# Query Optimization

Introduction to Databases

CompSci 316 Fall 2015



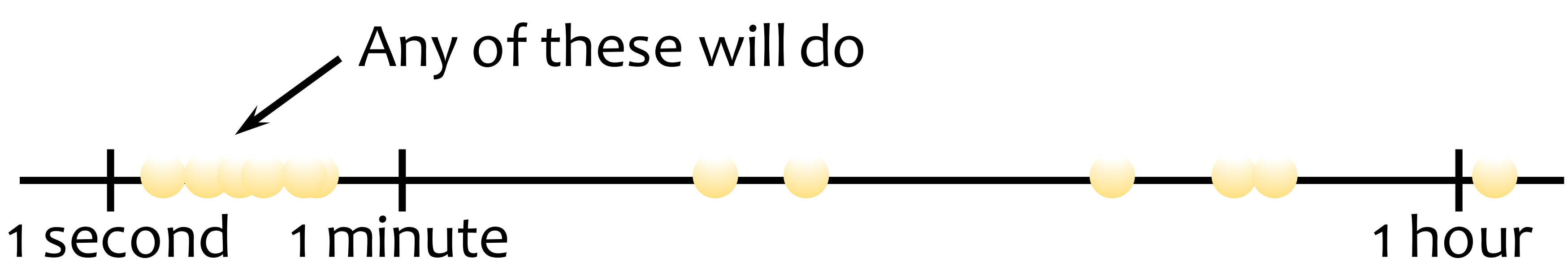
DUKE COMPUTER SCIENCE

# Announcements (Thu., Nov. 19)

- Homework #4 due on 12/01
- Project demos 12/3-12/9
  - Sign-ups to begin next week

# Query optimization

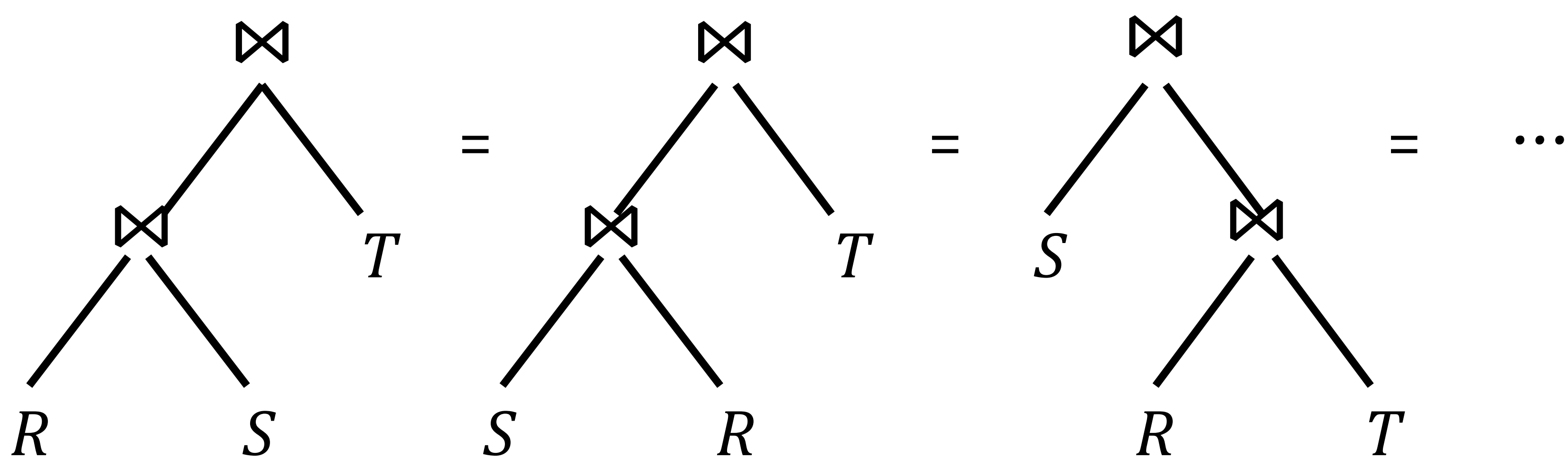
- One logical plan → “best” physical plan
- Questions
  - How to enumerate possible plans
  - How to estimate costs
  - How to pick the “best” one
- Often the goal is not getting the optimum plan, but instead avoiding the horrible ones



# Plan enumeration in relational algebra

- Apply relational algebra equivalences

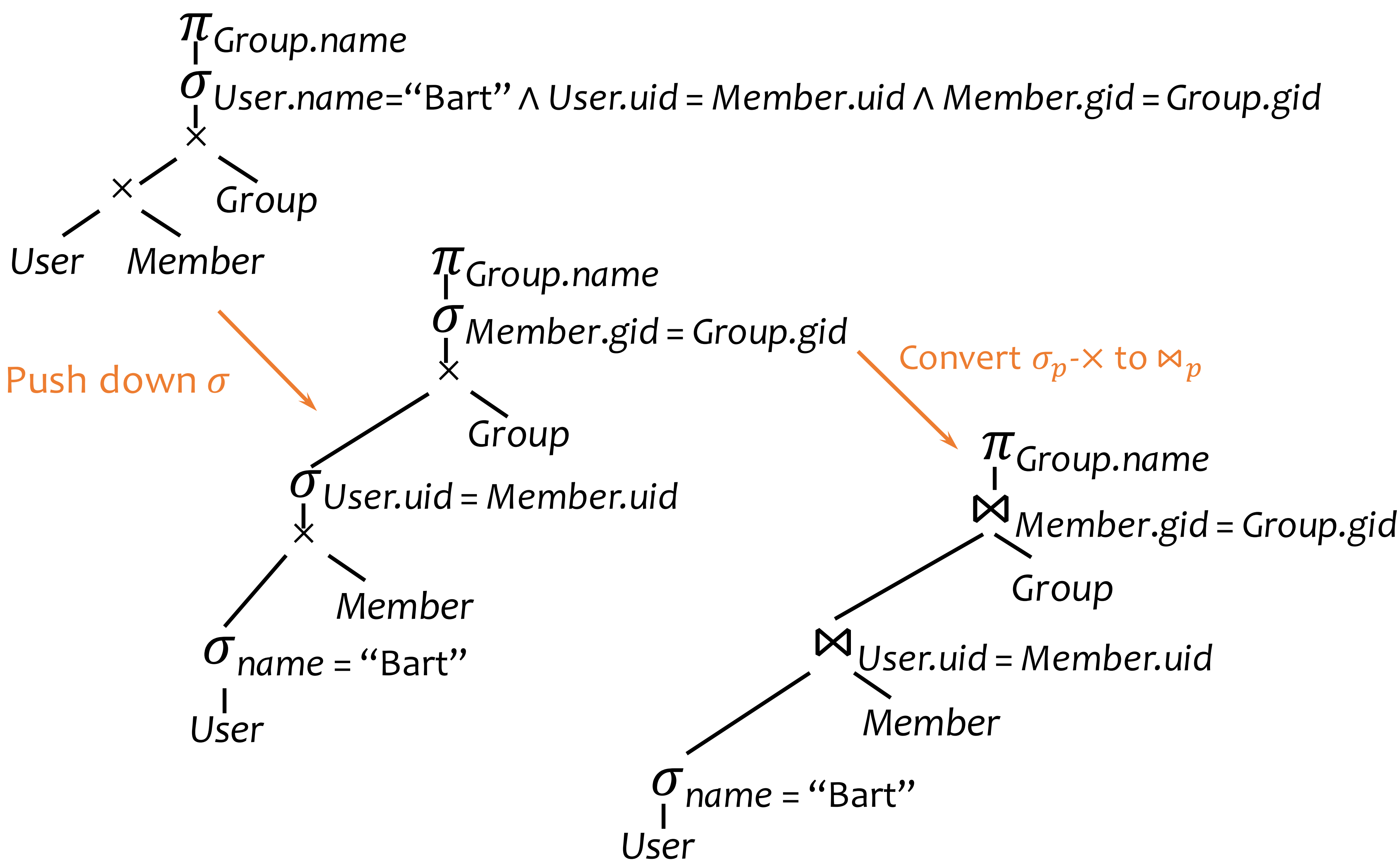
☞ Join reordering: × and ⋈ are associative and commutative (except column ordering, but that is unimportant)



# More relational algebra equivalences

- Convert $\sigma_p$-$\times$ to/from $\bowtie_p$: $\sigma_p(R \times S) = R \bowtie_p S$
- Merge/split $\sigma$'s: $\sigma_{p_1}\left(\sigma_{p_2} R\right) = \sigma_{p_1 \wedge p_2} R$
- Merge/split $\pi$'s: $\pi_{L_1}\left(\pi_{L_2} R\right) = \pi_{L_1} R$, where $L_1 \subseteq L_2$
- Push down/pull up $\sigma$:
  $\sigma_{p \wedge p_r \wedge p_s}\left(R \bowtie_{p'} S\right) = \left(\sigma_{p_r} R\right) \bowtie_{p \wedge p'} \left(\sigma_{p_s} S\right)$, where
  - $p_r$ is a predicate involving only $R$ columns
  - $p_s$ is a predicate involving only $S$ columns
  - $p$ and $p'$ are predicates involving both $R$ and $S$ columns
- Push down $\pi$: $\pi_L\left(\sigma_p R\right) = \pi_L\left(\sigma_p\left(\pi_{LL'} R\right)\right)$, where
  - $L'$ is the set of columns referenced by $p$ that are not in $L$
- Many more (seemingly trivial) equivalences…
  - Can be systematically used to transform a plan to new ones

# Relational query rewrite example



# Heuristics-based query optimization

- Start with a logical plan
- Push selections/projections down as much as possible
  - Why?
  - Why not?
- Join smaller relations first, and avoid cross product
  - Why?
  - Why not?
- Convert the transformed logical plan to a physical plan (by choosing appropriate physical operators)

# SQL query rewrite

- More complicated—subqueries and views divide a query into nested "blocks"
  - Processing each block separately forces particular join methods and join order
  - Even if the plan is optimal for each block, it may not be optimal for the entire query
- Unnest query: convert subqueries/views to joins

☞We can just deal with select-project-join queries

  - Where the clean rules of relational algebra apply

# SQL query rewrite example

- ```
  SELECT name
  FROM User
  WHERE uid = ANY (SELECT uid FROM Member);
  ```
- ```
  SELECT name
  FROM User, Member
  WHERE User.uid = Member.uid;
  ```
- ```
  SELECT name
  FROM (SELECT DISTINCT User.uid, name
        FROM User, Member
        WHERE User.uid = Member.uid);
  ```
  - Right—assuming `User.uid` is a key
  ```

# Dealing with correlated subqueries

- ```
  SELECT gid FROM Group
  WHERE name LIKE 'Springfield%'
  AND min_size > (SELECT COUNT(*) FROM Member
                  WHERE Member.gid = Group.gid);
  ```
- ```
  SELECT gid
  FROM Group, (SELECT gid, COUNT(*) AS cnt
               FROM Member GROUP BY gid) t
  WHERE t.gid = Group.gid AND min_size > t.cnt
  AND name LIKE 'Springfield%’;
  ```

# "Magic" decorrelation

- ```
  SELECT gid FROM Group
  WHERE name LIKE 'Springfield%'
  AND min_size > (SELECT COUNT(*) FROM Member
                  WHERE Member.gid = Group.gid);
  ```
- ```
  WITH Supp_Group AS
  (SELECT * FROM Group WHERE name LIKE 'Springfield%'),
  ```
  Process the outer query without the subquery
  ```
  Magic AS
  (SELECT DISTINCT gid FROM Supp_Group),
  ```
  Collect bindings
  ```
  DS AS
  ((SELECT Group.gid, COUNT(*) AS cnt
    FROM Magic, Member WHERE Magic.gid = Member.gid
    GROUP BY Member.gid) UNION
   (SELECT gid, 0 AS cnt
    FROM Magic WHERE gid NOT IN (SELECT gid FROM Member)))
  ```
  Evaluate the subquery with bindings
  ```
  SELECT Supp_Group.gid FROM Supp_Group, DS
  WHERE Supp_Group.gid = DS.gid
  AND min_size > DS.cnt;
  ```
  Finally, refine the outer query

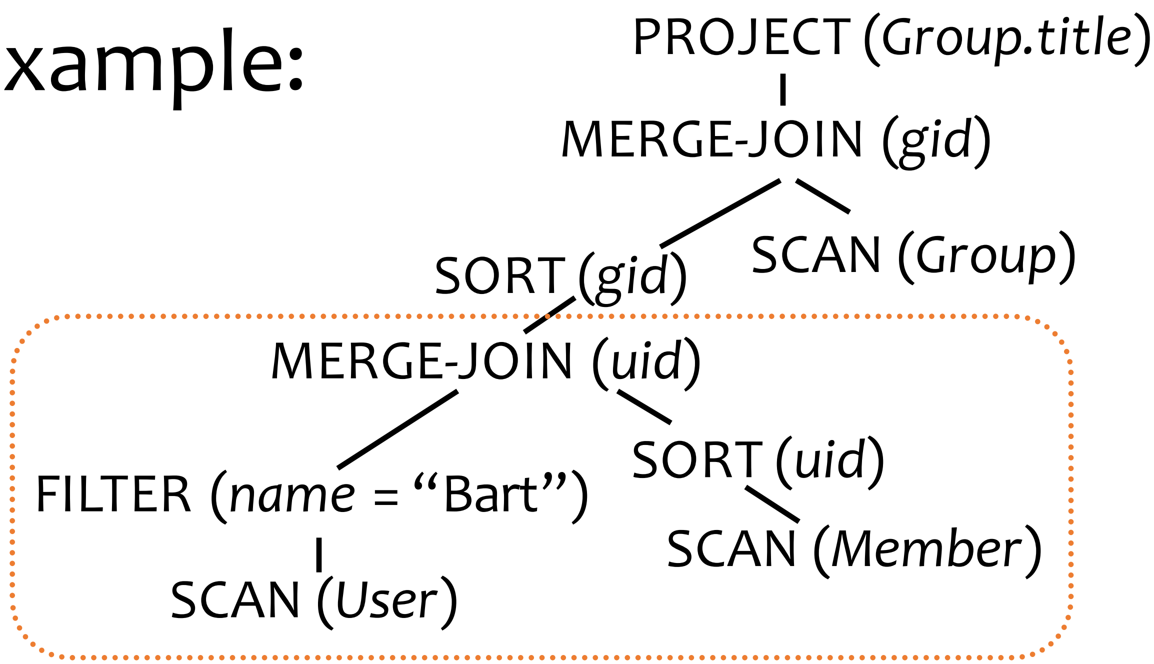
# Heuristics- vs. cost-based optimization

- Heuristics-based optimization
  - Apply heuristics to rewrite plans into cheaper ones
- Cost-based optimization
  - Rewrite logical plan to combine “blocks” as much as possible
  - Optimize query block by block
    - Enumerate logical plans (already covered)
    - Estimate the cost of plans
    - Pick a plan with acceptable cost
  - Focus: select-project-join blocks

# Cost estimation

## Physical plan example:

Input to SORT(*gid*):




- We have: cost estimation for each operator
  - Example: SORT(*gid*) takes $O(B(\text{input}) \times \log_M B(\text{input}))$
    - But what is $B(\text{input})$?
- We need: size of intermediate results

# Cardinality estimation


http://www.learningresources.com/product/estimation+station.do

# Selections with equality predicates

- $Q$: $\sigma_{A=v}R$
- Suppose the following information is available
  - Size of $R$: $|R|$
  - Number of distinct $A$ values in $R$: $|\pi_A R|$
- Assumptions
  - Values of $A$ are uniformly distributed in $R$
  - Values of $v$ in $Q$ are uniformly distributed over all $R.A$ values
- $|Q| \approx {}^{|R|}/_{|\pi_A R|}$
  - Selectivity factor of $(A = v)$ is ${}^{1}/_{|\pi_A R|}$

# Conjunctive predicates

- $Q$: $\sigma_{A=u \wedge B=v} R$
- Additional assumptions
  - $(A = u)$ and $(B = v)$ are independent
    - Counterexample: major and advisor
  - No "over"-selection
    - Counterexample: $A$ is the key
- $|Q| \approx {}^{|R|}/_{|\pi_A R| \cdot |\pi_B R|}$
  - Reduce total size by all selectivity factors

# Negated and disjunctive predicates

- $Q$: $\sigma_{A \neq v} R$
  - $|Q| \approx |R| \cdot \left(1 - {}^{1}/_{|\pi_A R|}\right)$
    - Selectivity factor of $\neg p$ is $(1 -$ selectivity factor of $p)$
- $Q$: $\sigma_{A=u \,\vee\, B=v} R$
  - $|Q| \approx |R| \cdot \left({}^{1}/_{|\pi_A R|} + {}^{1}/_{|\pi_B R|}\right)$?
  - $|Q| \approx |R| \cdot \left({}^{1}/_{|\pi_A R|} + {}^{1}/_{|\pi_B R|} - {}^{1}/_{|\pi_A R||\pi_B R|}\right)$
    - Inclusion-exclusion principle

# Range predicates

- $Q$: $\sigma_{A>v} R$
- Not enough information!
  - Just pick, say, $|Q| \approx |R| \cdot {}^{1}/_{3}$
- With more information
  - Largest *R.A* value: $\text{high}(R.A)$
  - Smallest *R.A* value: $\text{low}(R.A)$
  - $|Q| \approx |R| \cdot \frac{\text{high}(R.A) - v}{\text{high}(R.A) - \text{low}(R.A)}$
  - In practice: sometimes the second highest and lowest are used instead

# Two-way equi-join

- $Q$: $R(A,B) \bowtie S(A,C)$
- Assumption: containment of value sets
  - Every tuple in the "smaller" relation (one with fewer distinct values for the join attribute) joins with some tuple in the other relation
  - That is, if $|\pi_A R| \leq |\pi_A S|$ then $\pi_A R \subseteq \pi_A S$
  - Certainly not true in general
  - But holds in the common case of foreign key joins
- $|Q| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)}$
  - Selectivity factor of $R.A = S.A$ is ${}^{1}/_{\max(|\pi_A R|, |\pi_A S|)}$

# Multiway equi-join

- $Q$: $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$
- What is the number of distinct $C$ values in the join of $R$ and $S$?
- Assumption: preservation of value sets
  - A non-join attribute does not lose values from its set of possible values
  - That is, if $A$ is in $R$ but not $S$, then $\pi_A(R \bowtie S) = \pi_A R$
  - Certainly not true in general
  - But holds in the common case of foreign key joins (for value sets from the referencing table)

# Multiway equi-join (cont'd)

- $Q$: $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$
- Start with the product of relation sizes
  - $|R| \cdot |S| \cdot |T|$
- Reduce the total size by the selectivity factor of each join predicate
  - $R.B = S.B$: $^1/_{\max(|\pi_B R|, |\pi_B S|)}$
  - $S.C = T.C$: $^1/_{\max(|\pi_C S|, |\pi_C T|)}$
  - $|Q| \approx \frac{|R| \cdot |S| \cdot |T|}{\max(|\pi_B R|, |\pi_B S|) \cdot \max(|\pi_C S|, |\pi_C T|)}$

# Cost estimation: summary

- Using similar ideas, we can estimate the size of projection, duplicate elimination, union, difference, aggregation (with grouping)
- Lots of assumptions and very rough estimation
  - Accurate estimate is not needed
  - Maybe okay if we overestimate or underestimate consistently
  - May lead to very nasty optimizer "hints"

```
SELECT * FROM User WHERE pop > 0.9;
SELECT * FROM User WHERE pop > 0.9 AND pop > 0.9;
```

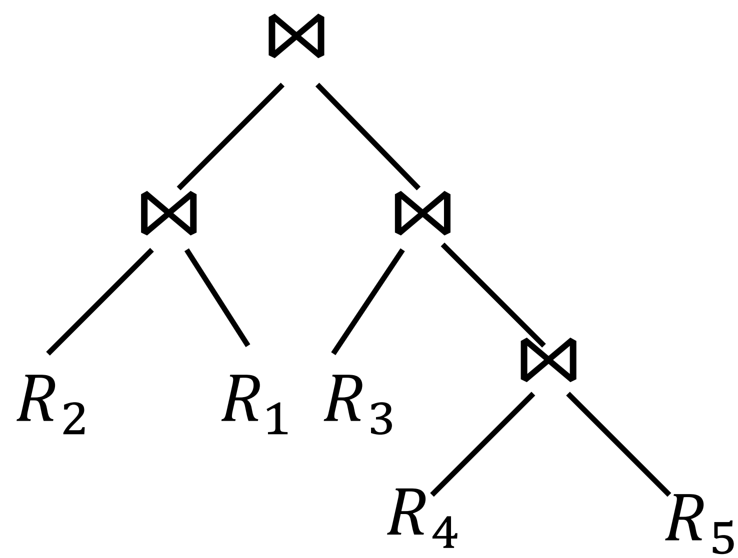
- Not covered: better estimation using histograms

# Search strategy


http://1.bp.blogspot.com/-Motdu8reRKs/TgyAi4ki5QI/AAAAAAAAAKE/mi8ejfZ8S7U/s1600/cornMaze.jpg

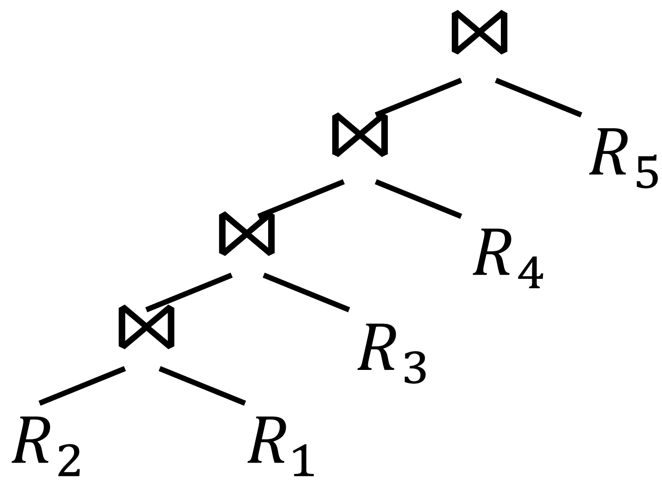
# Search space

- Huge!
- "Bushy" plan example:




- Just considering different join orders, there are $\frac{(2n-2)!}{(n-1)!}$ bushy plans for $R_1 \bowtie \cdots \bowtie R_n$
  - $30240$ for $n = 6$
- And there are more if we consider:
  - Multiway joins
  - Different join methods
  - Placement of selection and projection operators

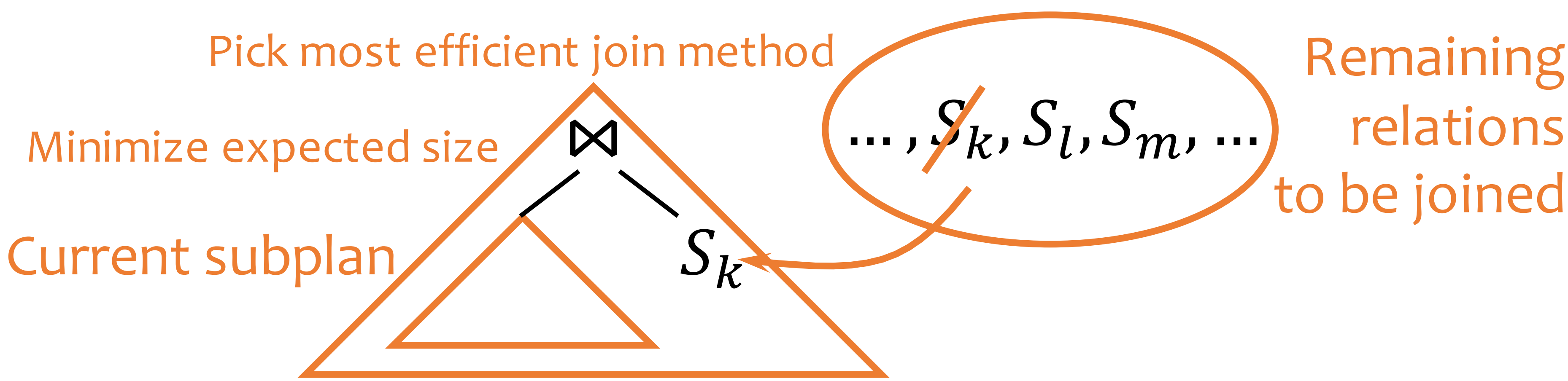
# Left-deep plans




- Heuristic: consider only "left-deep" plans, in which only the left child can be a join
  - Tend to be better than plans of other shapes, because many join algorithms scan inner (right) relation multiple times—you will not want it to be a complex subtree
- How many left-deep plans are there for $R_1 \bowtie \cdots \bowtie R_n$?

# A greedy algorithm

- $S_1, \dots, S_n$
  - Say selections have been pushed down; i.e., $S_i = \sigma_p(R_i)$
- Start with the pair $S_i, S_j$ with the smallest estimated size for $S_i \bowtie S_j$
- Repeat until no relation is left:
  Pick $S_k$ from the remaining relations such that the join of $S_k$ and the current result yields an intermediate result of the smallest size



# A dynamic programming approach

- Generate optimal plans bottom-up
  - Pass 1: Find the best single-table plans (for each table)
  - Pass 2: Find the best two-table plans (for each pair of tables) by combining best single-table plans
  - …
  - Pass $k$: Find the best $k$-table plans (for each combination of $k$ tables) by combining two smaller best plans found in previous passes
  - …
- Rationale: Any subplan of an optimal plan must also be optimal (otherwise, just replace the subplan to get a better overall plan)

☞Well, not quite…

# The need for "interesting order"

- Example: $R(A, B) \bowtie S(A, C) \bowtie T(A, D)$
- Best plan for $R \bowtie S$: hash join (beats sort-merge join)
- Best overall plan: sort-merge join $R$ and $S$, and then sort-merge join with $T$
  - Subplan of the optimal plan is not optimal!
- Why?
  - The result of the sort-merge join of $R$ and $S$ is sorted on $A$
  - This is an interesting order that can be exploited by later processing (e.g., join, dup elimination, `GROUP BY`, `ORDER BY`, etc.)!

# Dealing with interesting orders

When picking the best plan

- Comparing their costs is not enough
  - Plans are not totally ordered by cost anymore
- Comparing interesting orders is also needed
  - Plans are now partially ordered
  - Plan $X$ is better than plan $Y$ if
    - Cost of $X$ is lower than $Y$, and
    - Interesting orders produced by $X$ "subsume" those produced by $Y$
- Need to keep a set of optimal plans for joining every combination of $k$ tables
  - At most one for each interesting order

# Summary

- Relational algebra equivalence
- SQL rewrite tricks
- Heuristics-based optimization
- Cost-based optimization
  - Need statistics to estimate sizes of intermediate results
  - Greedy approach
  - Dynamic programming approach