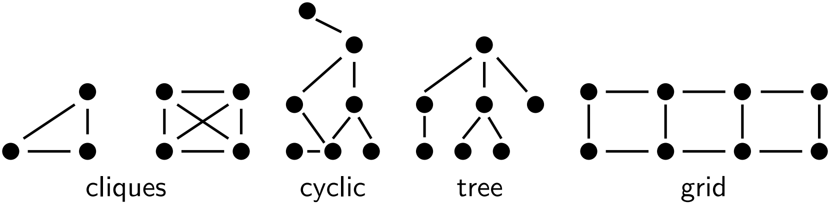
Queries of this type can be characterized by their query graph:

- the query graph is an undirected graph with $R_1, \ldots, R_n$ as nodes
- a predicate of the form $a_1 = a_2$, where $a_1 \in R_i$ and $a_2 \in R_j$ forms an edge between R_i and R_j labeled with the predicate
- a predicate of the form $a_1 = a_2$, where $a_1 \in R_i$ and a_2 is a constant forms a self-edge on R_i labeled with the predicate
- most algorithms will not handle self-edges, they have to be pushed down

Sample Query Graph



Shapes of Query Graphs



- real world queries are somewhere in-between
- chain, cycle, star and clique are interesting to study
- they represent certain kind of problems and queries