Recap

- Data-savviness is the future!
- Notion of a DBMS
- The relational data model and algebra: bags and sets
- SQL Queries
- SQL Modifications
- SQL DDL
- Database Design
- Views, constraints and triggers
- Indexes
- Query processing
- Next: query optimization

Recap: Given a SQL query Q

- How should Q be executed by the DBMS?
- What do we (the system) know?
  - We know Q, therefore we know the relations it is operating on, the predicates, the grouping and aggregating attributes
  - We know indexes on the relations, the sizes of the relations, statistics about the relations (# of columns and distinct values)…
- Goal: come up with a query execution plan:
  - We think of plans at two levels
    - **The logical level: at the level of operators** $\sigma, \Pi, \bowtie$
    - The physical level: specific implementations

Recap: Logical vs. Physical Operators

- Logical operators are relational algebra (RA) operators
    - Describe "what" is done
    - e.g., union, select, grouping, project
        - We covered only relational algebra operators for the basic operators, but there are operators for grouping and sorting as well
- Physical operators describe implementations of these operators
    - Describe "how" to do it
    - e.g., for join
        - nested-loop, sort-merge, hash join, …
    - Physical operators also pertain to non-RA operators such as scanning a table

OK…

- So we talked about physical implementations. Now how do we use them?
- Let's talk about the heart of the database engine
- Step 1: convert the SQL query to a logical query plan
- Step 2: apply rewriting to find other equivalent logical plans
- Step 3: use "optimization" pick among the logical plans, and pick the corresponding physical plan
- Step 4: feed the corresponding plan to the query processor

Converting a query to a logical query plan

- A logical query plan is simply an (extended) relational algebra expression
- We already know how to do this

SELECT a1, a2, …, aggs
FROM R1, … Rk
WHERE C
GROUP BY b1, …, bm
HAVING H

$$\Pi_{a1,a2,\ldots,an,aggs}(\sigma_H(\gamma_{b1,b2,\ldots,bm,aggs}(\sigma_C(R1 \times \ldots \times Rk))))$$

Usually Joins

Subqueries do not cause problems!

SELECT DISTINCT Product.name FROM Product

WHERE Product.maker IN (SELECT Company.Name FROM Company
WHERE Company.city = “Berkeley”)

Q: Can we rewrite without using a subquery?

SELECT DISTINCT Product.name FROM Product, Company

WHERE Product.maker = Company.name AND Company.city = “Berkeley”

Step 2: Rewriting the Logical Plan

- To find equivalent rewriting, we need algebraic laws that allow us to manipulate relational algebra expressions

- Let's focus on the **set** case (but the bag case is similar)
- Commutative, associative, and distributive laws, like:

$$R \cup S = S \cup R, R \cup (S \cup T) = (R \cup S) \cup T$$

$$R \bowtie S = S \bowtie R, R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$$

$$R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$$

Laws Involving Selection: Examples

$$\sigma_{C\ AND\ C'}(R) = \sigma_C(\sigma_{C'}(R))$$

$$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$$

$$\sigma_C(R \bowtie S) = \sigma_D(\sigma_E(R) \bowtie \sigma_F(S))$$

A very special rule called **predicate pushdown**
Q: What are D, E, F?

D: predicates involving R and S;
E (F): predicates involving R (S) attrib only

Laws Involving Selection: Examples

- R (A, B, C, D), S (E, F, G)
- Simplify as much as possible by predicate pushdown

$$\sigma_{F=3}(R \bowtie_{D=E} S) =$$

$$\sigma_{A=5\ AND\ G=9}(R \bowtie_{D=E} S) =$$

- The earlier we process selections/predicates, the less we need to manipulate later on, so is usually a good thing!

Laws Involving Projection

- Similar to selection (including a projection pushdown)

$$\Pi_M(\Pi_N(R)) = \Pi_M(R)$$

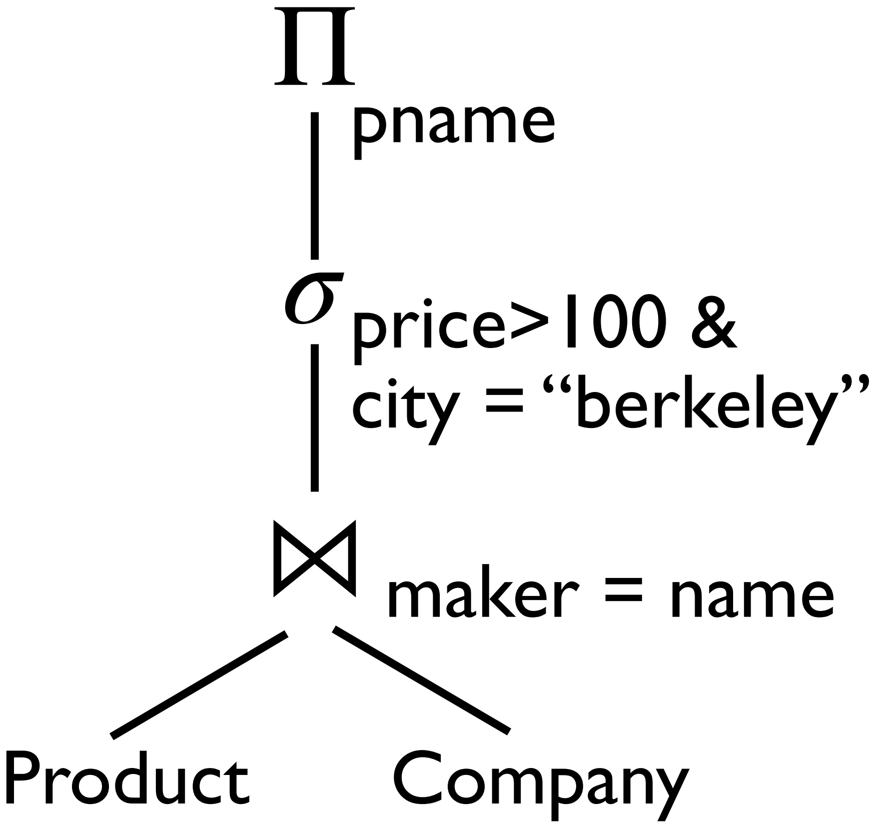
$$\Pi_M(R \bowtie S) = \Pi_N(\Pi_P(R) \bowtie \Pi_Q(S))$$

- Q: What should N, P, Q be?

$$R(A, B, C, D), S(E, F, G)$$

$$\Pi_{A,B,G}(R \bowtie_{D=E} S) = \Pi_?(\Pi_?(R) \bowtie_{D=E} \Pi_?(S))$$

How to use these rules



Product (maker, price, pname, category)

Company (name, city, owner, marketcap)

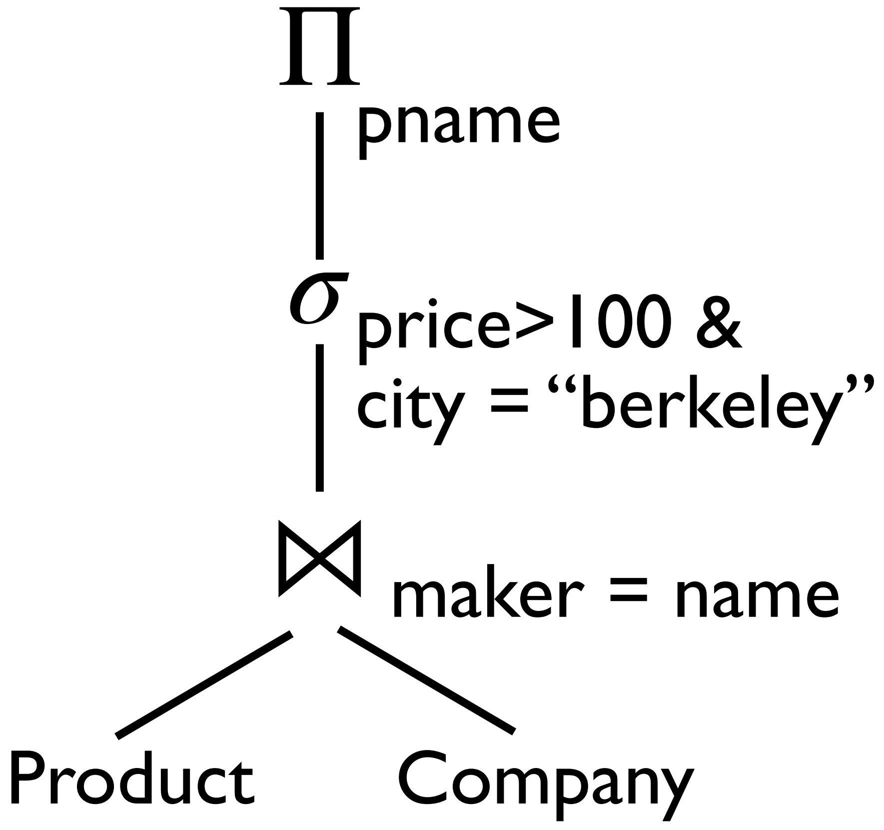
Query plans (RA exps) also depicted as trees

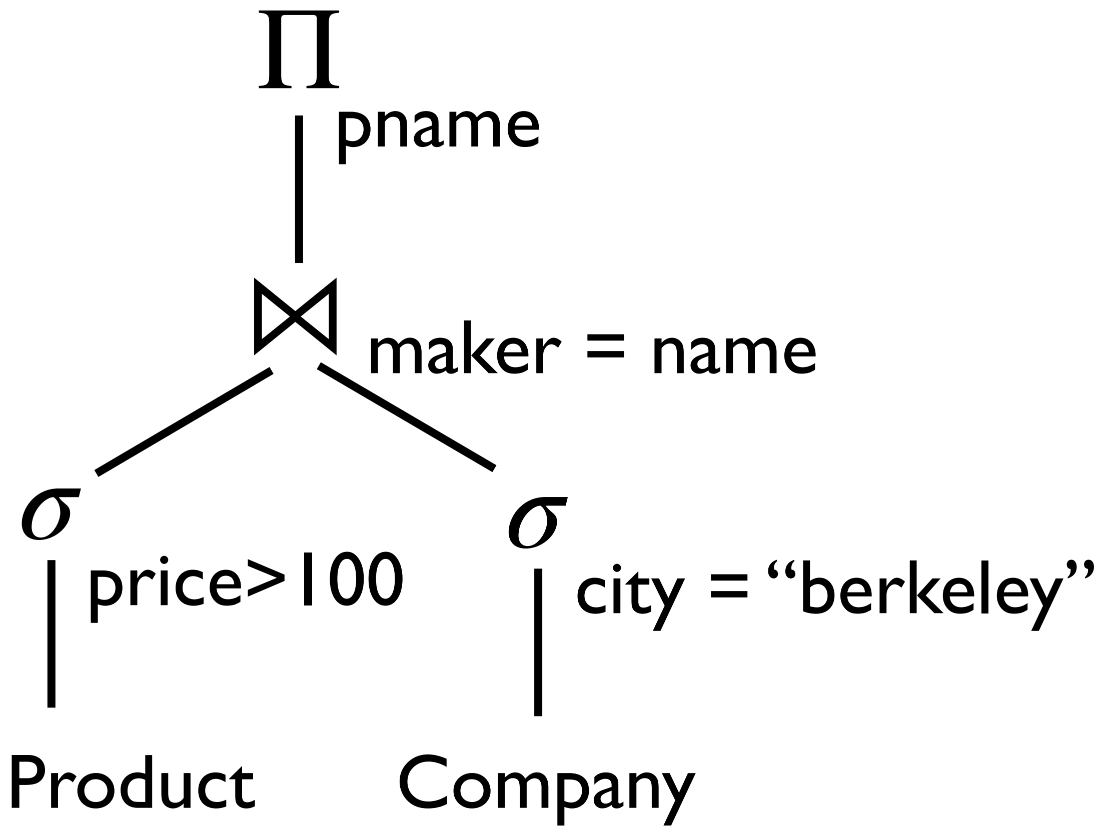
Q: What does this query evaluate to?

Q: Can we push the predicates down?

How to use these rules

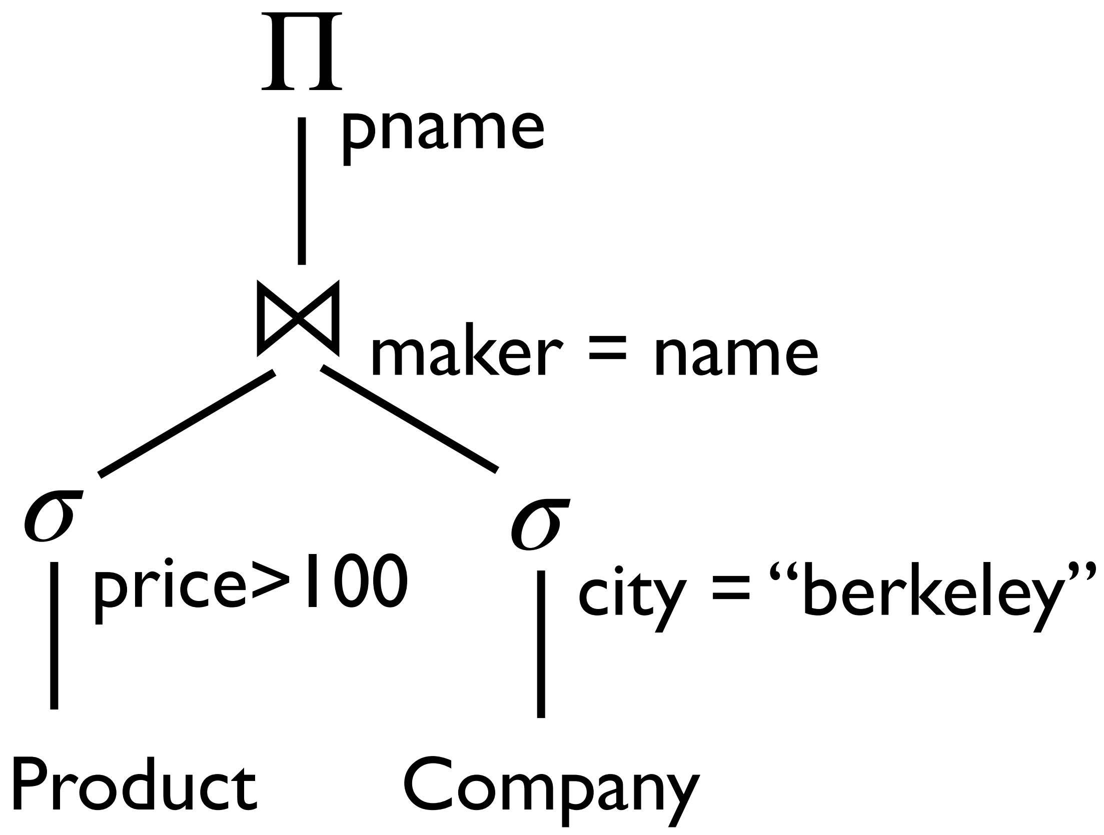
- Product (maker, price, pname, category)
- Company (name, city, owner, marketcap)





How to use these rules

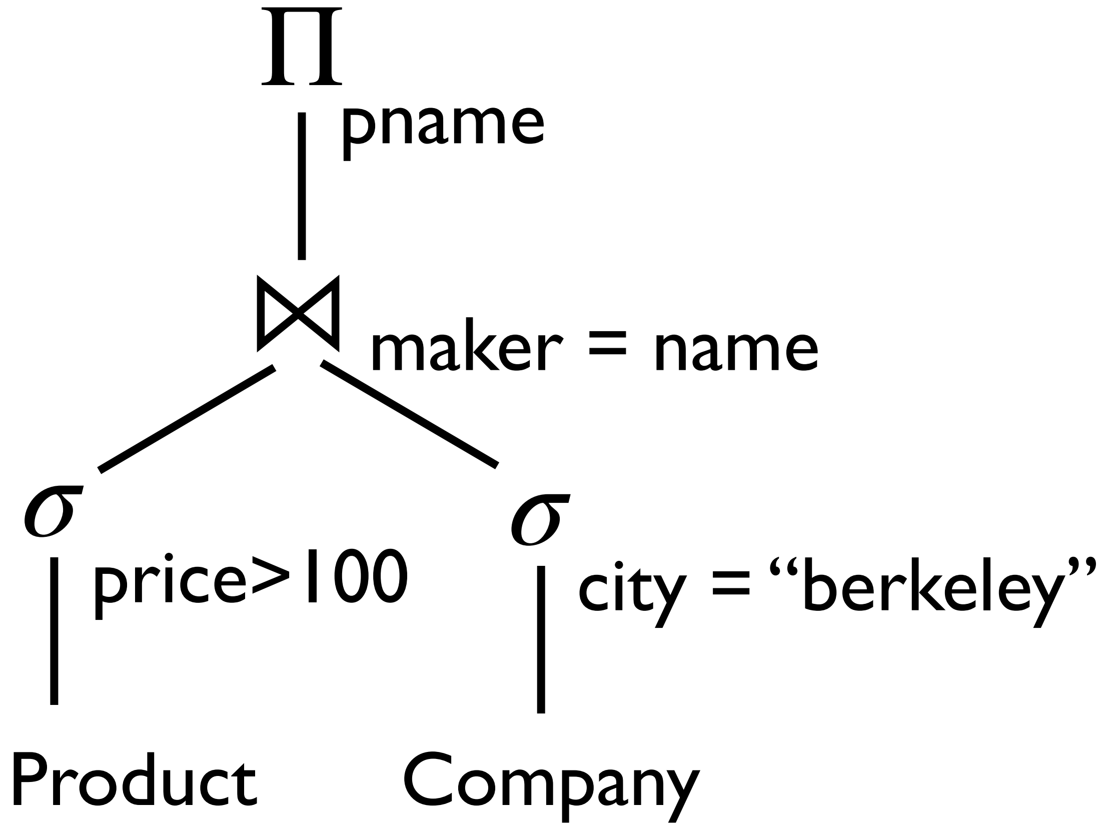
- Product (maker, price, pname, category)
- Company (name, city, owner, marketcap)

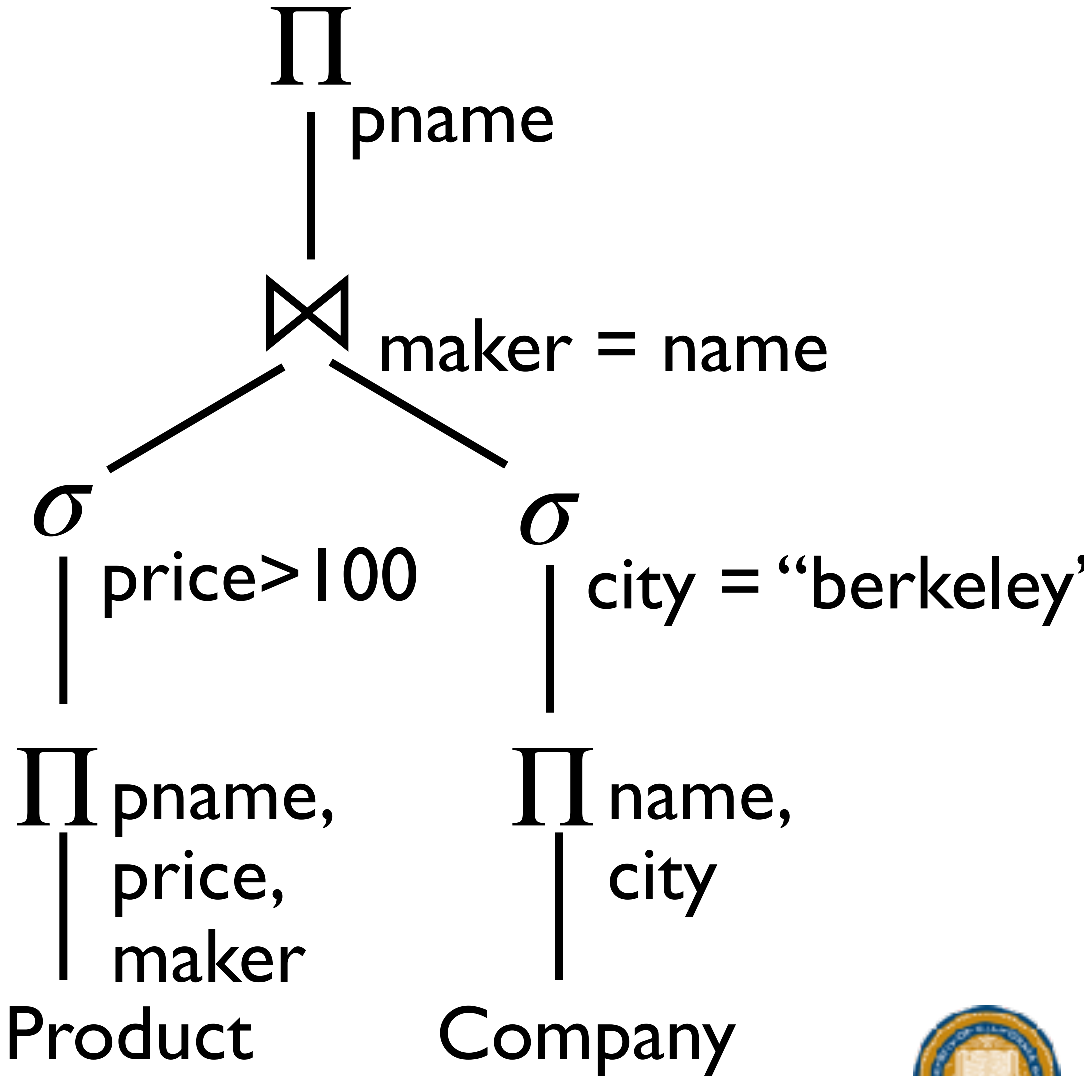


Q: can we push projections down?

How to use these rules

- Product (maker, price, pname, category)
- Company (name, city, owner, marketcap)





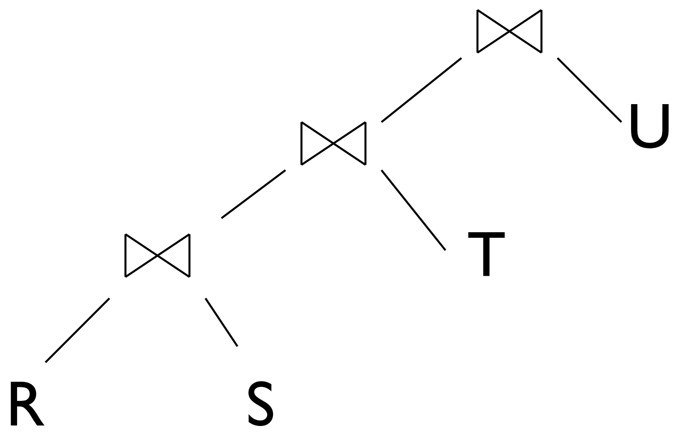
OK…

- So we talked about physical implementations. Now how do we use them?
- Let's talk about the heart of the database engine
- Step 1: convert the SQL query to a logical query plan
- Step 2: apply rewriting to find other equivalent logical plans
- **Step 3: use "optimization" pick among the logical plans, and pick the corresponding physical plan**
- Step 4: feed the corresponding plan to the query processor

Optimization

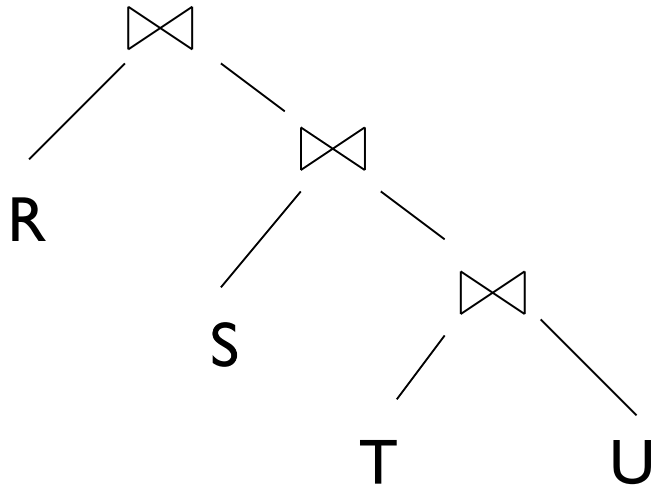
- Usually involves applying some heuristic rules that typically improve plans, e.g., predicate/selection pushdown
- Usually a dynamic programming algorithm that figures out the best order of joins
  - Joins are the hardest part! (More next…)
  - Called the "Selinger" algorithm after Pat Selinger at IBM
  - One of the crown jewels of database systems
- (Top-down approaches also possible: see Cascade query optimizer)
- The query optimizer estimates the cost across plans and picks the plan with the lowest cost
  - The cost is often inaccurate, since it is done based on coarse-grained statistics

Lots of join orders and trees!

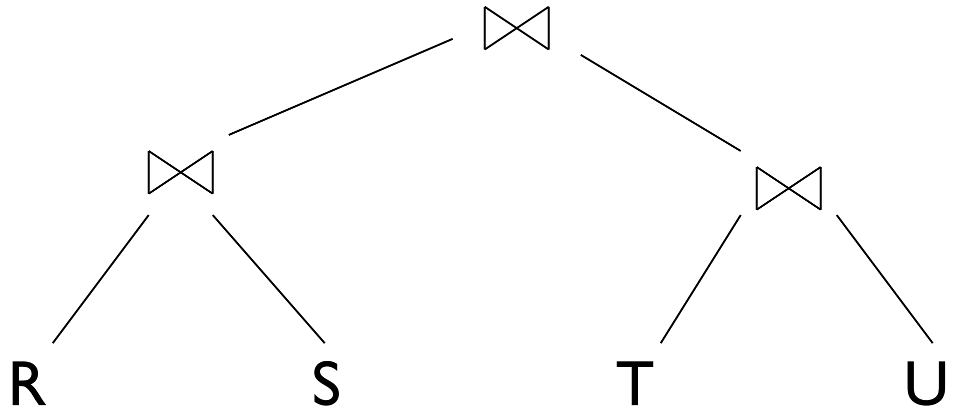


Left deep

Plus n! orders



Right deep



Bushy

Lots of Bad join orders

- Student (StudentID, Name, DOB) w/ 1M tuples
- StudentMajor (StudentID, MajorID) w/ 1.5M tuples
- Major (MajorID, Name, Department) w/ 1000 tuples
- Say we want to do the natural join across these three relations
  - Assume FK from StudentMajor to Student and Major
  - What will be the size of the join result?
- In what order should we do this join?
- Why is joining Student and Major first a BAD idea?

Lots of Bad join orders

- Student (StudentID, Name, DOB) w/ 1M tuples
- StudentMajor (StudentID, MajorID) w/ 1.5M tuples
- Major (MajorID, Name, Department) w/ 1000 tuples
- Say we want to do the natural join across these three relations
  - Assume FK from StudentMajor to Student and Major

- With Joins there is a real danger of having GIANT intermediate relations, especially if it is k-way join
- The join order really matters to make sure this doesn't blow up in our face

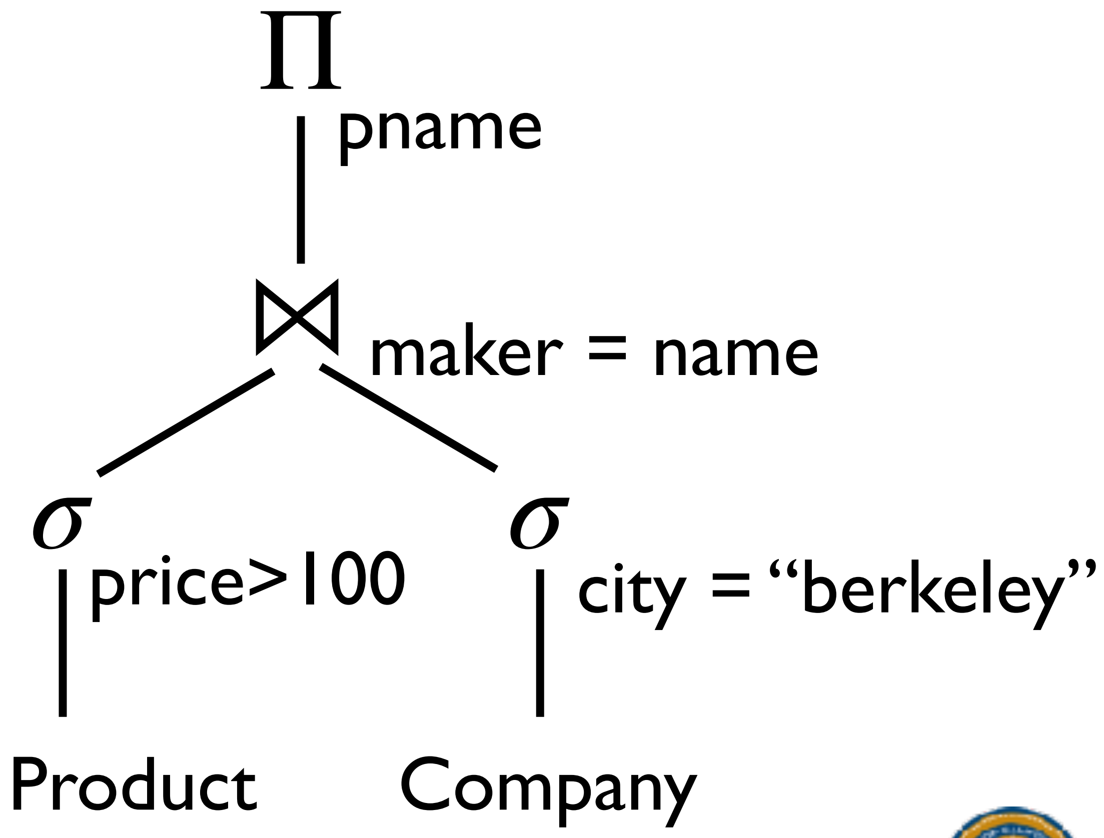
Do other operators have the same problem?

- Join/cross-product is the only operator that "multiplies"

- Not true for
  - Selection, Projection (both reduce size)
  - Union, Difference (grow additively)
  - Sorting, Grouping, Aggregation (all reduce size, but require more work than Selection and Projection)

What Does Optimization Give Us?

- A physical query plan
  - A sequence or workflow of physical operators
    - Scans: sequential or indexed
    - Joins: hash/sort-merge/NL
  - Whether intermediate results are “pipelined” or materialized
    - Pipelining means operators are doing work in parallel
  - Each operator itself could also be “parallelized” (partitioned)



Why Do We Care?

- As users or administrators of database systems, we need to understand enough of what is going on under the covers
- SQL queries are rewritten into logical query plans (RA expressions)
  - Algebraic rules allow us to manipulate these logical q plans
- Optimization allows us to pick the best logical query plan, and best corresponding physical query plan
- Rules of thumb:
  - Joins are expensive: the main focus of many query optimizers
  - Reducing intermediate results can help!
    - Do joins in the right order
    - Pushing predicates/projections down
    - Using an index
  - Parallelism, pipelined or partitioned can help