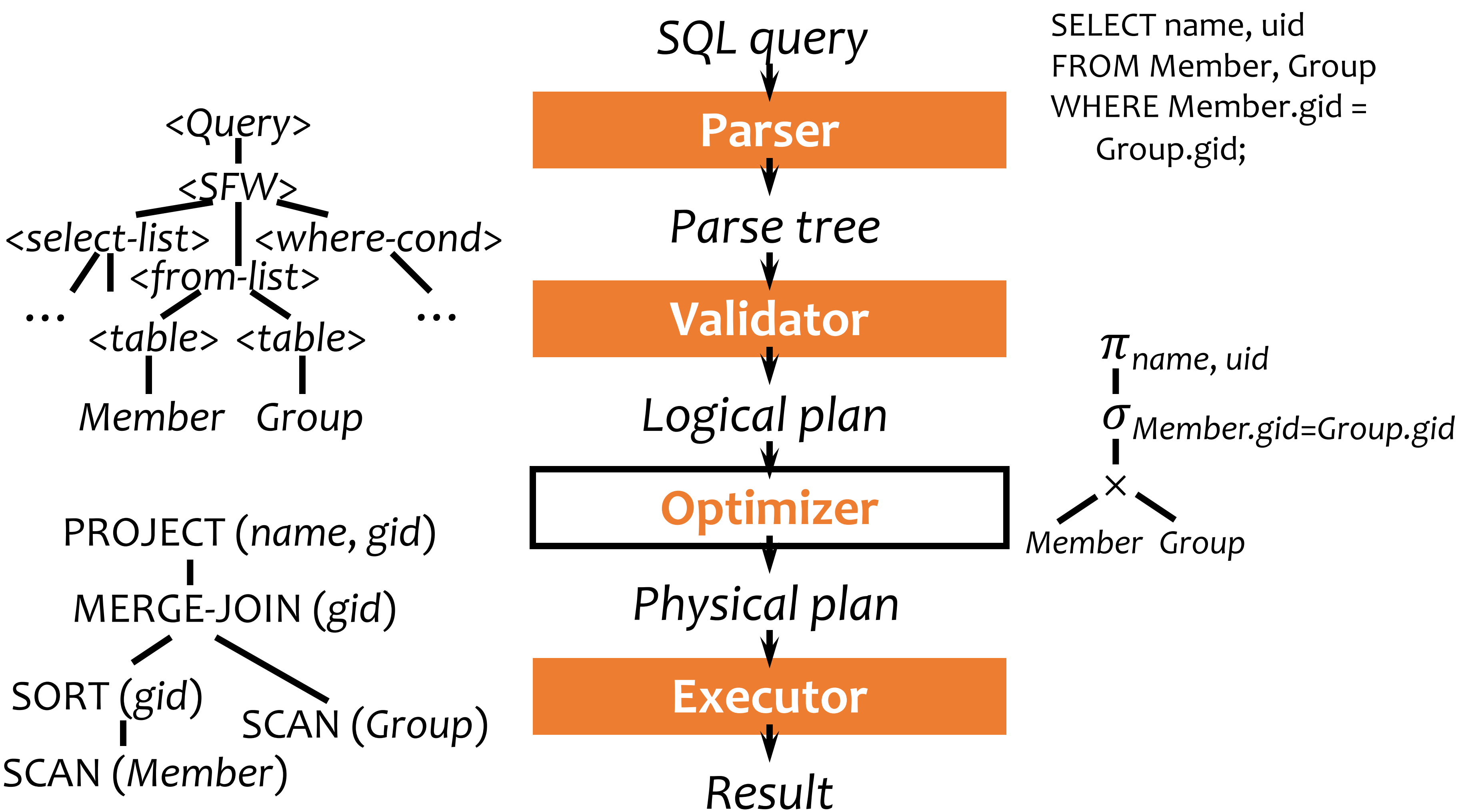
Lecture 18: Query Processing & Optimization

CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**

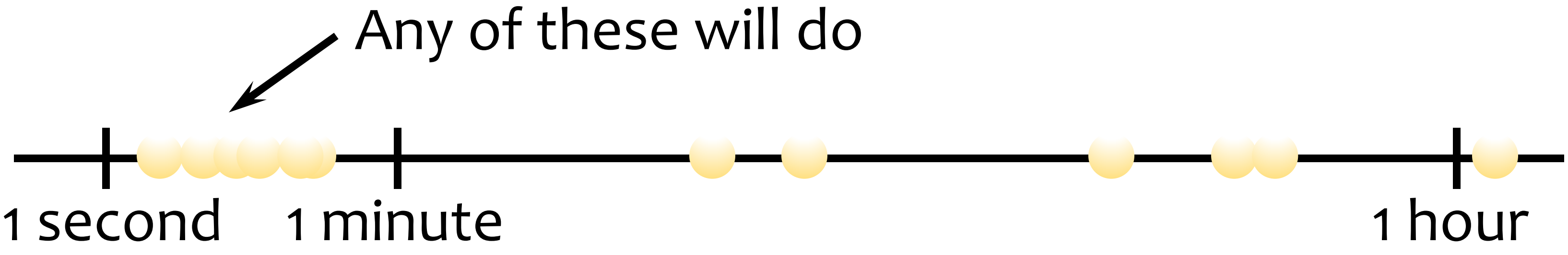
Sections: 001, 002, 003

A query’s trip through the DBMS



(Recap) Query optimization

- Why query optimization?
- Search space
  - What are the possible equivalent logical plans? (lecture 17)
  - What are the possible physical plans? (lectures 16-17)
- Search strategy
  - Rule-based strategy
  - Cost-estimation-based strategy



Search strategy

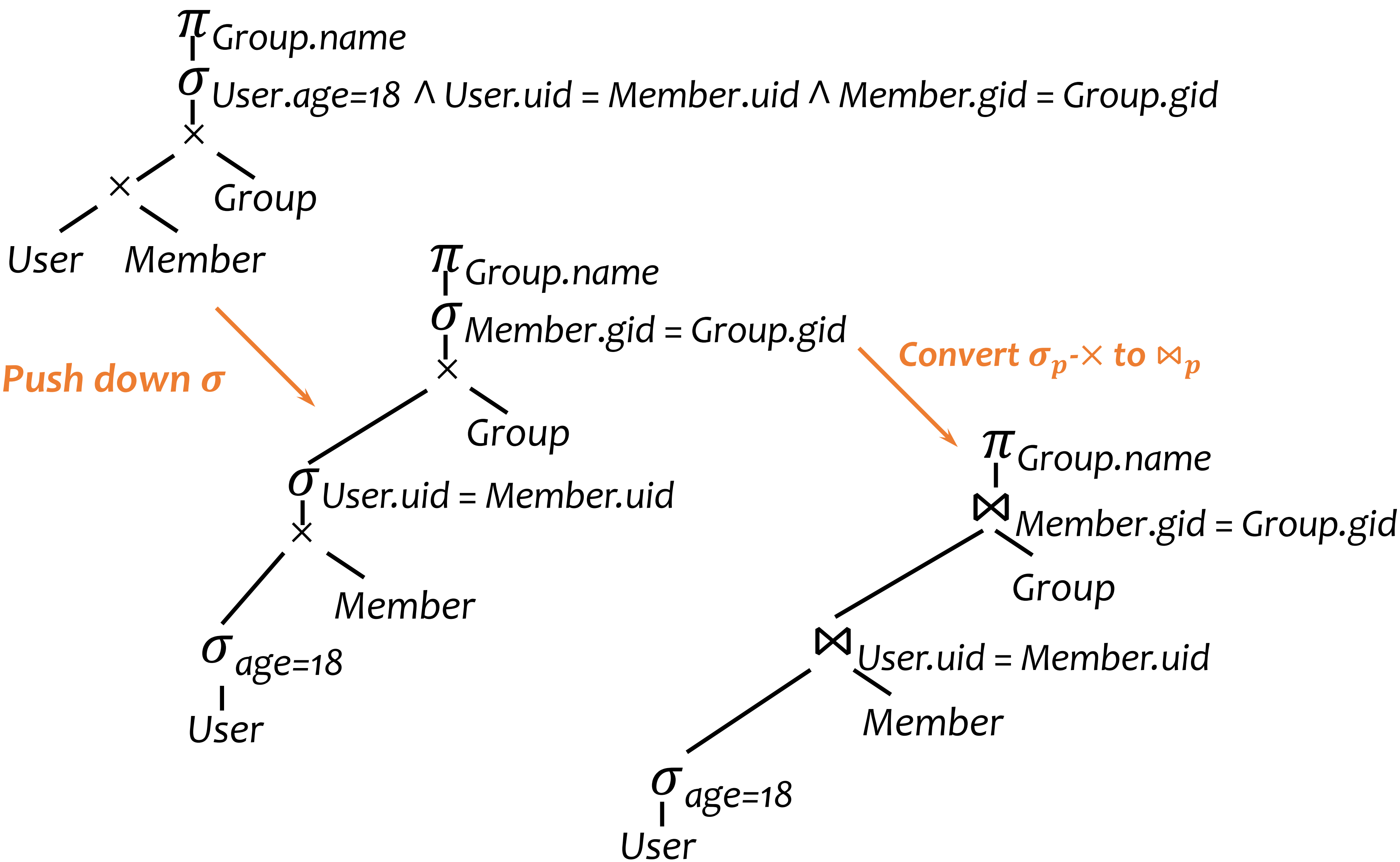
http://1.bp.blogspot.com/-Motdu8reRKs/TgyAi4ki5QI/AAAAAAAAAKE/mi8ejfZ8S7U/s1600/cornMaze.jpg

Rule-based query optimization

- Push $\sigma / \pi / -$ down as much as possible
  - $\sigma_{p \wedge p_R \wedge p_S}(R \bowtie_{p'} S) = (\sigma_{p_R} R) \bowtie_{p \wedge p'} (\sigma_{p_S} S)$
  - $\pi_L(\sigma_p R) = \pi_L\left(\sigma_p(\pi_{L \cup L'} R)\right)$
  - $\pi_L(R \bowtie_p S) = \pi_L\left((\pi_{L_R} R) \bowtie_p (\pi_{L_S} S)\right)$
  - $(R \bowtie S) - (T \bowtie S) = (R - T) \bowtie S$
  - $(R \bowtie S) - (T \bowtie W) = \left((R - T) \bowtie S\right) \cup \left(R \bowtie (S - W)\right)$
- Join smaller relations first and avoid cross product
- (Many other rules to be further exploited)

Why? Reduce the size of intermediate results

An Example



π Group.name
σ User.age=18 ∧ User.uid = Member.uid ∧ Member.gid = Group.gid
×
×
Group
User
Member
Push down σ
π Group.name
σ Member.gid = Group.gid
×
Group
σ User.uid = Member.uid
×
Member
σ age=18
User
Convert σp-× to ⋈p
π Group.name
⋈ Member.gid = Group.gid
Group
⋈ User.uid = Member.uid
Member
σ age=18
User

From rule-based to cost-based opt.

- Rule-based optimization
  - Apply algebraic equivalence 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 the plans
    - Pick a plan with an acceptable cost
  - Focus: select-project-join blocks

"Selinger"-style query optimization ← *Patricia Selinger*

Cost estimation

Physical plan example:

PROJECT (*Group.name*)

INDEX-NESTED-LOOP-JOIN (*gid*)

Index on *Group*(*gid*)

INDEX-NESTED-LOOP-JOIN (*uid*)

Index on *Member*(*uid*)

INDEX-SCAN (*age* = 18)

Index on *User*(*age*)

Input to Join(*uid*):

What is its input size? How many users with age 18?

- We have I/O cost formula for each operator

  Lectures 16- 17

  - Example: INDEX-NESTED-LOOP-JOIN (*uid*) takes

    $$O(B(R) + |R| \cdot \text{lookup} + \text{fetch})$$

- We need the size of the intermediate results

Cardinality Estimation

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

Selections with equality predicates

Consider $\sigma_{A=v}R$

- DBMSs typically store the following in the catalog
  - Size of R: $|R|$
  - Number of distinct A values in R: $|\pi_A R|$
- Assumption of uniformity: A-values are uniformly distributed in tuples from R
- $|\sigma_{A=v}R| \approx {}^{|R|}/_{|\pi_A R|}$
  - Selectivity factor of $(A = v)$ is ${}^{1}/_{|\pi_A R|}$
  - Selectivity: the probability that any row will satisfy a predicate

Conjunctive predicates

Consider $\sigma_{A=u \,\wedge\, B=v} R$

- Assumption of selection independence: $(A = u)$ and $(B = v)$ independently select tuple in R
  - Counterexample: major and advisor, or A is the key
- $|\sigma_{A=u \,\wedge\, B=v} R\,| \approx {}^{|R|}/_{|\pi_A R| \cdot |\pi_B R|}$
  - Selectivity factor of $(A = u)$ is ${}^{1}/_{|\pi_A R|}$
  - Selectivity factor of $(B = v)$ is ${}^{1}/_{|\pi_B R|}$
  - Selectivity factor of $(A = u) \wedge (B = v)$ is ${}^{1}/_{|\pi_A R| \cdot |\pi_B R|}$
  - Reduce total size by all selectivity factors

Negated and disjunctive predicates

Consider $\sigma_{A \neq v} R$

- $|\sigma_{A \neq v} R\ | \approx |R| \cdot \left(1 - {}^{1}/_{|\pi_A R|}\right)$
  - Selectivity factor of $\neg p$ is $(1 -$ selectivity factor of $p)$

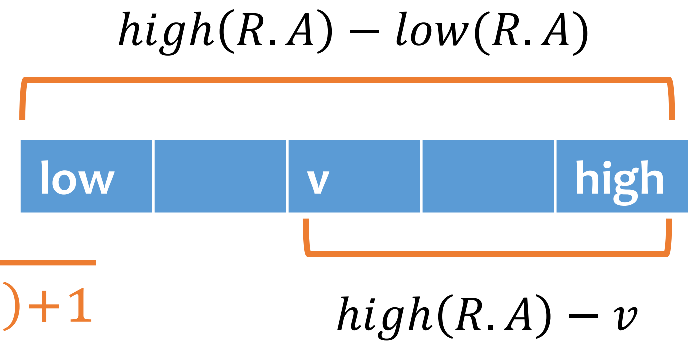
Consider $\sigma_{A=u\ \vee\ B=v} R$

- $|\sigma_{A=u\ \vee\ B=v} R| \approx |R| \cdot \left({}^{1}/_{|\pi_A R|} + {}^{1}/_{|\pi_B R|}\right)$?
  - Tuples satisfying $(A = u)$ and $(B = v)$ are counted twice!
- $|\sigma_{A=u\ \vee\ B=v} R| \approx |R| \cdot \left({}^{1}/_{|\pi_A R|} + {}^{1}/_{|\pi_B R|} - {}^{1}/_{|\pi_A R||\pi_B R|}\right)$
  - Inclusion-exclusion principle

Range predicates

Consider $\sigma_{A>v} R$

- DBMSs typically store the following in the catalog
  - Largest *R.A* value: $\mathrm{high}(R.A)$
  - Smallest *R.A* value: $\mathrm{low}(R.A)$



- $|\sigma_{A>v} R\ | \approx |R| \cdot \frac{\mathrm{high}(R.A) - v}{\mathrm{high}(R.A) - \mathrm{low}(R.A) + 1}$
  - Selectivity factor is $\frac{\mathrm{high}(R.A) - v}{\mathrm{high}(R.A) - \mathrm{low}(R.A)}$

Two-way natural join

- $Q = R(A, B) \bowtie S(A, C)$
- Assumption of 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 many practical cases
- $|Q| \approx \dfrac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)}$
  - Selectivity factor of $R.A = S.A$ is ${}^{1}/_{\max(|\pi_A R|, |\pi_A S|)}$

Multiway natural 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 of preservation of value sets
  - A non-join attribute does not lose values from its set of possible values
  - That is, if C is in S but not R, then $\pi_C(R \bowtie S) = \pi_C S$
  - Certainly not true in general
  - But holds many practical cases

Multiway natural join (cont'd)

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

Summary of Cardinality Estimation

- Lots of assumptions and very rough estimation
  - An accurate estimator is not needed
  - Maybe okay if we overestimate or underestimate, since it may not change the query plan selection

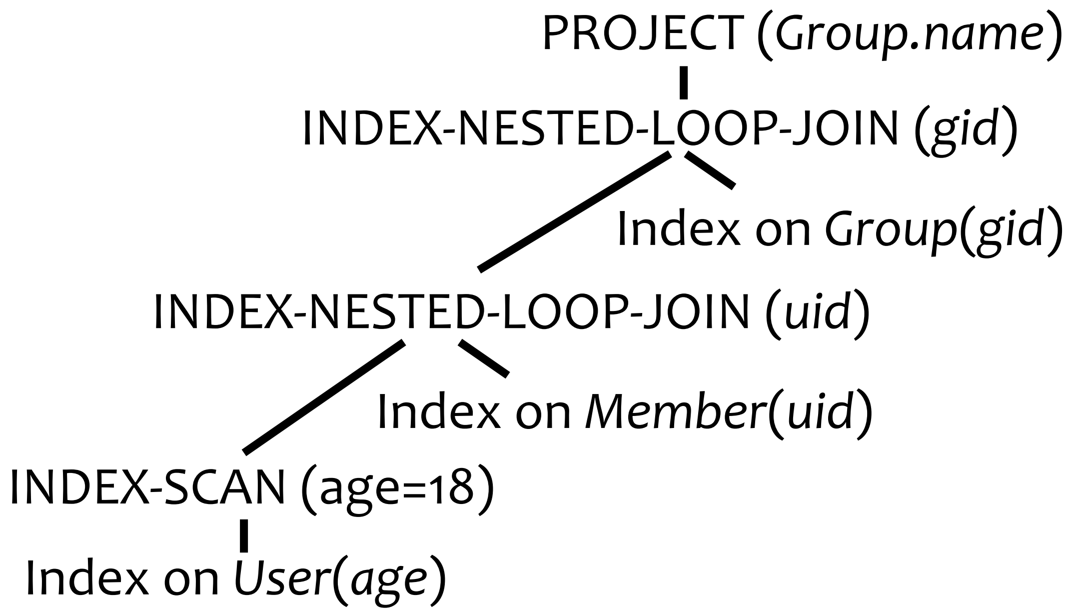
- (MUCH) Better techniques
  - Histograms
  - Sketches
  - Machine learning approaches (one of the most popular topics in database research now)

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

Data statistics:

- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{name} User| = 50$
- $|\pi_{uid} Member| = 500$
- $\left|\pi_{gid} Member\right| = 100$
- $|\pi_{gid} Group| = 100$



- What are the sizes of intermediate join results?

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

PROJECT (*Group.name*)
INDEX-NESTED-LOOP-JOIN (*gid*)
Index on *Group*(*gid*)
INDEX-NESTED-LOOP-JOIN (*uid*)
Index on *Member*(*uid*)
INDEX-SCAN (*age* = 18)
Index on *User*(*age*)

Data statistics:

- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age} User| = 50$
- $|\pi_{uid} Member| = 500$
- $|\pi_{gid} Member| = 100$
- $|\pi_{gid} Group| = 100$

- What is the intermediate size?
  - Assumption of uniformity
  - $|\sigma_{age=18}(User)| \approx \frac{1000}{50} = 20$

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

PROJECT (*Group.name*)

INDEX-NESTED-LOOP-JOIN (*gid*)

Index on *Group*(*gid*)

INDEX-NESTED-LOOP-JOIN (*uid*)

Index on *Member*(*uid*)

INDEX-SCAN (*age* = 18)

Index on *User*(*age*)

Data statistics:

- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age}User| = 50$
- $|\pi_{uid}Member| = 500$
- $|\pi_{gid}Member| = 100$
- $|\pi_{gid}Group| = 100$

- What is the intermediate join size?
  - Assume uniformity of age in User
  - Assume containment of value sets in User and Member on uid
  - $|\sigma_{age=18}(User) \bowtie Member\ | \approx \dfrac{|\sigma_{age=18}(User)| \cdot |Member|}{\max(|\pi_{uid}\sigma_{age=18}(User)|, |\pi_{uid}Member|)}$

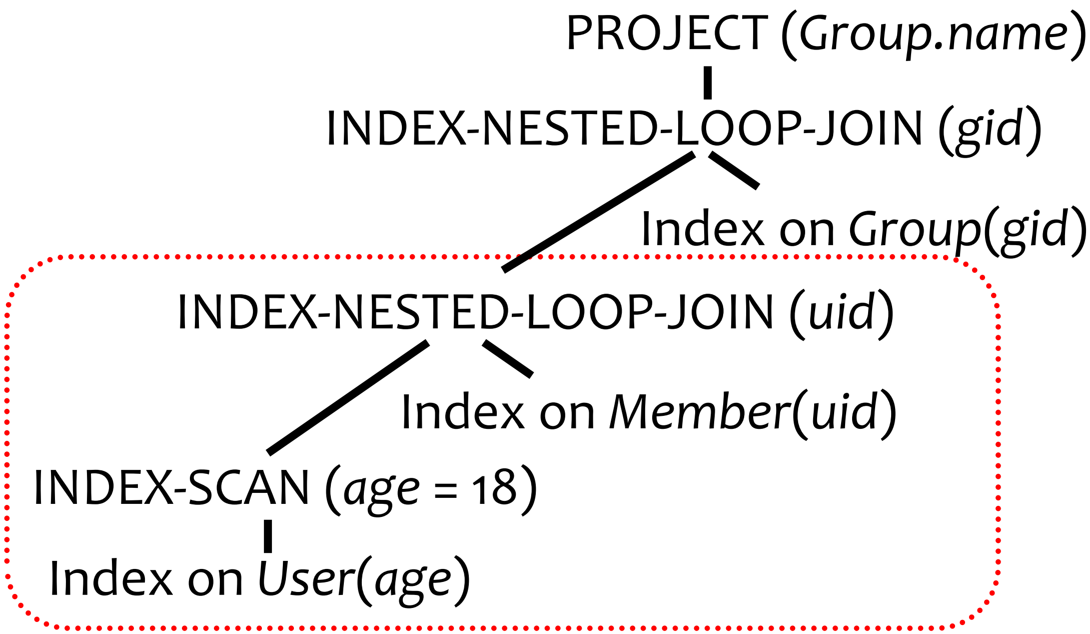
$$= \frac{20 \cdot 50000}{\max(20{,}500)} = 2000$$

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

Data statistics:

- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age} User| = 50$
- $|\pi_{uid} Member| = 500$
- $|\pi_{gid} Member| = 100$
- $|\pi_{gid} Group| = 100$



- What is the intermediate join size? (an alternative estimate)
  - Assume the instance follows the foreign-key constraint
  - Assume that within each group, users of different ages join with equal probability
  - $|\sigma_{age=18}(User) \bowtie Member\ | = |\sigma_{age=18}(User \bowtie Member)|$

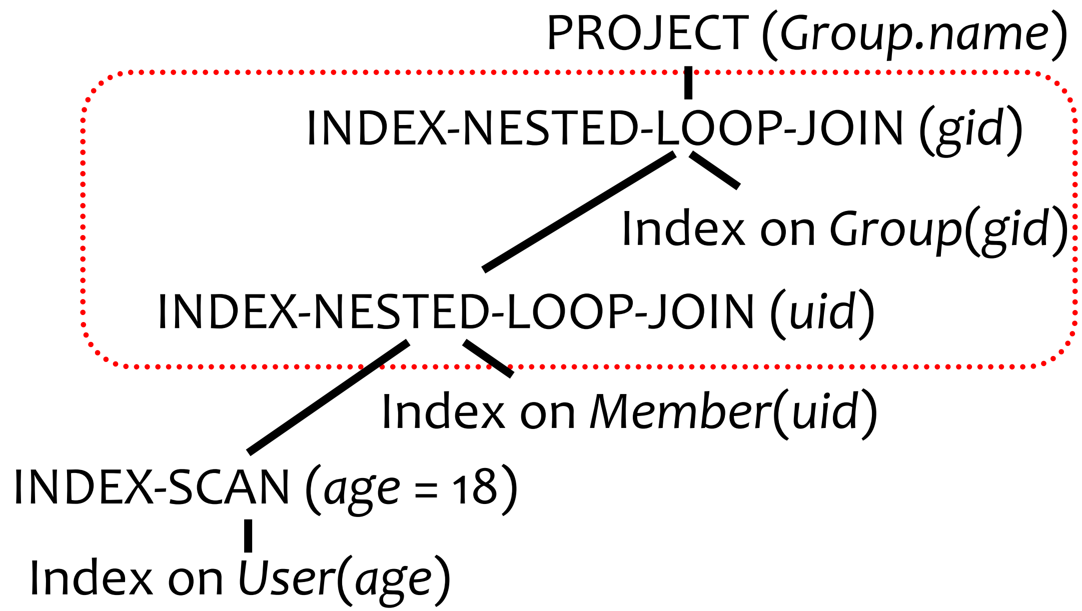
$$\approx \frac{1}{50} \cdot 50000 = 1000$$

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

Data statistics:

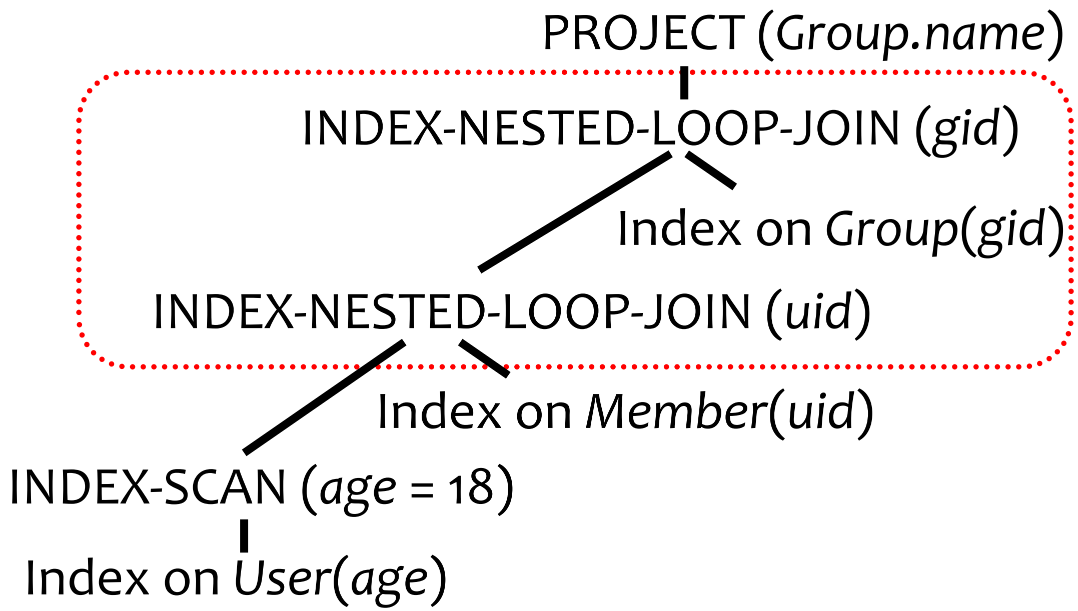
- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age}User| = 50$
- $|\pi_{uid}Member| = 500$
- $|\pi_{gid}Member| = 100$
- $|\pi_{gid}Group| = 100$



- What is the intermediate join size?
  - Assumption of preservation of value sets
  - $Q' = \sigma_{age=18}(User) \bowtie Member$ (suppose we use estimator 2000)
  - $|Q' \bowtie Group\,| \approx \frac{|Q'|\cdot|Group|}{\max(|\pi_{gid}Q'|,|\pi_{gid}Group|)} = \frac{2000\cdot 100}{\max(100,100)} = 2000$

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)



Data statistics:

- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age} User| = 50$
- $|\pi_{uid} Member| = 500$
- $|\pi_{gid} Member| = 100$
- $|\pi_{gid} Group| = 100$

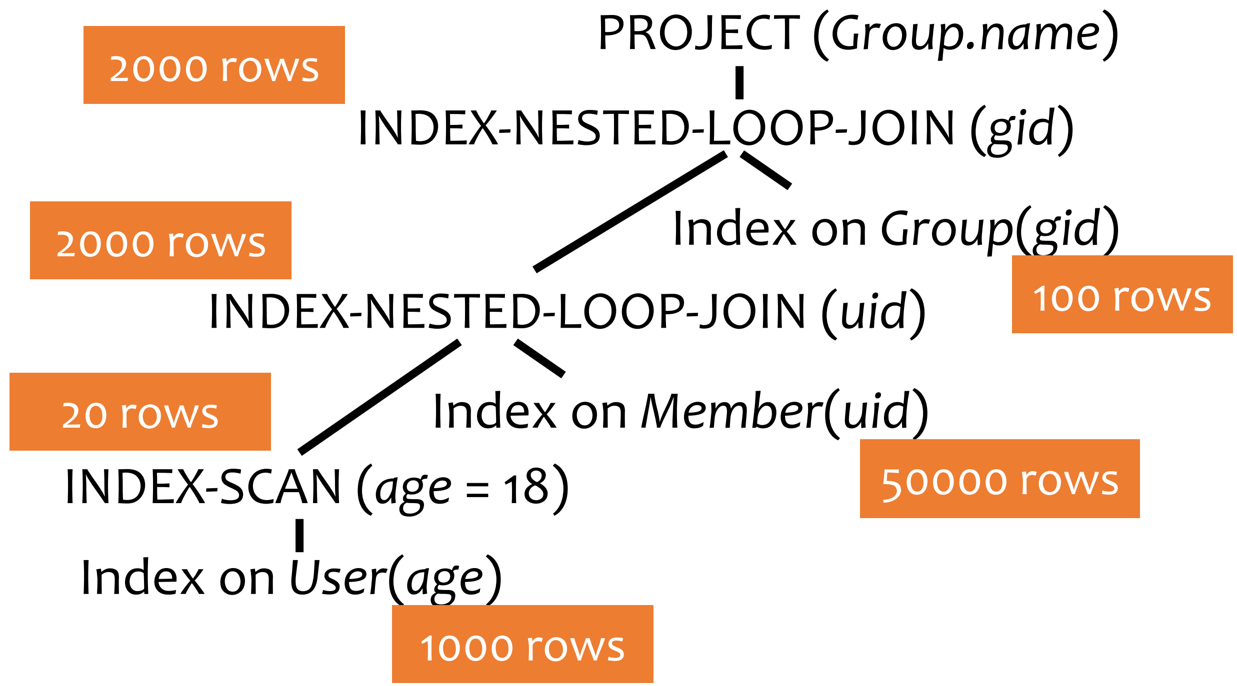
- What is the intermediate join size? (an alternative estimate)
  - Assume the instance follows the foreign-key constraint
  - $Q' = \sigma_{age=18}(User) \bowtie Member$ (suppose we use estimator 2000)
  - $|Q' \bowtie Group| \approx |Q'| = 2000$

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

Data statistics:

- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age} User| = 50$
- $|\pi_{uid} Member| = 500$
- $|\pi_{gid} Member| = 100$
- $|\pi_{gid} Group| = 100$



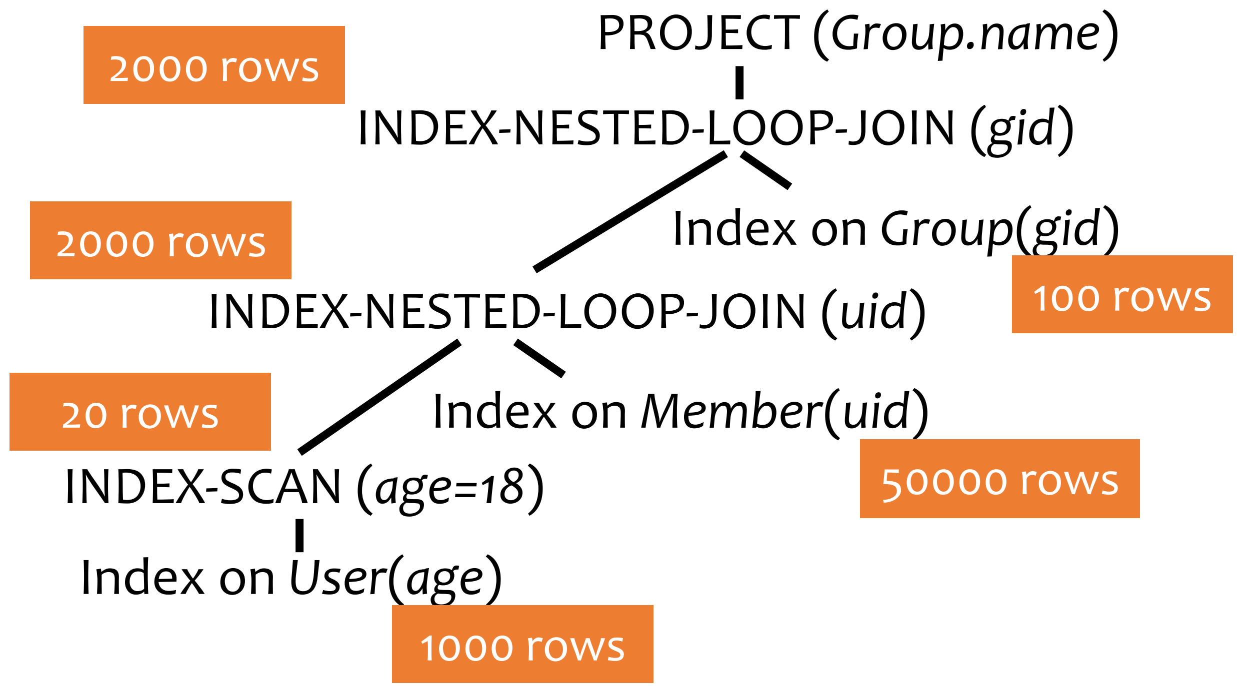
- What are the estimated I/O costs? (Some rough calculation)
- System requirements:
  - Each disk/memory block can hold up to 10 rows (from any table);
  - All tables are stored compactly on disk (10 rows per block);
  - 8 memory blocks are available for query processing: M=8

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

Data statistics:

- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age}User| = 50$
- $|\pi_{uid}Member| = 500$
- $|\pi_{gid}Member| = 100$
- $|\pi_{gid}Group| = 100$



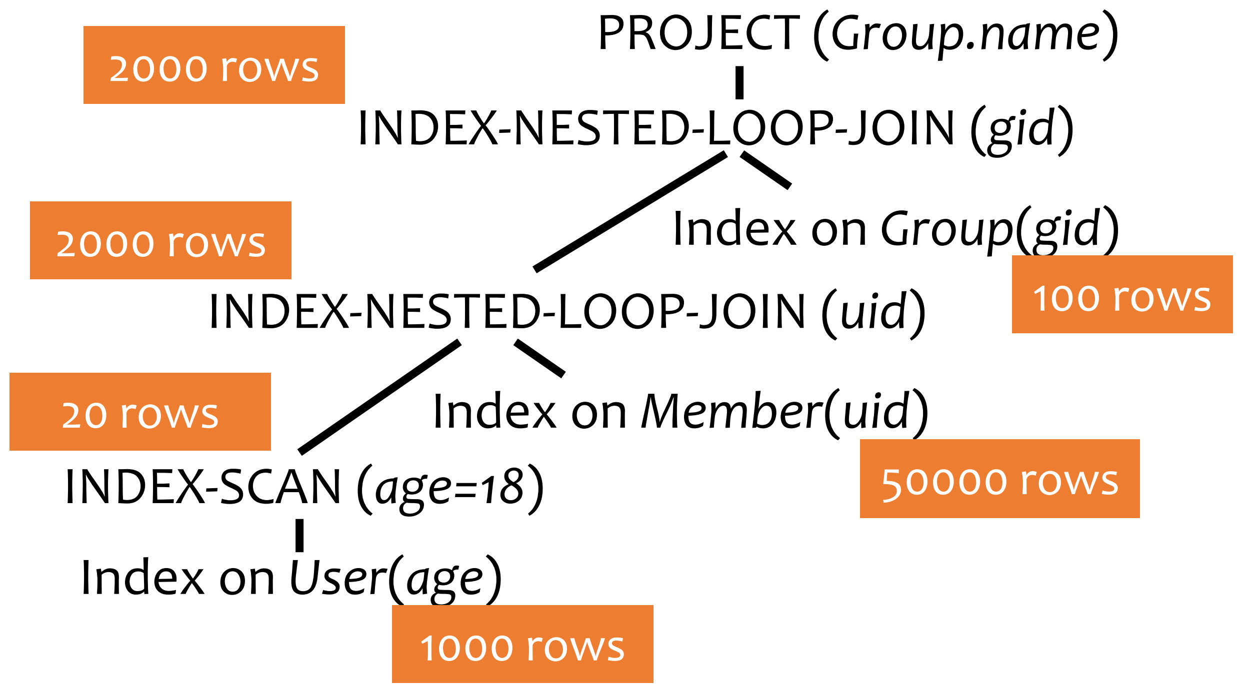
- INDEX-SCAN on User
  - Index lookup: typically 4 I/Os
  - Suppose tuples in the User table are clustered by uid but not age
  - In the worst case, we need 20 I/Os to retrieve tuples
  - Cache the intermediate results in main memory using 2 blocks

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)

Data statistics:

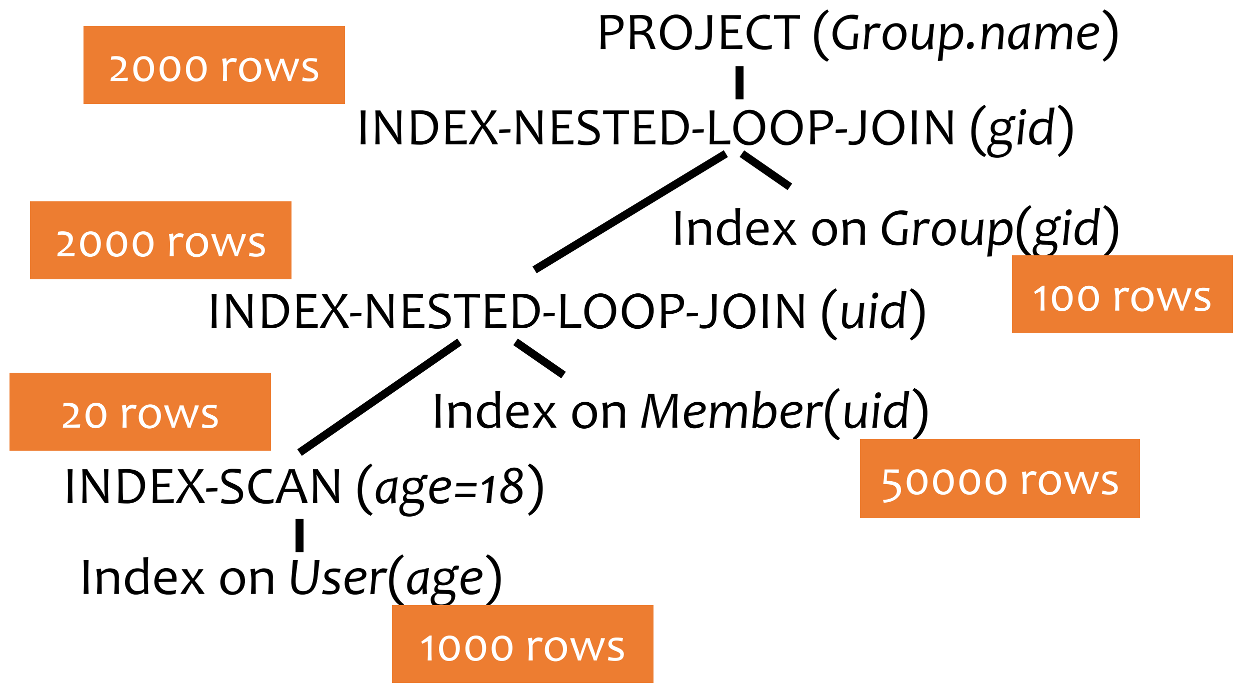
- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age} User| = 50$
- $|\pi_{uid} Member| = 500$
- $|\pi_{gid} Member| = 100$
- $|\pi_{gid} Group| = 100$



- INDEX-NESTED-LOOP-JOIN on $\sigma_{age=18}(User)$ and Member
  - Suppose tuples in Member table clustered by uid
  - Index lookup: typically 4 I/Os
  - Probing takes $20 \cdot \text{lookup} + \text{fetch} = 20 * 4 + \frac{2000}{10} = 280$ I/Os
  - Writing the intermediate results back to disk takes $\frac{2000}{10} = 200$ I/Os

Case Study

User (<u>*uid*</u> int, *name* string, *age* int, *pop* float)
Group (<u>*gid*</u> string, *name* string)
Member (<u>*uid*</u> int, <u>*gid*</u> string)



Data statistics:

- $|User| = 1000$
- $|Member| = 50000$
- $|Group| = 100$
- $|\pi_{age} User| = 50$
- $|\pi_{uid} Member| = 500$
- $|\pi_{gid} Member| = 100$
- $|\pi_{gid} Group| = 100$

- INDEX-NESTED-LOOP-JOIN on Q' and Group
  - $Q' = \sigma_{age=18}(User) \bowtie Member$
  - For each tuple in Q', probe the index on Group(gid)
  - Suppose tuples in Group table are clustered by gid
  - It takes $\mathrm{B(Q')} + |Q'| \cdot \mathrm{lookup} + \mathrm{fetch} = 200 + 2000 * 4 + 2000$ I/Os
  - When a join result is generated in memory, compute the projection on the fly

From rule-based to cost-based opt.

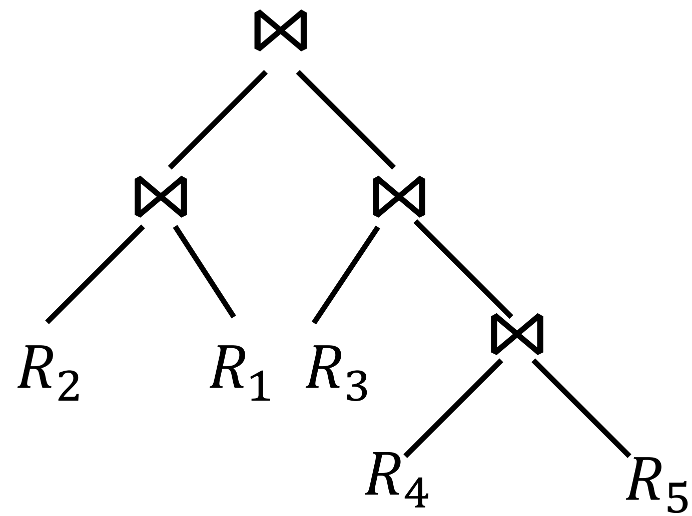
- Rule-based optimization
  - Apply algebraic equivalence to rewrite plans into cheaper ones
- Cost-based optimization
  - Rewrite logical plan to combine "blocks" as much as possible
  - Optimize the query block by block
    - Enumerate logical plans (already covered)
    - Estimate the cost of the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

"Selinger"-style query optimization ← *Patricia Selinger*

Search Good Join Ordering

Consider $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$:

- Search space is Huge!
    - “Bushy” plan example:

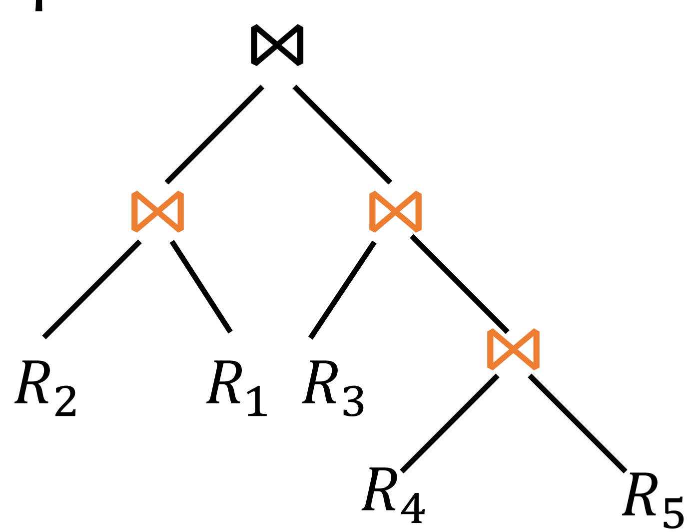


- Just considering different join orders, there are 30240 bushy plans for joining 6 tables !
- (there are more if we consider using different join algorithms)

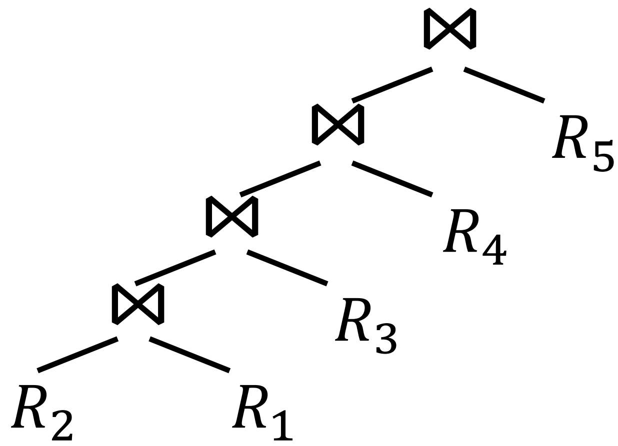
It is a matter of intermediate join size

- For simplicity, we use the sort-merge join algorithm for demonstration
- $R \bowtie S$ takes $O\left(B(R) + B(S) + \frac{B(R \bowtie S)}{MB}\right)$ I/Os (see Slide 44 in Lecture 16)
- Intermediate join results are needed to be written back to disk, and serve as an input table for the subsequent join

How to minimize the intermediate join size?



Left-deep plans

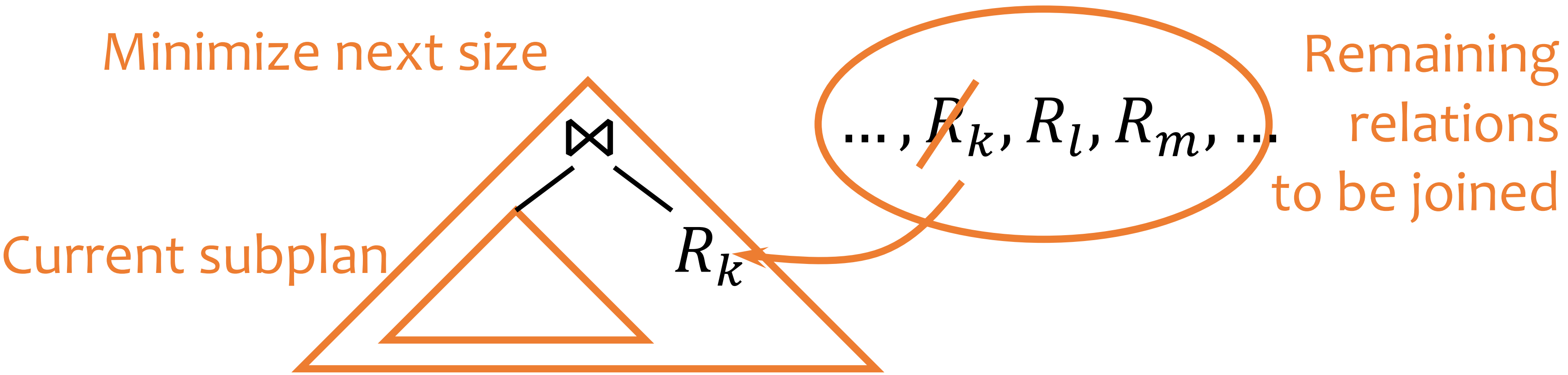


- Heuristic: consider only "left-deep" plans, in which only the left child can be a join
- How many left-deep plans are there for $R_1 \bowtie \cdots \bowtie R_n$?
  - Significantly fewer, but still lots— $n!$ (720 for $n = 6$)

A greedy algorithm for deep-left plans

Consider $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

- Start with the pair R_i, R_j with the smallest estimated size for $R_i \bowtie R_j$
- Repeat until no relation is left: Pick R_k from the remaining relations such that the join of R_k and the current result yields an intermediate result of the smallest estimated size



A DP algorithm for bushy plans

- Dynamic Programming for computing the optimal bushy plans in a bottom-up way:
  - 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)

Summary of Search Strategy

- Rule-based strategy
  - Apply algebraic equivalence to rewrite plans
    - But, we need more rewrite rules!
    - Existing rewrite plan is blind to the data statistics
- Cost-based strategy
  - Rewrite logical plan to combine “blocks” as much as possible
  - Optimize the query block by block
    - Enumerate logical plans (already covered)
    - Estimate the cost of the plans
      - Cardinality estimation is still challenging for multiple tables!
      - We need better end-to-end cost estimators
    - Pick a plan with an acceptable cost

A query’s trip through the DBMS