Advanced Database Systems

Spring 2024

Lecture #20:

Query Optimisation: Plan Space

R&G: Chapter 15

Query Optimisation

The bridge between a **declarative** domain-specific language…

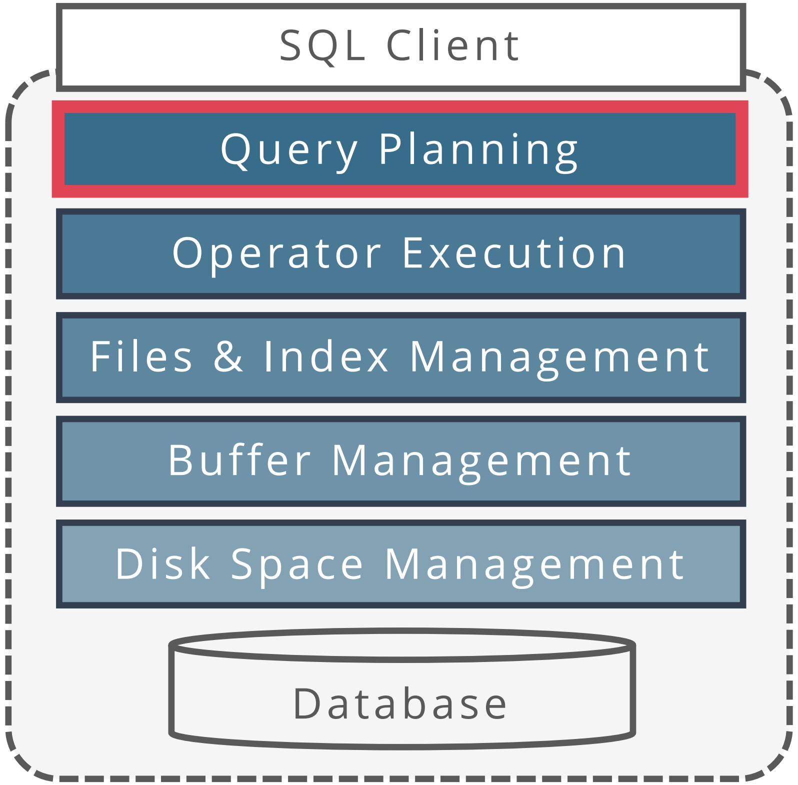
- "What" you want as an answer

… and custom **imperative** computer programs

- "How" to compute the answer

A lot of effort has been spent on this problem!

- Huge optimisation problem
- Big impact on performance!



Query Optimisation: The Goal

For a given query, find a **correct** execution plan that has the lowest "cost"

This is the part of a DBMS that is the hardest to implement well

- Proven to be NP-hard

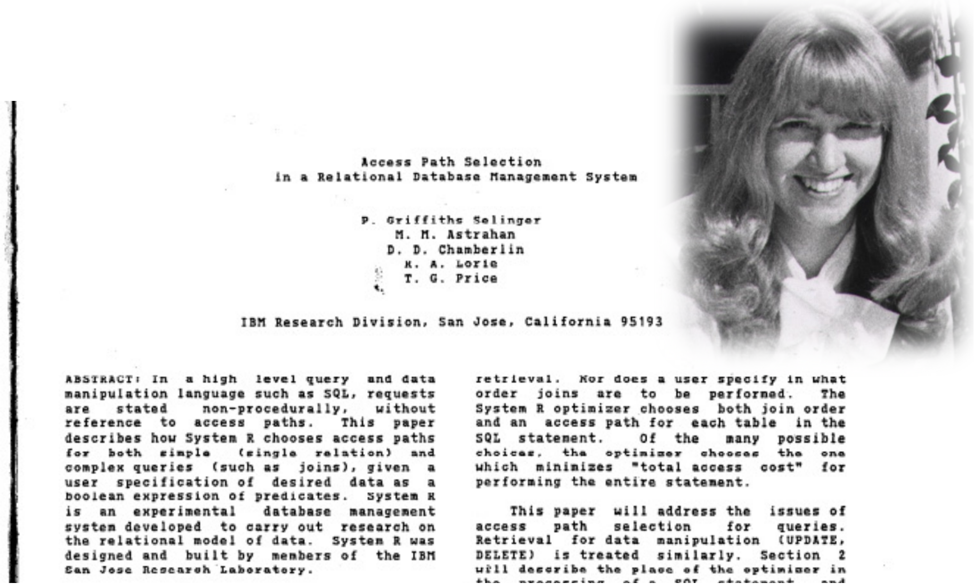
No optimizer truly produces the "optimal" plan

- Use estimation techniques to guess real plan cost
- Use heuristics to limit the search space
- At the very least, avoid really bad plans!

Query Optimisation Strategies

We will focus on **IBM's System R** optimisers

- Invented in 1979 by Pat Selinger et al.
- A lot of the concepts from System R's optimiser still used today in most DB systems



Access Path Selection
in a Relational Database Management System

P. Griffiths Selinger
M. M. Astrahan
D. D. Chamberlin
R. A. Lorie
T. G. Price

IBM Research Division, San Jose, California 95193

ABSTRACT: In a high level query and data manipulation language such as SQL, requests are stated non-procedurally, without reference to access paths. This paper describes how System R chooses access paths for both simple (single relation) and complex queries (such as joins), given a user specification of desired data as a boolean expression of predicates. System R is an experimental database management system developed to carry out research on the relational model of data. System R was designed and built by members of the IBM San Jose Research Laboratory.

retrieval. Nor does a user specify in what order joins are to be performed. The System R optimizer chooses both join order and an access path for each table in the SQL statement. Of the many possible choices, the optimizer chooses the one which minimizes "total access cost" for performing the entire statement.

This paper will address the issues of access path selection for queries. Retrieval for data manipulation (UPDATE, DELETE) is treated similarly. Section 2 will describe the place of the optimizer in

Other optimisation strategies

- Volcano / Cascades (SQL Server, Greenplum)
- Stratified search (IBM DB2, Oracle)
- Randomised search (PostgreSQL)
- AI-driven optimisation

Notable differences,
but similar big picture

Query Lifecycle

SQL Parser

Checks correctness, authorisation

Generates a parse tree

Heuristics / rule-based rewriting

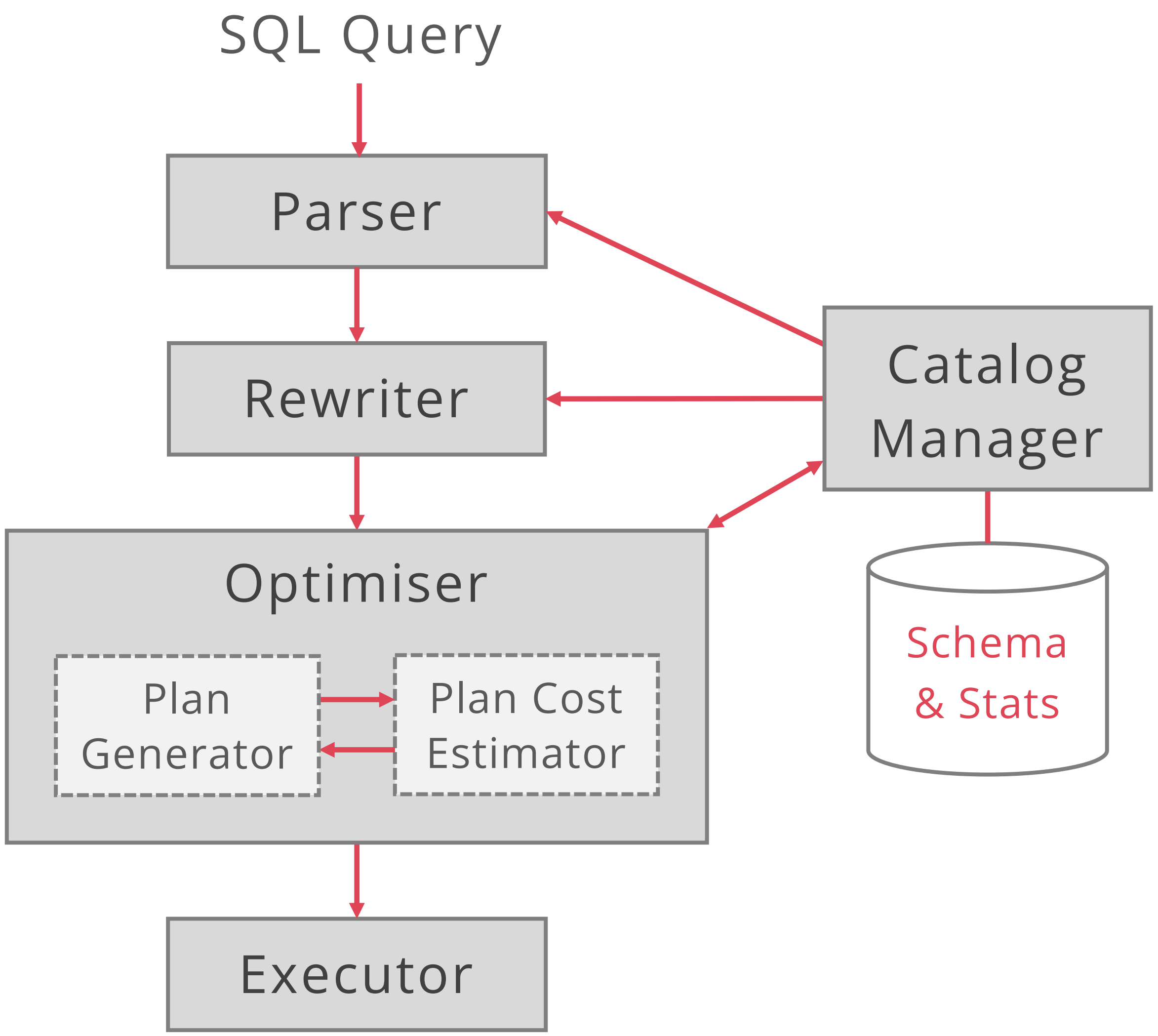
Remove stupid / inefficient things

Apply equivalence rules of RA

Cost-based optimisation

Enumerate multiple equivalent plans

Using a **cost model** pick the cheapest plan



QUERY PARSER

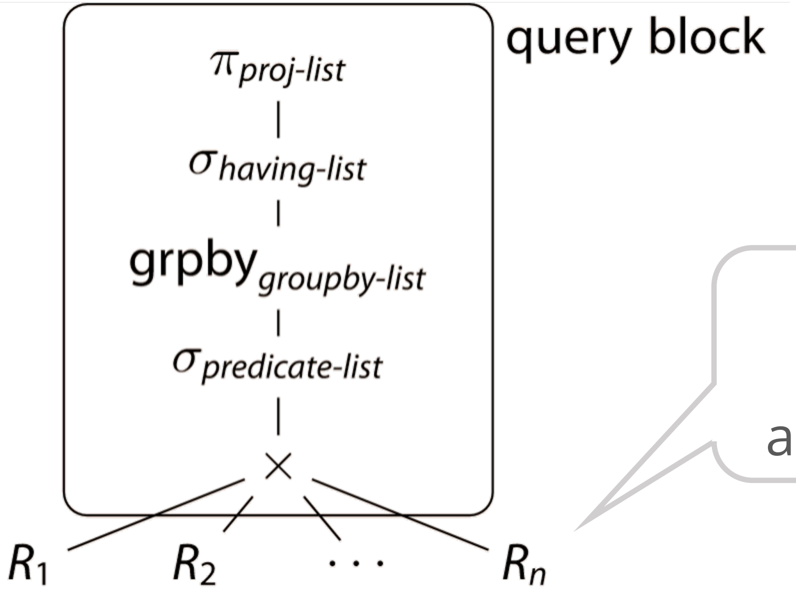
Performs syntactic & semantic analysis

Builds internal representation of the input query

SELECT-FROM-WHERE clauses translated into **query blocks**

SELECT *proj-list*
FROM $R_1, R_2, \ldots, R_n$
WHERE *predicate-list*
GROUP BY *groupby-list*
HAVING *having-list*

$\rightarrow$



Each R_i can be a base relation or another query block

Query Rewriter

Two relational algebra expressions are **equivalent** if they generate the same set of tuples on any given database instance

The query rewriter applies heuristics & RA rules, without looking into the actual database state (no info about cardinalities, indices, etc.)

Separated from cost-based optimisation to reduce search space

- Often only a few, very useful rules are applied
- Typically too expensive to explore all possibilities
- Rule-system often not confluent

Some Simplifications

```
CREATE TABLE R (
  id  INT PRIMARY KEY,
  val INT NOT NULL )
```

Impossible / unnecessary predicates

```
SELECT * FROM R WHERE 1 = 0
```

empty result

```
SELECT * FROM R WHERE 1 = 1
```

→

```
SELECT * FROM R
```

Join elimination

```
SELECT R1.*
  FROM R AS R1 JOIN R AS R2
    ON R1.id = R2.id
```

→

```
SELECT * FROM R
```

More Simplifications

```
CREATE TABLE R (
  id  INT PRIMARY KEY,
  val INT NOT NULL )
```

Ignoring nested subquery

```
SELECT * FROM R AS R1
 WHERE EXISTS (SELECT * FROM R AS R2
                WHERE R1.id = R2.id);
```

```
SELECT * FROM R
```

Merging predicates

```
SELECT * FROM R
 WHERE val BETWEEN 1 AND 100
    OR val BETWEEN 50 AND 150
```

```
SELECT * FROM R
 WHERE val BETWEEN 1 AND 150
```

Query Optimiser

Optimises one query block at a time

Enumerates all possible plans

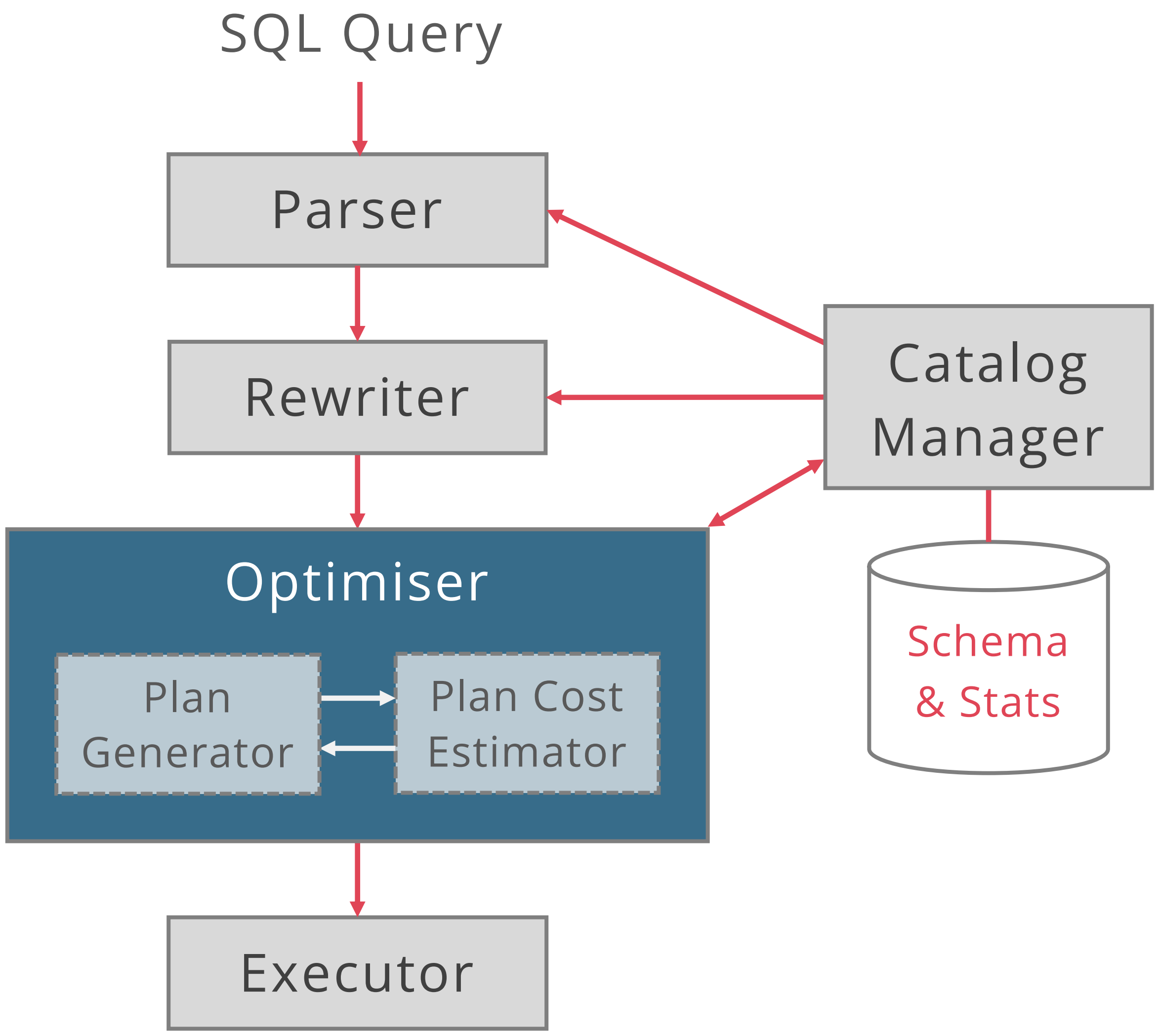
If this yields too many plans, at least enumerate “promising” plan candidates

Determines the **cost** of each plan

… using a cost model and catalog statistics

Chooses the **best** plan per query block

Often not truly “optimal”



Query Optimisation: The Components

Three (mostly) orthogonal concerns:

- **Plan space**
  - For a given query, what plans are considered?
  - Larger the plan space, more likely to find a cheaper plan, but harder to search
- **Cost estimation**
  - How is the cost of a plan estimated?
  - Want to find the cheapest plan
- **Search strategy**
  - How do we "search" in the "plan space"?

Plan Space

To generate a space of candidate plans, we need to think about how to rewrite relational algebra expressions into other ones

Therefore, need a set of **equivalence rules**

Relational Algebra Equivalences

Selections

$\sigma_{c1 \land c2 \land \ldots \land cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots \sigma_{cn}(R)))$ (cascade)

$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (commute)

Projections

$\pi_{a1}(\ldots (R)\ldots) \equiv \pi_{a1}(\ldots (\pi_{a1, \ldots, an-1}(R)) \ldots)$ (cascade)

Essentially, allows partial projection earlier in the expression

As long as we're keeping a1 (and everything else we need outside) we're OK

Relational Algebra Equivalences

Selections

$\sigma_{c1 \wedge c2 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots \sigma_{cn}(R)))$ (cascade)

$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (commute)

Projections

$\pi_{a1}(\ldots (R)\ldots) \equiv \pi_{a1}(\ldots (\pi_{a1, \ldots, an-1}(R)) \ldots)$ (cascade)

Cartesian products

$R \times (S \times T) \equiv (R \times S) \times T$ (associative)

$R \times S \equiv S \times R$ (commutative)

Recall that the ordering of attributes doesn't matter

Are Joins Associative and Commutative?

After all, just Cartesian products with selections

You can think of them as associative and commutative
… but beware of joins turning into cross-products!

Consider R(A,Y), S(A,B), T(B,Z)

Attempt 1: $(S \bowtie_{S.B=T.B} T) \bowtie_{S.A=R.A} R \not\equiv S \bowtie_{S.B=T.B} (T \bowtie_{S.A=R.A} R)$ — **Not legal!** (join on A not allowed)

Attempt 2: $(S \bowtie_{S.B=T.B} T) \bowtie_{S.A=R.A} R \not\equiv S \bowtie_{S.B=T.B} (T \times R)$ — **Not the same!** (no condition on A)

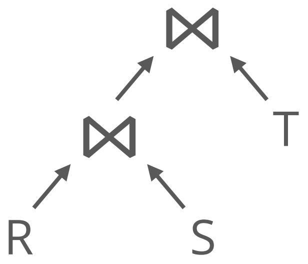
Attempt 3: $(S \bowtie_{S.B=T.B} T) \bowtie_{S.A=R.A} R \equiv S \bowtie_{S.B=T.B \wedge S.A=R.A} (T \times R)$ — **The same!**

JOIN ORDERING

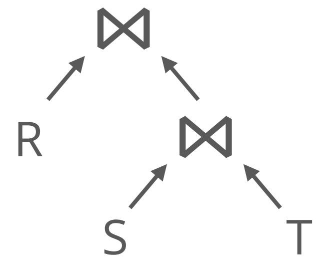
Similarly, note that some join orders have cross products, some don't

Equivalent for the query on the right:

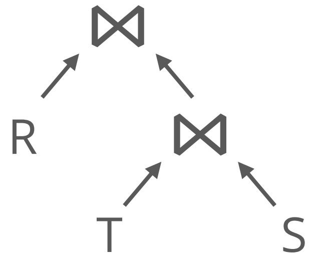
```
SELECT *
  FROM R, S, T
 WHERE R.A = S.A
   AND S.B = T.B
```



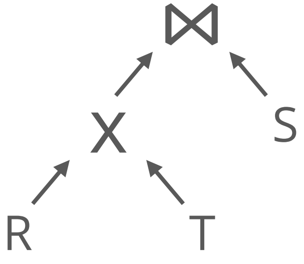
$(R \bowtie_{R.A=S.A} S) \bowtie_{S.B=T.B} T$



$R \bowtie_{R.A=S.A} (S \bowtie_{S.B=T.B} T)$



$R \bowtie_{R.A=S.A} (T \bowtie_{S.B=T.B} S)$



$(R \times T) \bowtie_{R.A=S.A \wedge S.B=T.B} S$

INTRODUCING ADDITIONAL JOIN CONDITIONS

Implicit join through transitivity…

```
SELECT * FROM R, S, T
 WHERE R.A = S.B AND S.B = T.C
```

… can be turned into

```
SELECT * FROM R, S, T
 WHERE R.A = S.B AND S.B = T.C AND R.A = T.C
```

… making the join ordering **(R ⋈ T) ⋈ S** possible (avoids a Cartesian product)

Plan Space

To generate a space of candidate plans, we need to think about how to rewrite relational algebra expressions into other ones

Therefore, need a set of equivalence rules – **done**

Need heuristics to restrict attention to plans that are mostly better

- We have already seen one of these in the relational algebra lecture

Common Heuristics: Selections

Filter as early as possible

Reorder predicates so that the DBMS applies the most selective one first

Break complex predicates and push down

$$\sigma_{c1 \wedge c2 \wedge \ldots \wedge cn}(R) = \sigma_{c1}(\sigma_{c2}(\ldots \sigma_{cn}(R)))$$

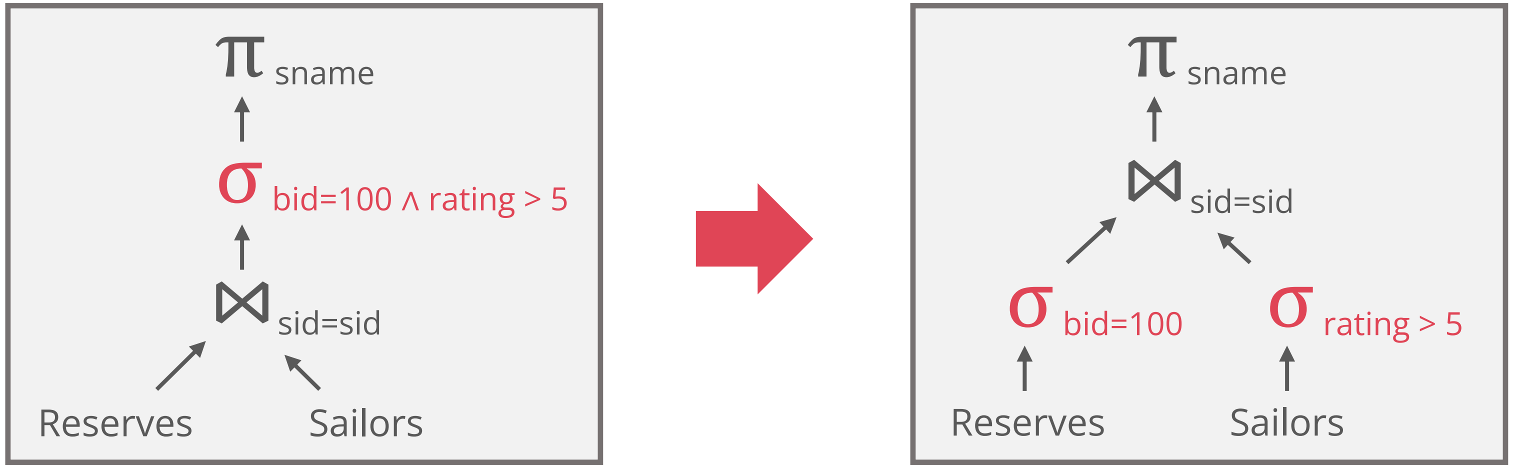
Simplify complex predicates

X = Y AND Y = 3 ⇒ X = 3 AND Y = 3

L.TAX * 100 < 5 ⇒ L.TAX < 0.05

Heuristics: Selection Pushdown

Apply selections as soon as you have the relevant columns



Why is this an improvement?

- Selection is essentially free, joins are expensive
- Side effect is that the intermediate inputs to joins are smaller

Common Heuristics: Projections

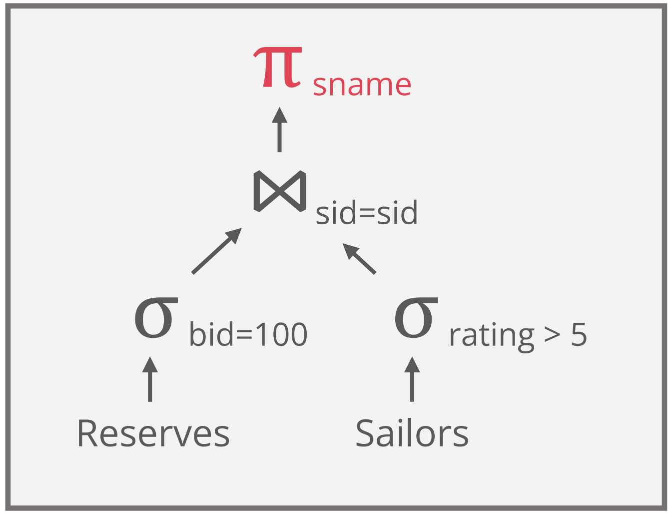
Perform them early to create smaller tuples and reduce intermediate results (if duplicates are eliminated)

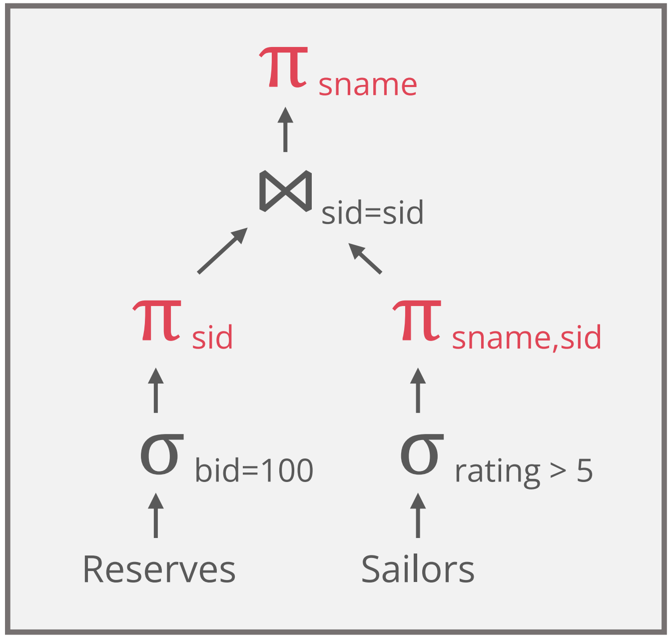
Project out all attributes except the ones requested or required (e.g., joining keys)

This is not important for column stores…

Heuristics: Projection Pushdown

Keep only the columns you need to evaluate downstream operators





Other rewritings exist! (reorder selection and projection)

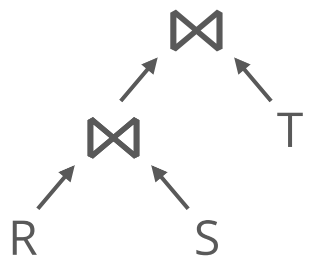
Common Heuristics

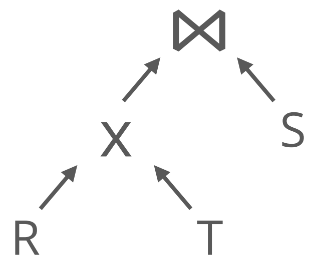
Avoid Cartesian products

Given a choice, do theta-joins rather than cross-products

Consider R(A,B), S(B,C), T(C,D)

Favour (R ⋈ S) ⋈ T over (R x T) ⋈ S





Case where this doesn't quite improve things:

- If R x T is small (e.g., R & T are very small and S is relatively large)
- Still it's a good enough heuristic that we will use it

Plan Space

To generate a space of candidate plans, we need to think about how to rewrite relational algebra expressions into other ones

Therefore, need a set of equivalence rules – **done**

Need heuristics to restrict attention to plans that are mostly better – **done**

Both of these were logical equivalences, need also physical equivalences

Physical Equivalences

Base table access

- Heap scan
- Index scan (if available on referenced columns)

Equijoins

- Block Nested Loops: simple, exploits extra memory
- Index Nested Loops: often good if 1 table is small and the other indexed properly
- Sort-Merge Join: good with small memory, equal-size tables
- Grace Hash Join: even better than sort with 1 small table

Non-Equijoins

- Block Nested Loops

Summary

- There are lots of plans
  - Even for a relatively simple query
  - Manual query planning can be tedious, technical
  - Machines are better at enumerating options than people
- Query rewriting
  - DBMSs can identify better query plans even without a cost model
  - Filtering as early as possible is usually a good choice