CS411
Database Systems

09: Query Optimization

answers all the same
"performance"

$Q \rightsquigarrow P(Q)$

that is optimal

Why this?

i) We really want to speed.

TPC measure perf of DBMS on Hardware

Why not "programmer"?

- difficult $\Rightarrow$ auto programmy
- vary every time

ii) Engineering Science

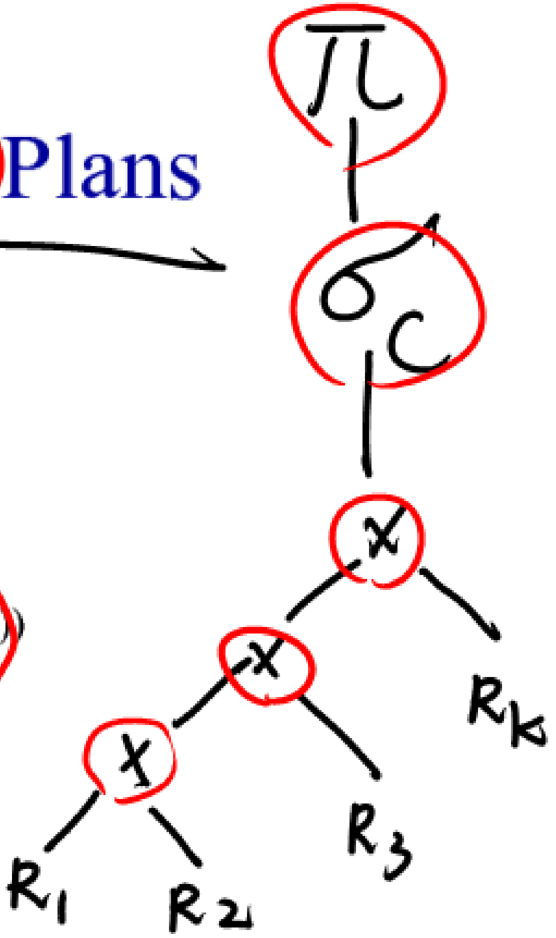
Optimization

- At the heart of the database engine
- Step 1: convert the SQL query to some logical plan (Rel Alg)
- Step 2: find a better logical plan, find an associated $\boxed{\text{physical}}$ plan

→ concrete w/ all details

SQL –> Logical Query Plans

Converting from SQL to initial Logical Plans



```
Select a1, …, an
From R1, …, Rk
Where C
```

$\pi\ \Pi_{a1,\ldots,an}(\sigma_C(R1 \cancel{\bowtie} R2 \cancel{\bowtie} \ldots \cancel{\bowtie} Rk))$

```
Select a1, …, an
From R1, …, Rk
Where C
Group by b1, …, bl
```

$\Pi_{a1,\ldots,an}(\gamma_{b1, \ldots, bm, aggs}(\sigma_C(R1 \bowtie R2 \bowtie \ldots \bowtie Rk)))$

Skip

Converting Nested Queries

```
Select distinct product.name
From product
Where product.maker in (Select company.name
                        From company
                        where company.city="Urbana")
```

```
Select distinct product.name
From product, company
Where product.maker = company.name AND
      company.city="Urbana"
```

Optimization: Logical Query Plan

- Now we have one logical plan
- Algebraic laws: get to diff logical plan
  - foundation for every optimization
- Two approaches to optimizations:
  - Rule-based (heuristics): apply laws that *seem* to result in cheaper plans
  - Cost-based: estimate size and cost of intermediate results, search systematically for best plan

Stud

dept	name	sid
CS	Alex	A01

Enrollment

sid	cid
A01	CS111

course

title	instructor	cid
OS.	Thomson	CS111

Motivating Example

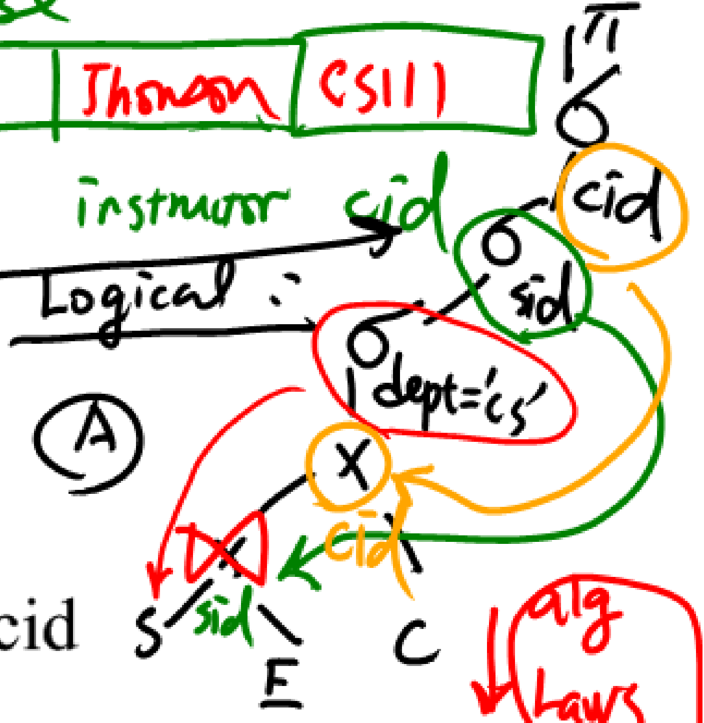
Select S.name, C.instructor

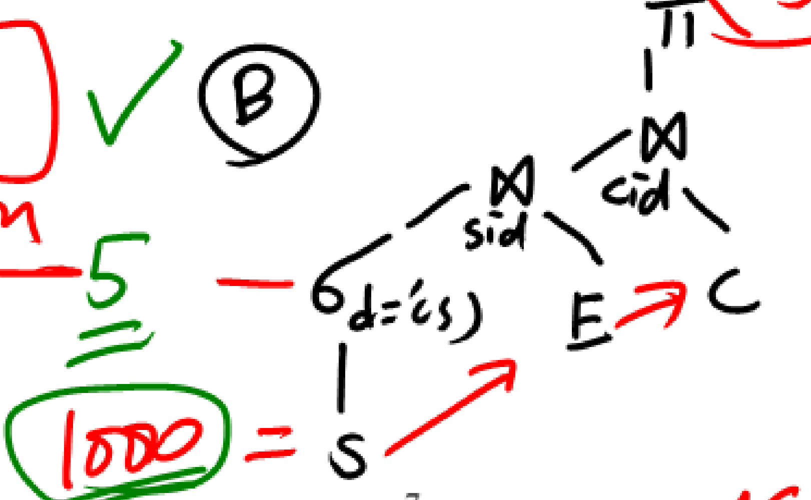
beers sells bars

From Students S, Enrollment E, Course C

Where S.dept = 'CS' and

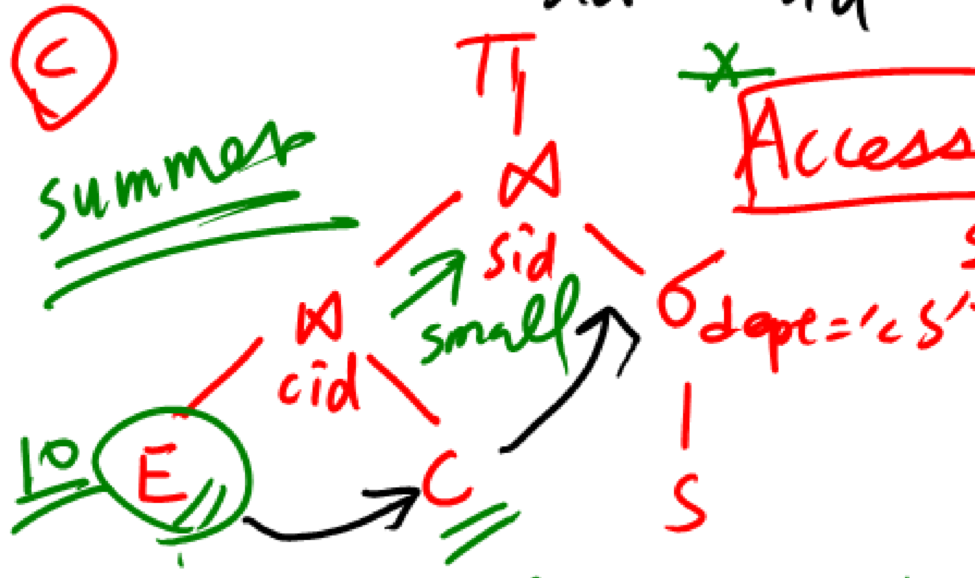
S.~~name~~ sid =E.~~name~~ sid and E.cid = C.cid





* Access Path ✓

selection



* you need auto opt.

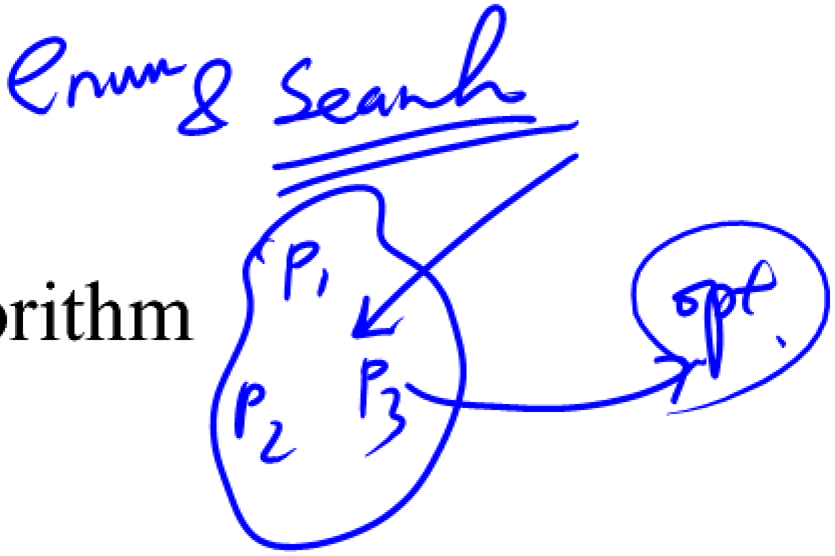
$\sigma_c(R \times S) = R \bowtie_c S$

The three components of an optimizer

We need three things in an optimizer:

- Algebraic laws
- An optimization algorithm
- A cost estimator

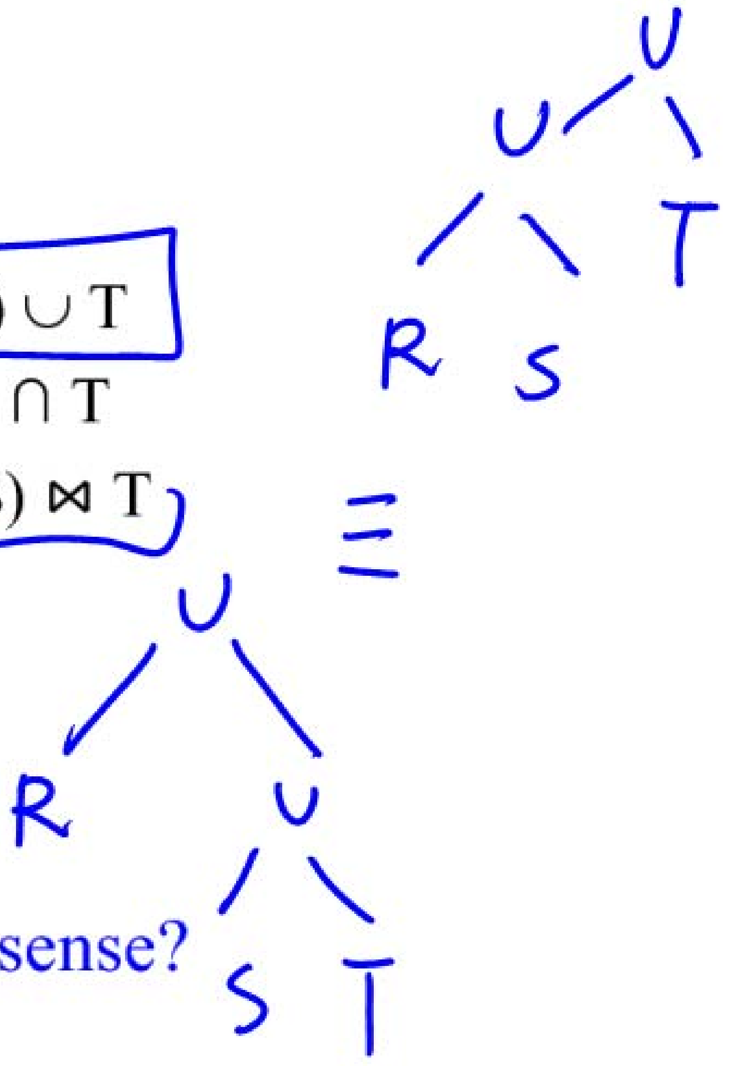
$P_1 \overset{?}{=} P_2$



Algebraic Laws

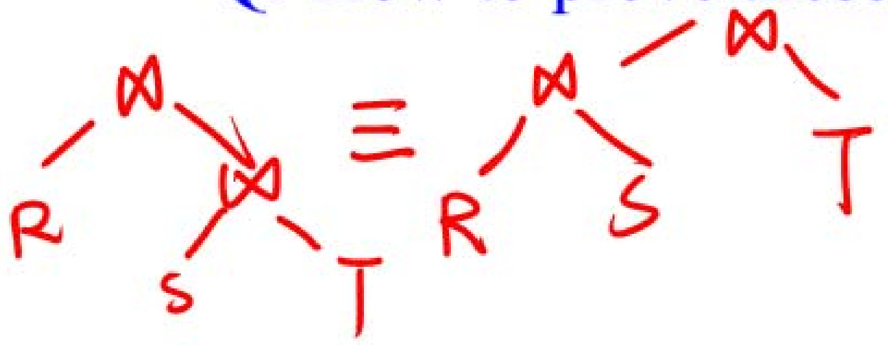
Algebraic Laws

- Commutative and Associative Laws
  - $R \cup S = S \cup R$, $\boxed{R \cup (S \cup T) = (R \cup S) \cup T}$
  - $R \cap S = S \cap R$, $R \cap (S \cap T) = (R \cap S) \cap T$
  - $R \bowtie S = S \bowtie R$, $\underbrace{R \bowtie (S \bowtie T)}_{} = \underbrace{(R \bowtie S) \bowtie T}_{}$
- Distributive Laws
  - $R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$



if $t \in LHS \Longleftrightarrow t \in RHS$

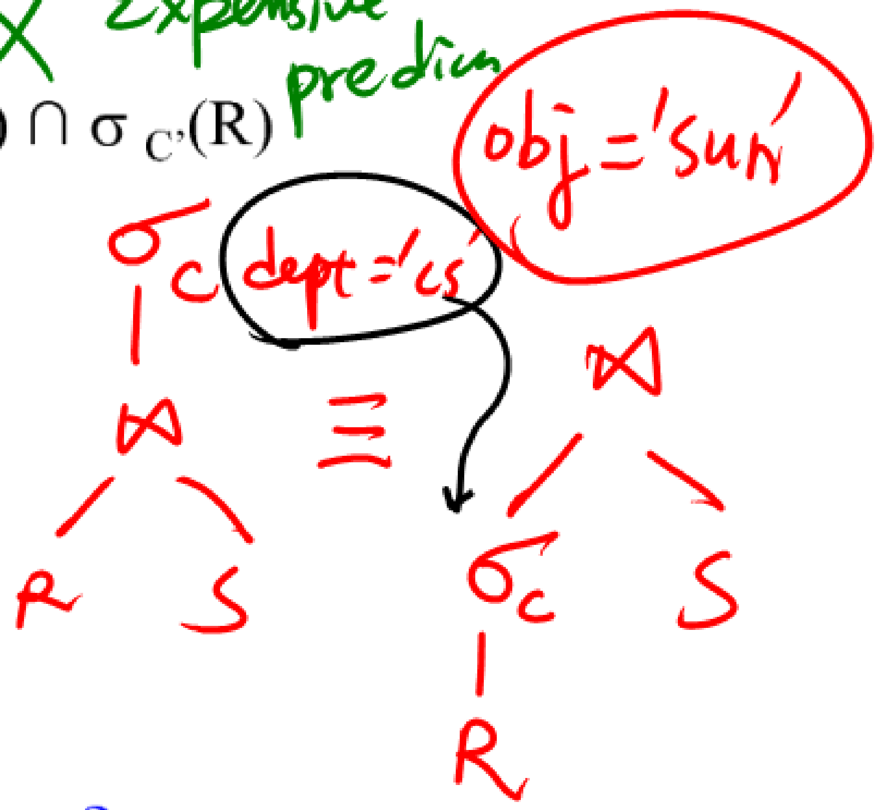
Q: How to prove these laws? Make sense?



Algebraic Laws

almost always ~~true~~

- Laws involving selection: σ
  - $\sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R)$ ✗ Expensive predicate
  - $\sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R)$
  - $\boxed{\sigma_C(R \bowtie S)} = \sigma_C(R) \bowtie S$
    - When C involves only attributes of R
  - $\sigma_C(R - S) = \sigma_C(R) - S$
  - $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
  - $\sigma_C(R \cap S) = \sigma_C(R) \cap S$



Q: What do they mean? Make sense?

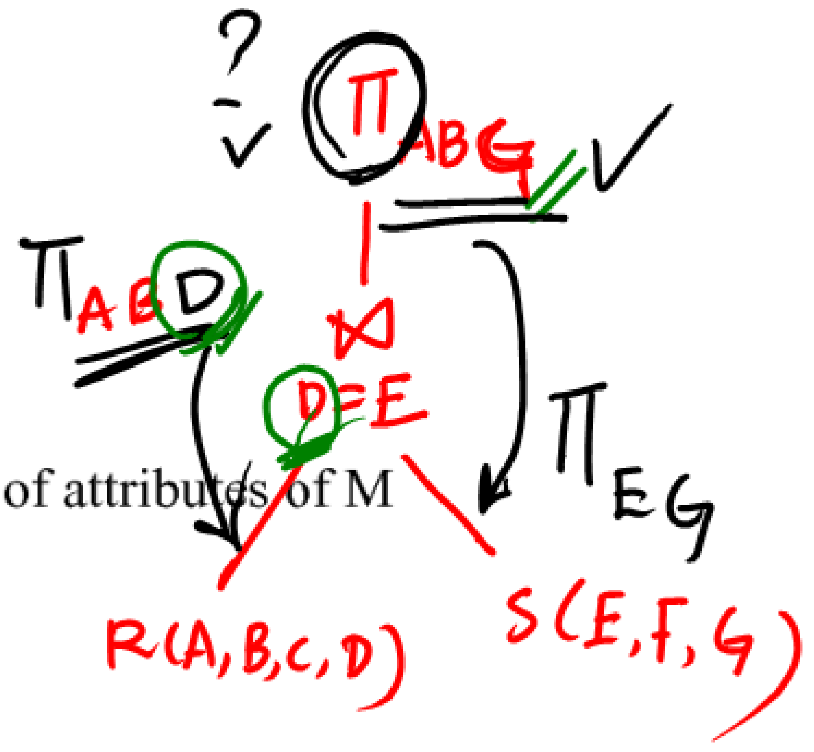
Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)
  - $\sigma_{F=3}(R \bowtie_{D=E} S) =$?
  - $\sigma_{A=5 \text{ AND } G=9}(R \bowtie_{D=E} S) =$?

$\sigma_{F=3}(S)$

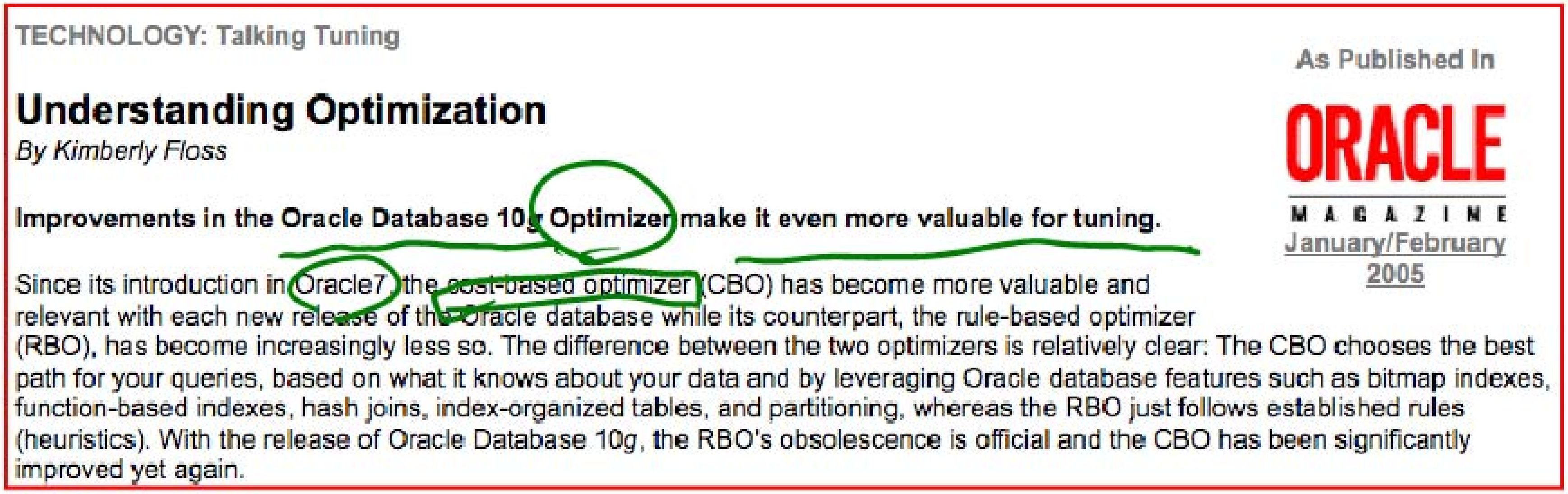
Algebraic Laws

- Laws involving projections
  - $\Pi_M(R \bowtie S) = \Pi_N(\Pi_P(R) \bowtie \Pi_Q(S))$
    - Where N, P, Q are appropriate subsets of attributes of M
  - $\Pi_M(\Pi_N(R)) = \Pi_{M \cap N}(R)$
- Example R(A,B,C,D), S(E, F, G)
  - $\Pi_{A,B,G}(R \bowtie_{D=E} S) = \Pi_{?}(\Pi_?(R) \bowtie \Pi_?(S))$



Q: Again, what do they mean? Make sense?

Behind the Scene: Oracle RBO and CBO


TECHNOLOGY: Talking Tuning

As Published In ORACLE MAGAZINE January/February 2005

Understanding Optimization

By Kimberly Floss

Improvements in the Oracle Database 10g Optimizer make it even more valuable for tuning.

Since its introduction in Oracle7, the cost-based optimizer (CBO) has become more valuable and relevant with each new release of the Oracle database while its counterpart, the rule-based optimizer (RBO), has become increasingly less so. The difference between the two optimizers is relatively clear: The CBO chooses the best path for your queries, based on what it knows about your data and by leveraging Oracle database features such as bitmap indexes, function-based indexes, hash joins, index-organized tables, and partitioning, whereas the RBO just follows established rules (heuristics). With the release of Oracle Database 10*g*, the RBO's obsolescence is official and the CBO has been significantly improved yet again.

- Oracle 7 (1992) prior (since 1979): RBO.
- Oracle 7-10: RBO + CBO.
- Oracle 10g (2003): CBO.

Behind the Scene: Oracle RBO and CBO

Rule-based optimization sometimes provided better performance than the early versions of Oracle's cost-based optimizer for specific situations. The rule-based optimizer had several weaknesses, including offering only a simplistic set of rules. The Oracle rule-based optimizer had about 20 rules and assigned a weight to each one of them. In a complex database, a query can easily involve several tables, each with several indexes and complex selection conditions and ordering. This complexity means that there were a lot of options, and the simple set of rules used by the rule-based optimizer might not differentiate the choices well enough to make the best choice.

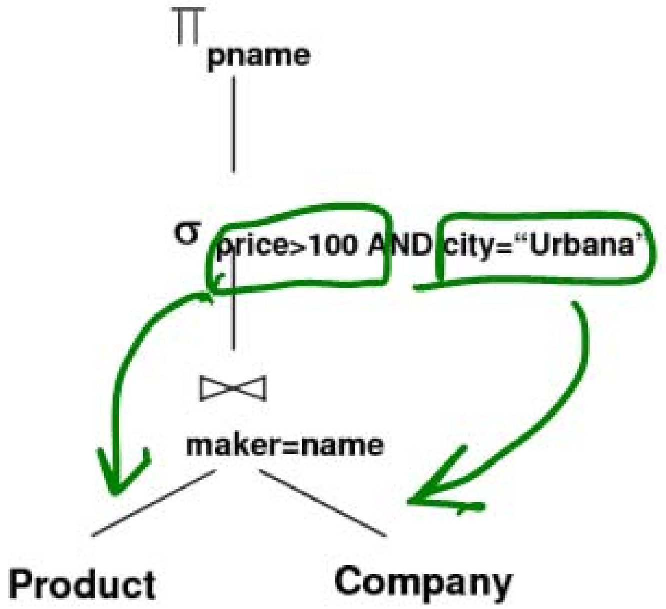
The rule-based optimizer assigned an optimization score to each potential execution path and then took the path with the best optimization score. Another weakness in the rule-based optimizer was resolution of optimization choices made in the event of a "tie" score. When two paths presented the same optimization score, the rule-based optimizer looked to the syntax of the SQL statement to resolve the tie. The winning execution path was based on the order in which the tables occur in the SQL statement.

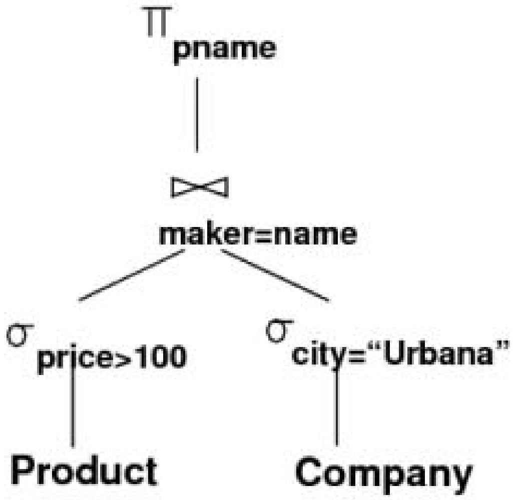
Rule-based Optimization

Heuristic Based Optimizations

- Query rewriting based on algebraic laws
- Result in better queries most of the time
- Heuristics number 1:
  - Push selections down
- Heuristics number 2:
  - Sometimes push selections up, then down

Predicate Pushdown





The earlier we process selections, less tuples we need to manipulate higher up in the tree (but may cause us to loose an important ordering of the tuples, if we use indexes).

min X

Predicate Pushdown

if a group has max(price) > 100 ⟹ then it must come from some prod. w/ price > 100

```
Select   y.name, Max(x.price)
From     product x, company y
Where    x.maker = y.name
GroupBy  y.name
Having   Max(x.price) > 100
```

each company

→ min

```
Select y.name, Max(x.price)
From     product x, company y
Where    x.maker=y.name  and
           x.price > 100
GroupBy   y.name
Having Max(x.price) > 100
```

- *For each company, find the maximal price of its products.*
- Advantage: the size of the join will be smaller.
- Requires transformation rules specific to the grouping/aggregation operators.
- **Won't work if we replace Max by Min.**

Ex: push selectin into views (subqueries)

Pushing predicates up

Bargain view V1: categories with some price<20, and the cheapest price

```
Select   V2.name, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price
```

```
Create View V1 AS
Select   x.category,
         Min(x.price) AS p
From     product x
Where    x.price < 20
GroupBy  x.category
```

```
Create View V2 AS
Select   y.name, x.category, x.price
From     product x, company y
Where    x.maker=y.name
```

Query Rewrite: Pushing predicates up

Bargain view V1: categories with some price<20, and the cheapest price

```
Select   V2.name, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price AND V1.p < 20
```

```
Create View V1 AS
Select   x.category,
           Min(x.price) AS p
From     product x
Where    x.price < 20
GroupBy  x.category
```

```
Create View V2 AS
Select   y.name, x.category, x.price
From     product x, company y
Where    x.maker=y.name
```

Query Rewrite: Pushing predicates up

Bargain view V1: categories with some price<20, and the cheapest price

```
Select   V2.name, V2.price
From     V1, V2
Where    V1.category = V2.category  and
         V1.p = V2.price AND V1.p < 20
```

```
Create View V1 AS
Select   x.category,
         Min(x.price) AS p
From     product x
Where    x.price < 20
GroupBy  x.category
```

```
Create View V2 AS
Select   y.name, x.category, x.price
From     product x, company y
Where    x.maker=y.name
   AND x.price < 20
```

Rule-based Optimization

$\updownarrow$

✓ Cost-based Optimization

1980 0

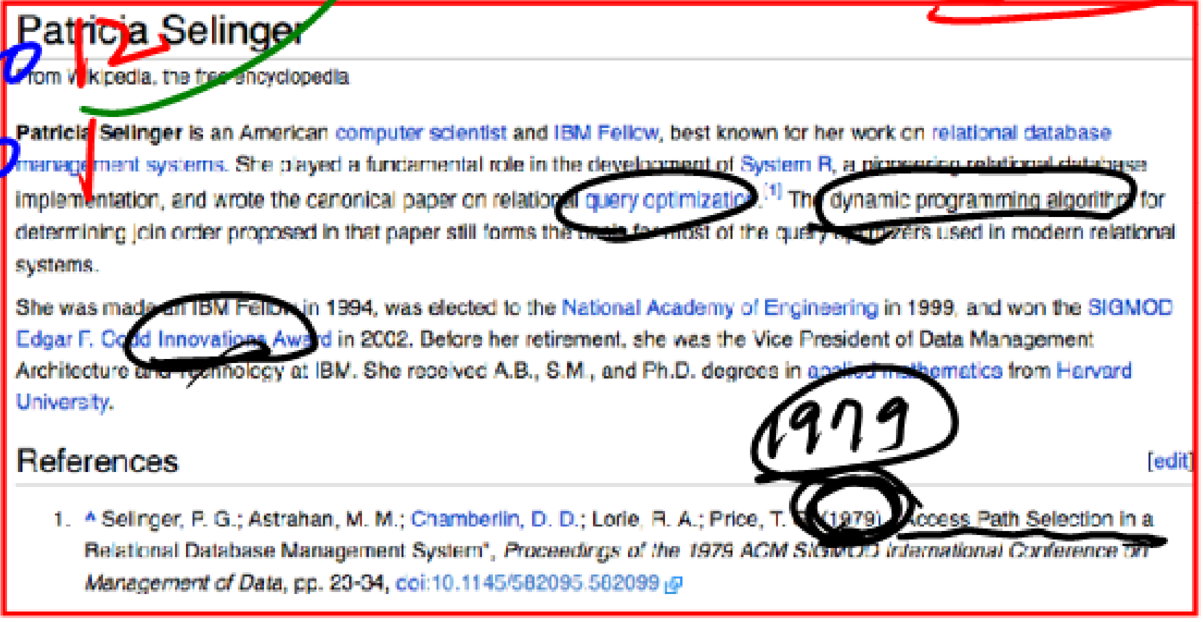
1990 10

2000 12

2010

2003

Behind the Scene: The Selinger Style!



Patricia Selinger

From Wikipedia, the free encyclopedia

Patricia Selinger is an American computer scientist and IBM Fellow, best known for her work on relational database management systems. She played a fundamental role in the development of System R, a pioneering relational database implementation, and wrote the canonical paper on relational query optimization.[1] The dynamic programming algorithm for determining join order proposed in that paper still forms the basis for most of the query optimizers used in modern relational systems.

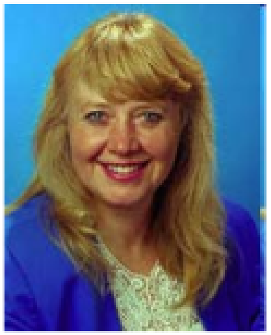
She was made an IBM Fellow in 1994, was elected to the National Academy of Engineering in 1999, and won the SIGMOD Edgar F. Codd Innovations Award in 2002. Before her retirement, she was the Vice President of Data Management Architecture and Technology at IBM. She received A.B., S.M., and Ph.D. degrees in applied mathematics from Harvard University.

1979

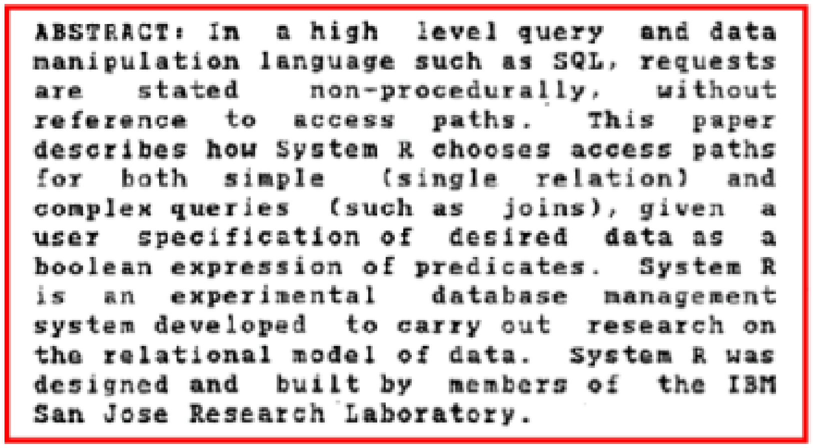
References [edit]

1. ^ Selinger, P. G.; Astrahan, M. M.; Chamberlin, D. D.; Lorie, R. A.; Price, T. G. (1979). "Access Path Selection in a Relational Database Management System", *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pp. 23-34, doi:10.1145/582095.582099

was



is



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.

Behind the Scene: The Selinger Style!

In my view, the query optimizer was the first attempt at what we call autonomic computing or self-managing, self-timing technology. Query optimizers have been 25 years in development, with enhancements of the cost-based query model and the optimization that goes with it, and a richer and richer variety of execution techniques that the optimizer chooses from. We just have to keep working on this. It's a never-ending quest for an increasingly better model and repertoire of optimization and execution techniques. So the more the model can predict what's really happening in the data and how the data is really organized, the closer and closer we will come [to the ideal system].

In hindsight, I didn't really have enough experience as a system designer and developer to realize that System R was going to be something that people [within IBM] were going to take almost as is, and make a productized copy of it and [sell] it. So designing control blocks, designing interfaces and APIs to programming languages---we just sort of put together something that we thought would work. The error control block, the way that you pass variables and parameters into the system---those were not designed with thought and care and elegance. They were [just]

Cost-based Optimizations

- Main idea: apply algebraic laws, until estimated cost is minimal   search $\Rightarrow$ construction
- Practically: start from partial plans, introduce operators one by one
  - Will see in a few slides
- Problem: there are too many ways to apply the laws, hence too many (partial) plans

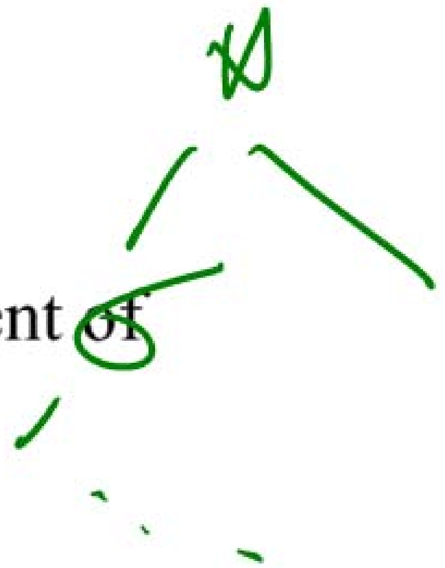
Cost-based Optimizations

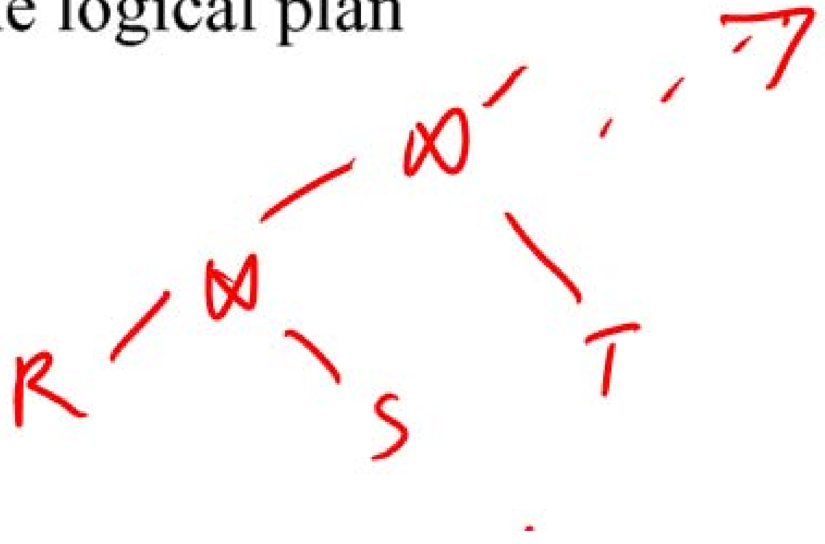
Approaches:

- **Top-down**: the partial plan is a top fragment of the logical plan

D. P.

- **Bottom up**: the partial plan is a bottom fragment of the logical plan



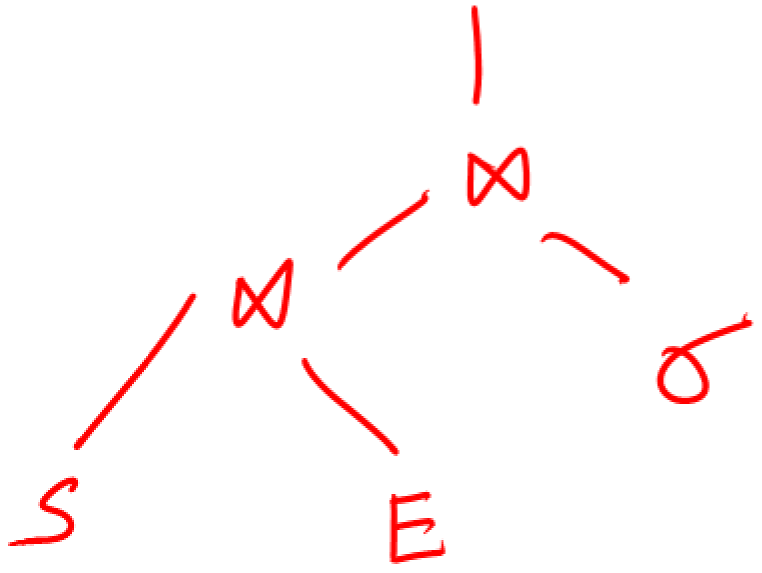


Search Strategies

- **Branch-and-bound**:
  - Remember the cheapest complete plan P seen so far and its cost C
  - Stop generating partial plans whose cost is > C
  - If a cheaper complete plan is found, replace P, C
- **Hill climbing**:
  - Remember only the cheapest partial plan seen so far
- **Dynamic programming**: bottom-up
  - Remember the all cheapest partial plans

Dynamic Programming

Unit of Optimization: $\underbrace{\text{select-project-join}}$



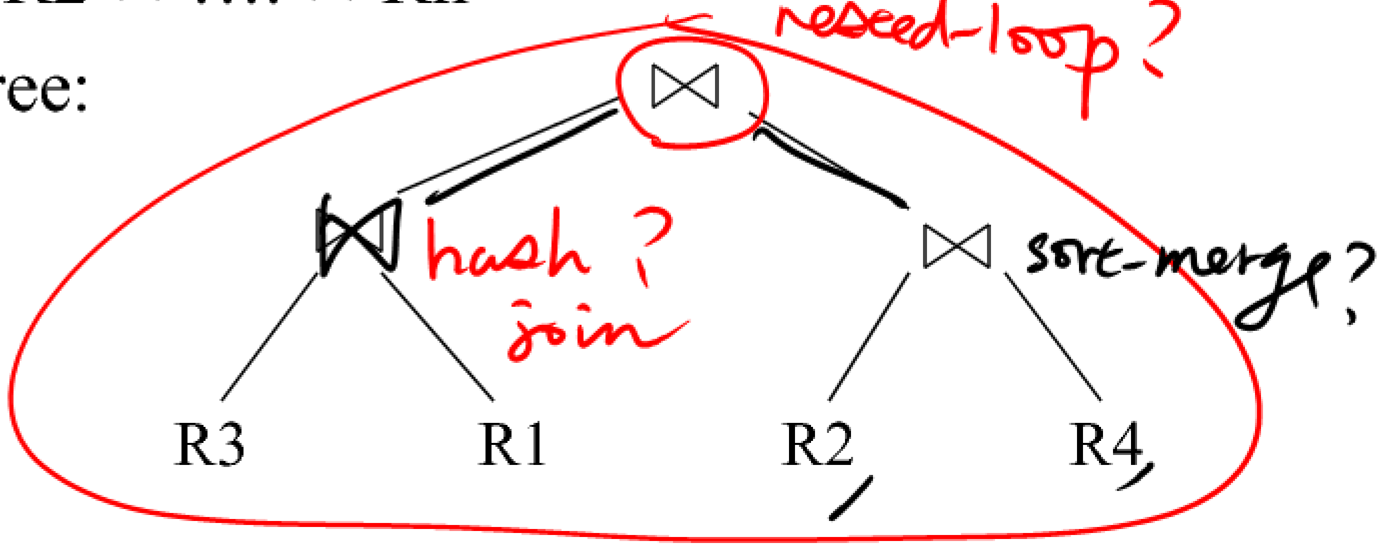
$R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$

Assumption:

join is binary op

Join Trees

- R1 ⋈ R2 ⋈ …. ⋈ Rn
- Join tree:



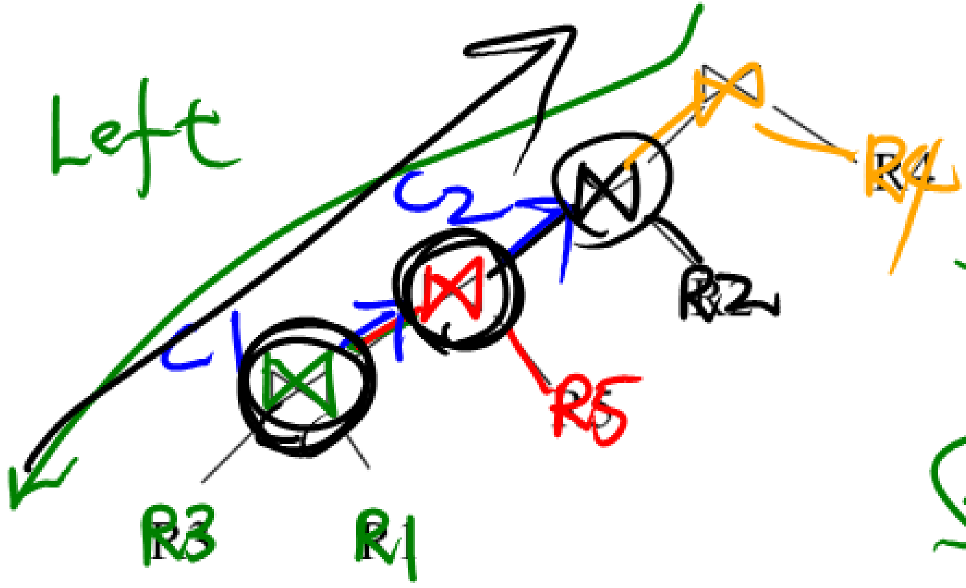
- A plan = a join tree
- A partial plan = a subtree of a join tree

Types of Join Trees

Assump :
only care about left-deep

- Left deep:

iterator



benefits :

① Easily pipelined,

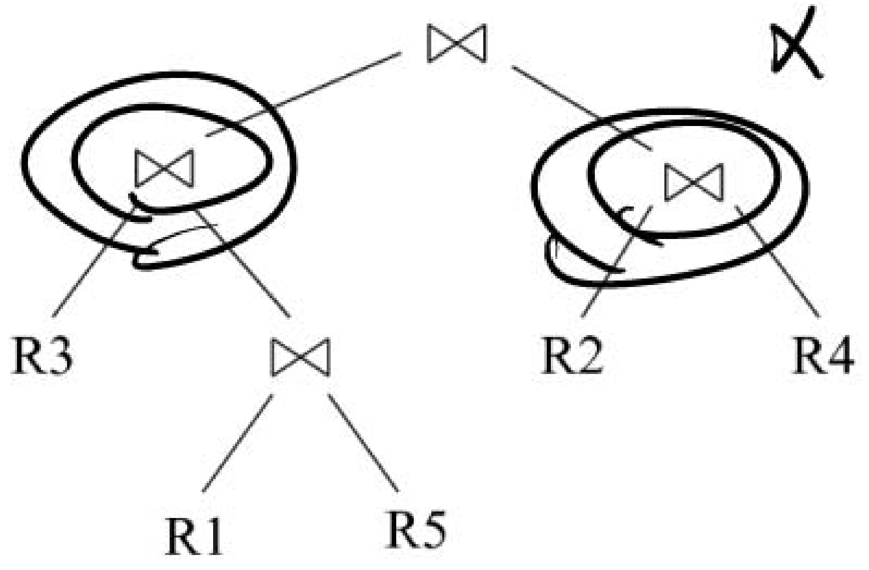
② mem req small

Types of Join Trees

- Bushy: ✓ ✗

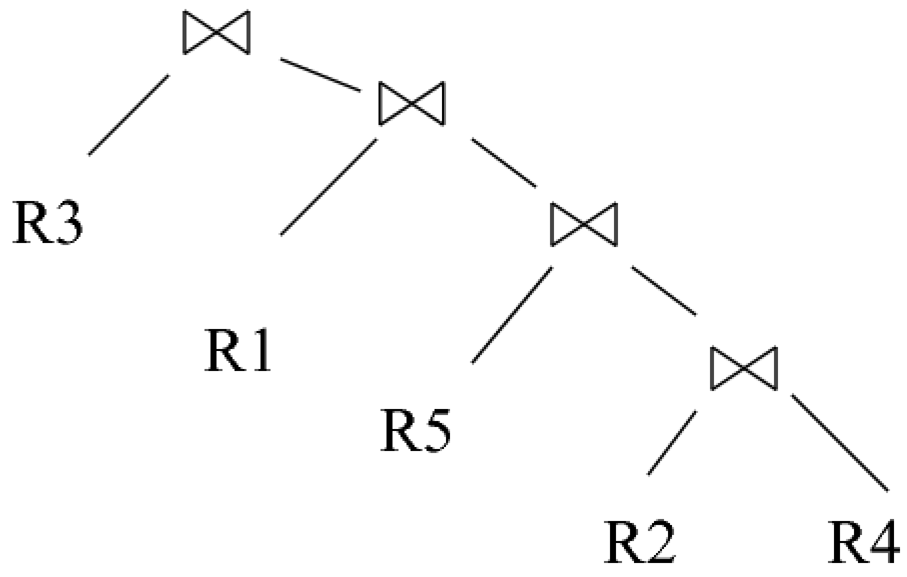
× pipeline not easy

× mem req high



Types of Join Trees

- Right deep: $\leftrightarrow$ Left deep



Problem

- Given: a query $\boxed{R1 \bowtie R2 \bowtie \ldots \bowtie Rn}$
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

- Idea: for each subset of {R1, …, Rn}, compute the best plan for that subset
- In increasing order of set cardinality:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}
- It is a bottom-up strategy
- A subset of {R1, …, Rn} is also called a *subquery*

Dynamic Programming

- For each subquery Q ⊆ {R1, …, Rn} compute the following:
  - Size(Q)
  - A best plan for Q: Plan(Q)
  - The cost of that plan: Cost(Q)

Dynamic Programming

- **Step 1**: For each {Ri} do:
  - Size({Ri}) = B(Ri)
  - Plan({Ri}) = Ri
  - Cost({Ri}) = (cost of scanning Ri)

Dynamic Programming

- **Step i**: For each $Q \subseteq \{R1, \ldots, Rn\}$ of cardinality i do:
  - Compute Size(Q)   (later…)
  - For every pair of subqueries Q', Q''
    s.t. $Q = Q' \cup Q''$
    compute cost(Plan(Q') $\bowtie$ Plan(Q''))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan({R1, …, Rn})

Dynamic Programming

To illustrate, we will make the following simplifications:

- Cost(P1 $\bowtie$ P2) = Cost(P1) + Cost(P2) + size(intermediate result
- Intermediate results:
  - If P1 = a join, then the size of the intermediate result is size(P1), otherwise the size is 0
  - Similarly for P2
- Cost of a scan = 0

Dynamic Programming

- Example:
- Cost(R5 $\bowtie$ R7) = 0 (no intermediate results)
- Cost((R2 $\bowtie$ R1) $\bowtie$ R7)

  = Cost(R2 $\bowtie$ R1) + Cost(R7) + size(R2 $\bowtie$ R1)

  = size(R2 $\bowtie$ R1)

Dynamic Programming

- Relations: R, S, T, U
- Number of tuples: 2000, 5000, 3000, 1000
- Size estimation: $T(A \bowtie B) = 0.01*T(A)*T(B)$