

# Database Systems I

CMPT 354 Summer 2024

Zhengjie Miao

# Announcements (Wed. July 24)

- A3 grade released
- Exam II grade releasing soon *When?*
- Expect sample questions for Exam III in the weekend
- A5 dues on next Monday

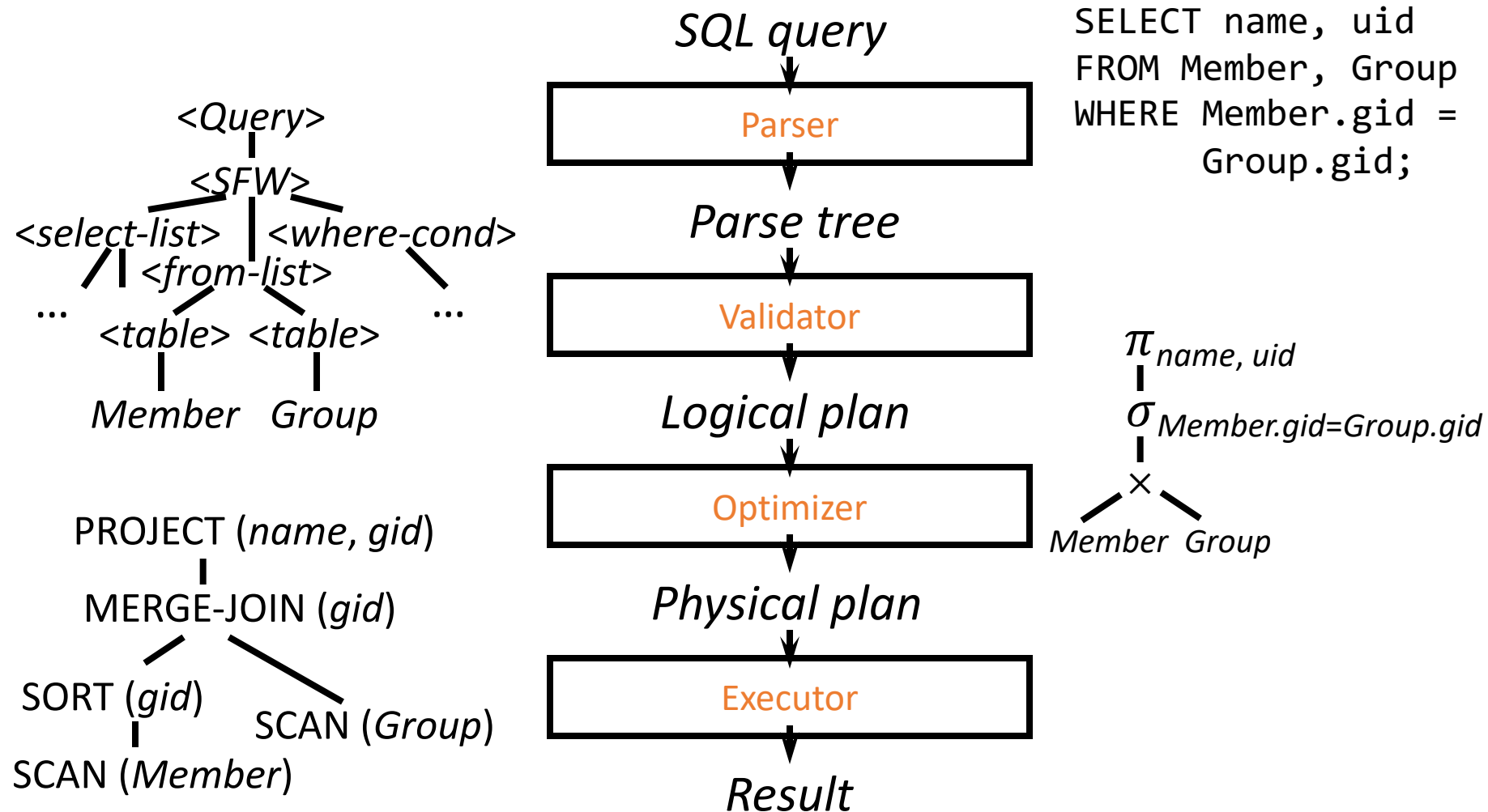
# Overview

- Many ways to process the same query
  - Scan? Sort? Hash? Use an index?
  - All have different performance characteristics and/or make different assumptions about data
- **Best choice depends on the situation**
  - Implement all alternatives
  - Let the **query optimizer** choose at run-time

# Outline

- System view of query processing
  - Logical plan and physical plan
- Cost calculation of the physical plan
  - Cardinality estimation
- Search space and search strategy
  - Transformation rules
  - Other heuristics

# A query's trip through the DBMS



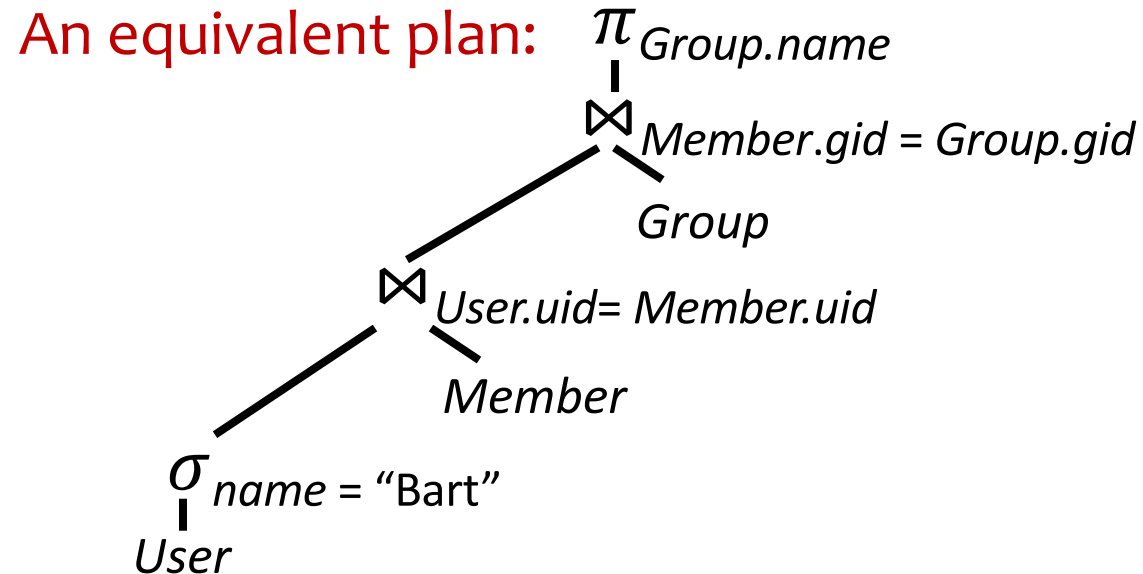
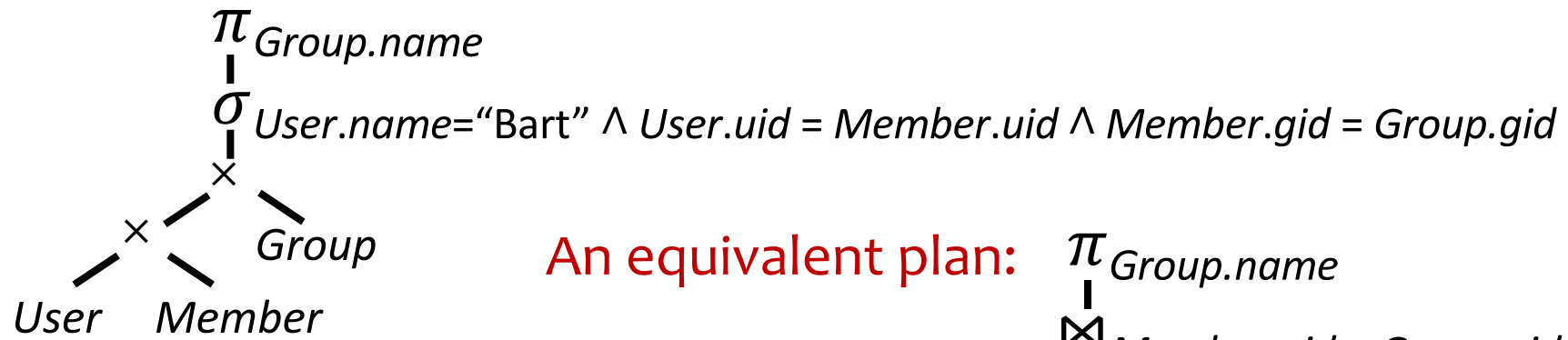
# Parsing and validation

- **Parser: SQL → parse tree**
  - Detect and reject **syntax** errors
- **Validator: parse tree → logical plan**
  - Detect and reject **semantic** errors
    - Nonexistent tables/views/columns?
    - Insufficient access privileges?
    - Type mismatches?
      - Examples: AVG(name), name + pop, User UNION Member
  - Also
    - Resolve column references
    - Expand \*, Expand view definitions
  - Information required for semantic checking is found in **system catalog** (which contains all schema information)

Parser creates AST, the validator checks that everything in the tree complies with the restrictions (types, existing tables, etc)

# Logical plan

- Nodes are **logical** operators (often relational algebra operators)
- There are many equivalent logical plans



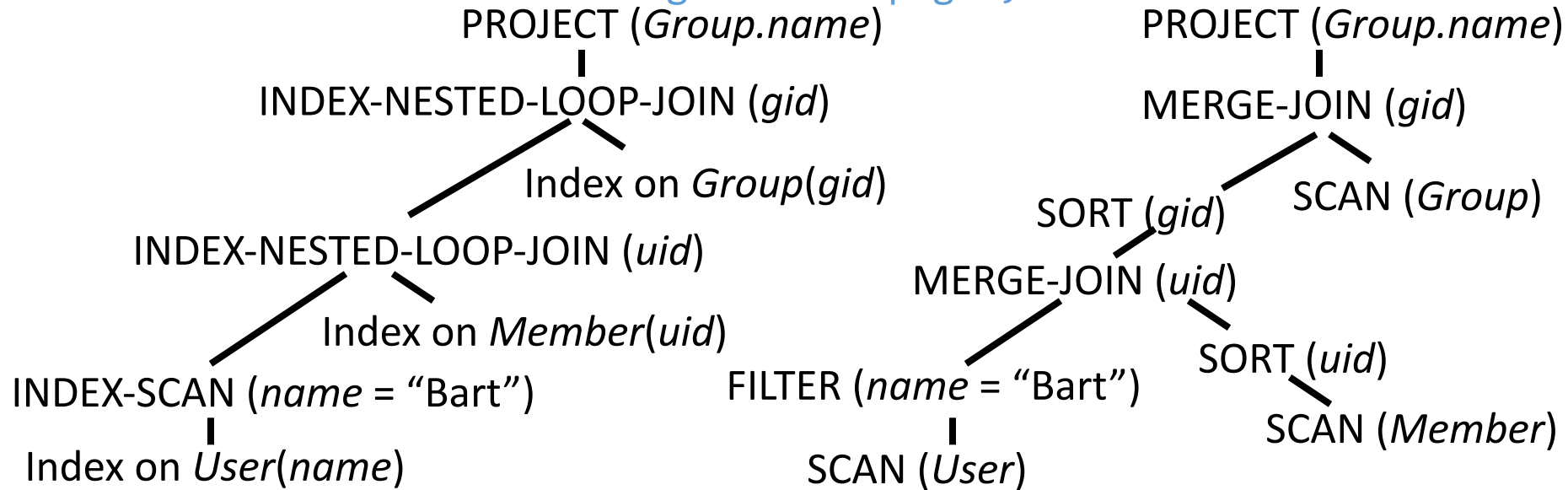
# Physical (execution) plan

- A complex query may involve multiple tables and various query processing algorithms
  - E.g., table scan, index nested-loop join, sort-merge join, hash-based duplicate elimination... (covered in Lec 16)
- A **physical plan** for a query tells the DBMS query processor how to execute the query
  - A tree of **physical plan operators**
  - Each operator implements a query processing algorithm
  - Each operator accepts one or more input tables/streams and produces a single output table/stream



# Examples of physical plans

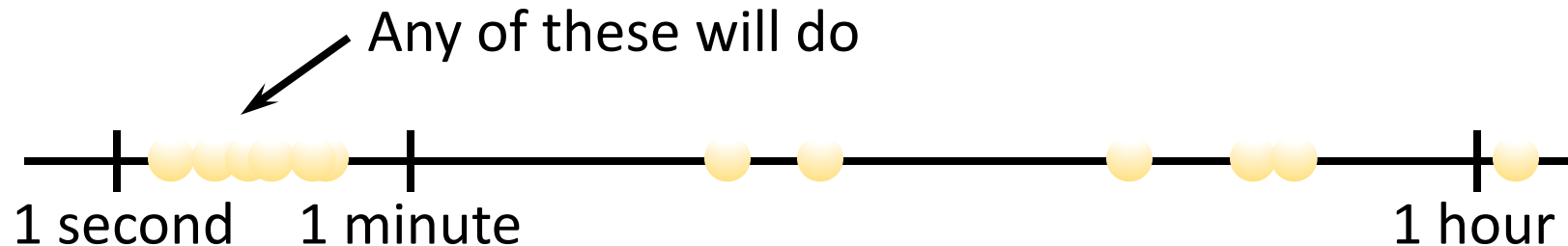
```
SELECT Group.name  
FROM User, Member, Group  
WHERE User.name = 'Bart'  
AND User.uid = Member.uid AND Member.gid = Group.gid;
```



- Many physical plans for a single query
  - Equivalent results, but different costs and assumptions!
  - 👉 DBMS query optimizer picks the “best” possible physical plan

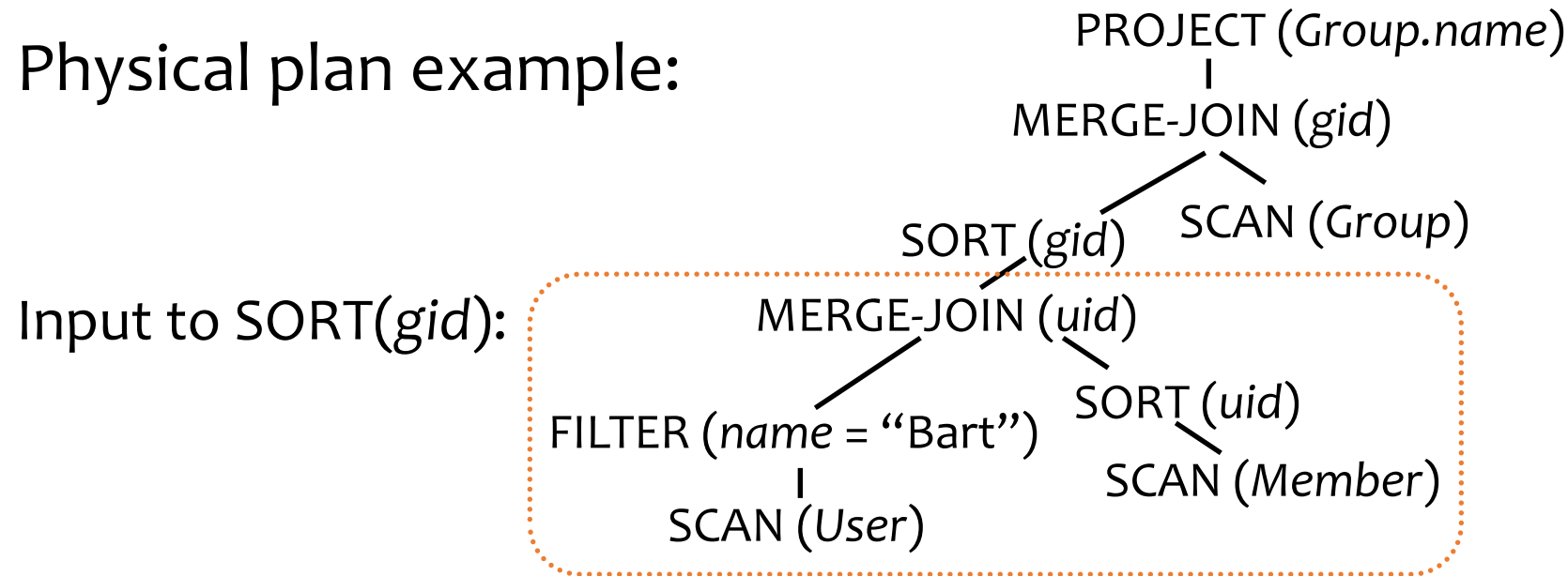
# Query optimization: how to pick the physical plan?

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



# Cost estimation

Physical plan example:



- We have: cost estimation for each operator
  - Example:  $\text{SORT}(gid)$  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

Cardinality estimation for:

- Equality predicates
- Range predicates
- Joins
- Other operators refer to the textbook



# 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 are unrealistic, but they work in practice.
- 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 What is over-selection?
    - Counterexample:  $A$  is the key
- $|Q| \approx \frac{|R|}{|\pi_A R| \cdot |\pi_B R|}$ 
  - Reduce total size by all selectivity factors

# Negated and disjunctive predicates

- $Q: \sigma_{A \neq v} R$ 
  - $|Q| \approx |R| \cdot \left(1 - \frac{1}{|\pi_A R|}\right)$ 
    - Selectivity factor of  $\neg p$  is  $(1 - \text{selectivity factor of } p)$
- $Q: \sigma_{A=u \vee B=v} R$ 
  - $|Q| \approx |R| \cdot \left(\frac{1}{|\pi_A R|} + \frac{1}{|\pi_B R|}\right)?$ 
    - No!
  - $|Q| \approx |R| \cdot \left(\frac{1}{|\pi_A R|} + \frac{1}{|\pi_B R|} - \frac{1}{|\pi_A R| |\pi_B R|}\right)$ 
    - Inclusion-exclusion principle

When you sum, you will overcount.  
Must apply inclusion-exclusion to avoid this.

# 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  $1 / \max(|\pi_A R|, |\pi_A S|)$

# Example

- Database:
  - User(uid, name, age, pop), Member(gid,uid,date), Group(gid, gname)
  - $|User| = 1000$  rows,  $|Group| = 100$  rows,  $|Member| = 50000$  rows
  - $|\pi_{name}(User)| = 50$
  - $|\pi_{uid}(Member)| = 500$
- Estimate size  $|User \bowtie Member| = ?$ 
  - $|\pi_{uid}(User)| = 1000$
  - $|\pi_{uid}(Member)| = 500$
  - $1000 * 50000 / \max(500, 1000) = 50000$

# 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)

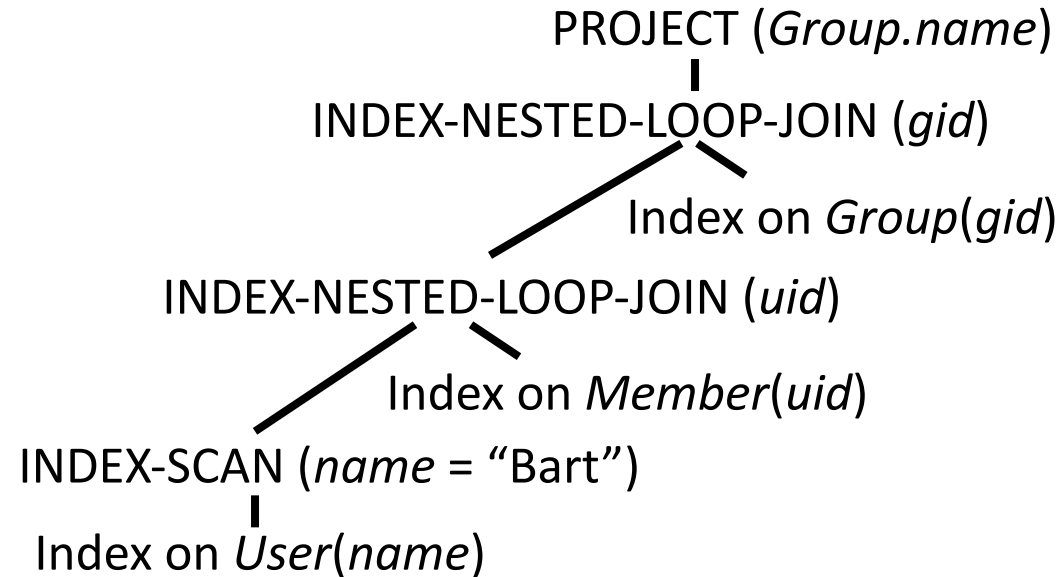
- $Q: R(A, B) \bowtie S(B, C) \bowtie T(C, D)$
- Start with the product of relation sizes
  - $|R| \cdot |S| \cdot |T|$
- Reduce the total size by the selectivity factor of each join predicate
  - $R.B = S.B: \frac{1}{\max(|\pi_B R|, |\pi_B S|)}$
  - $S.C = T.C: \frac{1}{\max(|\pi_C S|, |\pi_C T|)}$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

# Cost estimation example

- $Q: R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ 
  - $|R| = 1000, |\pi_B R| = 20$
  - $|S| = 2000, |\pi_B S| = 50, |\pi_C S| = 100$
  - $|T| = 5000, |\pi_C T| = 500$
- Estimation method 1:  $(R \bowtie S) \bowtie T$ 
  - $|R \bowtie S| = \frac{|R| \cdot |S|}{\max(|\pi_B R|, |\pi_B S|)} = \frac{1000 \times 2000}{50} = 40K$
  - $|(R \bowtie S) \bowtie T| = \frac{|R \bowtie S| \cdot |T|}{\max(|\pi_C(R \bowtie S)|, |\pi_C T|)} = \frac{40000 \times 5000}{500} = 400K$
- Estimation method 2:  $R \bowtie (S \bowtie T)$ 
  - $|S \bowtie T| = \frac{|S| \cdot |T|}{\max(|\pi_C S|, |\pi_C T|)} = \frac{2000 \times 5000}{500} = 20K$
  - $|R \bowtie (S \bowtie T)| = \frac{|R| \cdot |S \bowtie T|}{\max(|\pi_B(R)|, |\pi_B(S \bowtie T)|)} = \frac{1000 \times 20000}{50} = 400K$

# More example

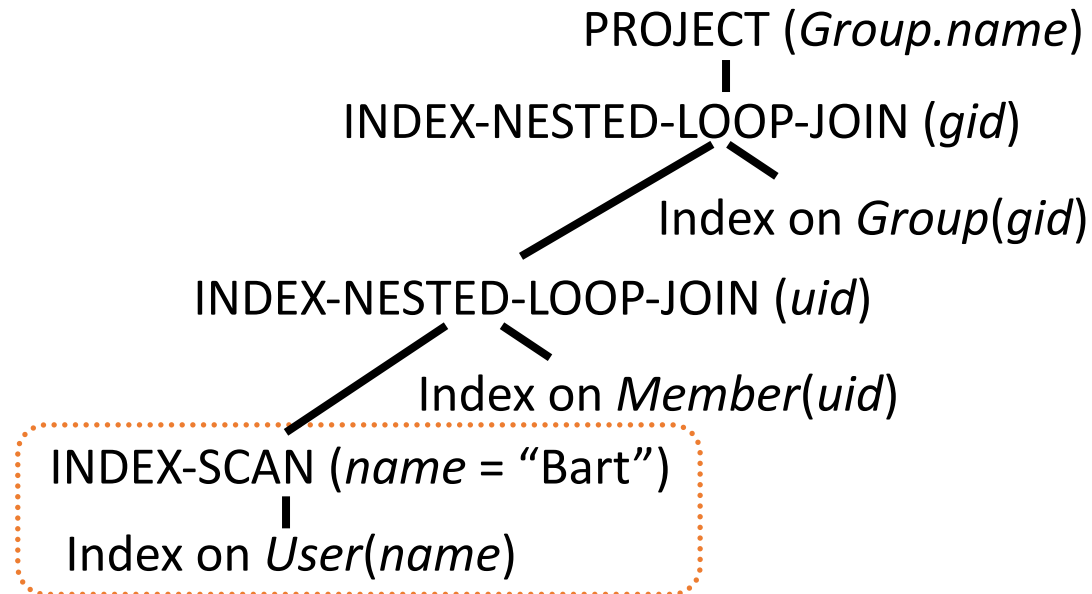
Physical plan example:



- 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**
- Database:
  - User(uid, age, pop), Member(gid,uid,date), Group(gid, gname)
  - | User |=1000 rows, | Group |=100 rows, | Member |=50000 rows
  - #of blocks: B(User) = 1000/10 = 100; B(Group) = 100/10 = 10; B(Member) = 50000/10=5k

# More example

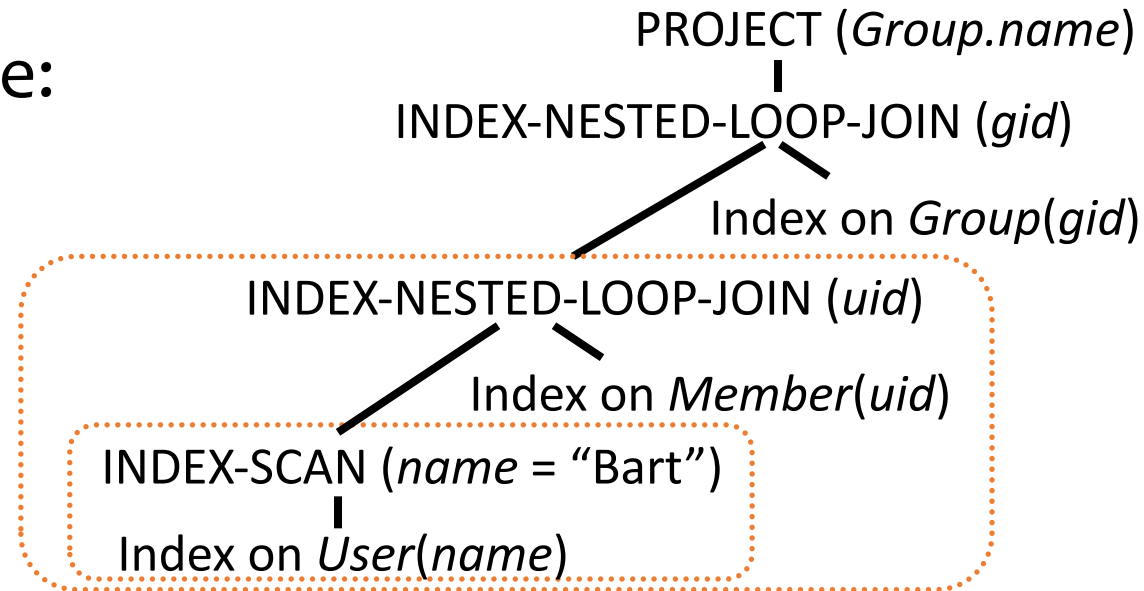
Physical plan example:



- Given  $|User| = 1000$ ,  $|\pi_{name}(User)| = 50$
- $|\sigma_{name="Bart"}(User)| = 1000 / 50 = 20$  records
- INDEX-SCAN on User
  - IO COST: index lookup ( $\sim 4$  IOs, depending on the height of the index tree)

# More example

Physical plan example:

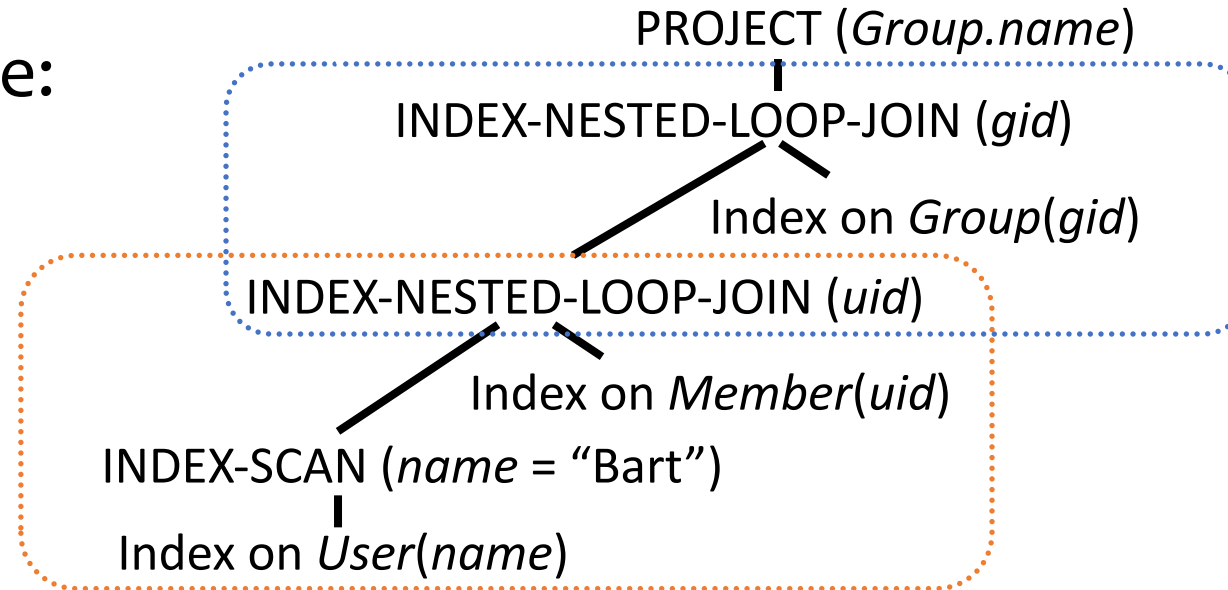


- Given  $|User|=1000$ ,  $|\pi_{name}(User)| = 50$ ,  $|\sigma_{name="Bart"}(User)| = 1000/50 = 20$  records
- INDEX-SCAN on User
  - IO COST: index lookup (4 IOs, depending on the height of the index tree)
- JOIN: For each record with name = "Bart", probe the index on  $Member(uid)$ 
  - IO cost:  $B(R) + |R| \cdot (\text{index lookup} + \text{record fetch})$
  - 20 rows are not clustered  $\rightarrow$  in the worst case, 20 blocks of data to be retrieved
  - $20 + 20 * (4 \text{ IOs for index} + \text{record fetches})$



# More example

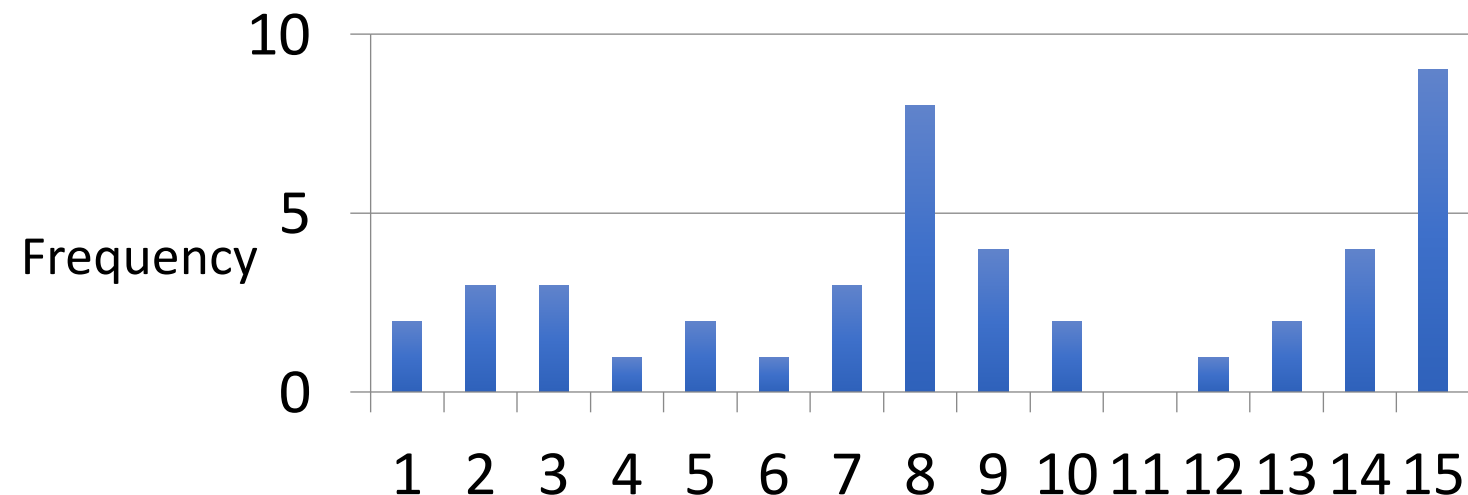
Physical plan example:



- Given  $|\pi_{uid}(\sigma_{name="Bart"}(User))| = 20$ ,  $|\pi_{uid}(Member)| = 500$ ,  $|\pi_{uid}(Group)| = 100$
- $|JOIN(uid)| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)} = \frac{20 \cdot 50k}{\max(20, 500)} = \frac{1000k}{500} = 2k$
- Assume preservation of value sets:  $|\pi_{gid}(User \bowtie Member)| = |\pi_{gid} Member|$ , but how about  $|\pi_{gid}(\sigma_{name="Bart"}(User) \bowtie Member)|$ ?
  - Depending on the distribution, here we assume  $|\pi_{gid}(\sigma_{name="Bart"}(User) \bowtie Member)| = 50$
- $|JOIN(gid)| \approx \frac{|R| \cdot |S|}{\max(|\pi_A R|, |\pi_A S|)} = \frac{2k \cdot 100}{\max(50, 100)} = \frac{200k}{100} = 2k$

# Cost estimation: summary

- Using similar ideas, we can estimate the size of projection, deduplication, 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
- Not covered: better estimation using **histograms** and **machine learning**

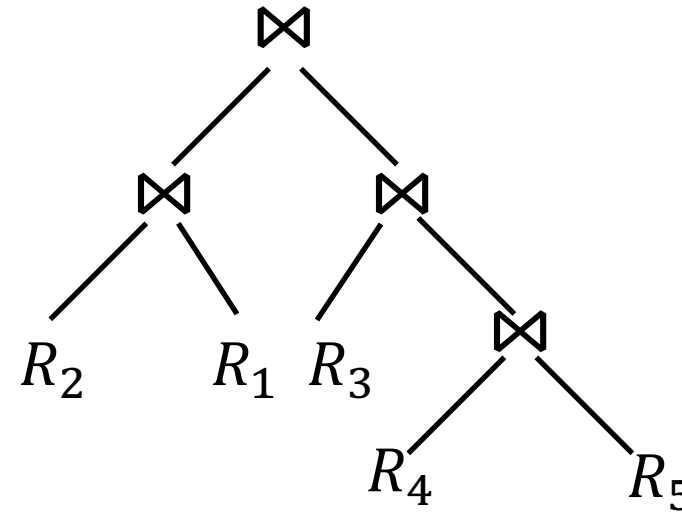


# Outline

- System view of query processing
  - Logical plan and physical plan
- Cost calculation of the physical plan
  - Cardinality estimation
- **Search space and search strategy**
  - Transformation rules
  - Other heuristics

# Search space

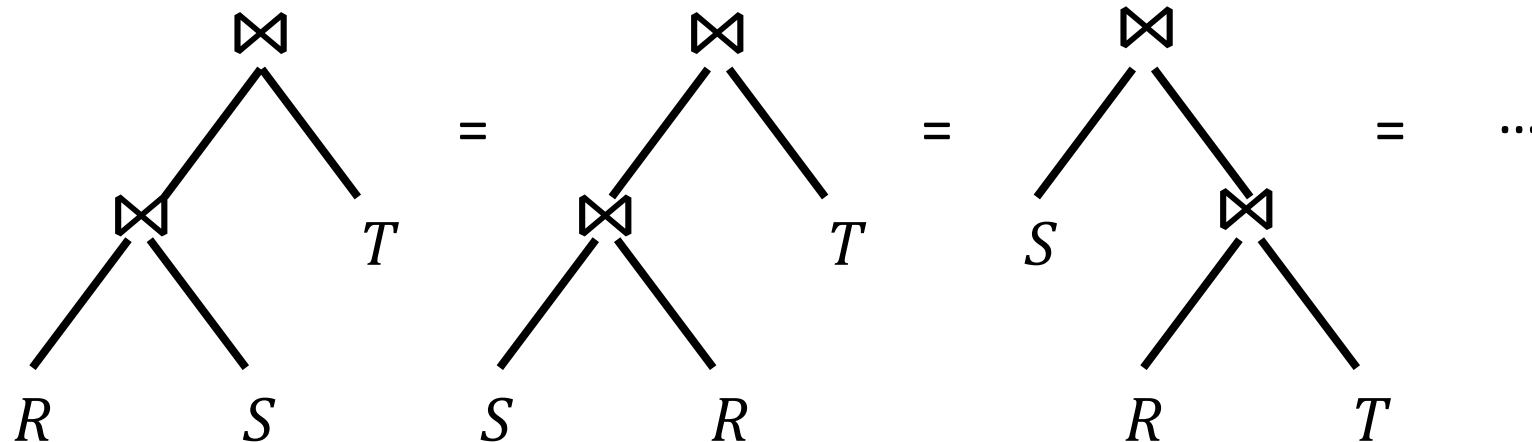
- Huge!
- “Bushy” plan example:



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

# Plan enumeration in relational algebra

- Do we need to exam all the logical plans?
  - No
- Apply relational algebra equivalences to find a cheaper logical plan
  - ☞ Join reordering:  $\times$  and  $\bowtie$  are associative and commutative (except column ordering, but that is unimportant)



# More relational algebra equivalences

- Convert  $\sigma_p$ - $\times$  to/from  $\bowtie_p$ :  $\sigma_p(R \times S) = R \bowtie_p S$ 
  - Example:  $\sigma_{\text{User.uid}=\text{Member.uid}}(\text{User} \times \text{Member}) = \text{User} \bowtie \text{Member}$
- Merge/split  $\sigma$ 's:  $\sigma_{p_1}(\sigma_{p_2} R) = \sigma_{p_1 \wedge p_2} R$ 
  - Example:  $\sigma_{\text{age} > 20}(\sigma_{\text{pop} > 0.5} \text{User}) = \sigma_{\text{age} > 20 \wedge \text{pop} > 0.5} \text{User}$
- Merge/split  $\pi$ 's:  $\pi_{L_1}(\pi_{L_2} R) = \pi_{L_1} R$ , where  $L_1 \subseteq L_2$ 
  - Example:  $\pi_{\text{age}}(\pi_{\text{age}, \text{pop}} \text{User}) = \pi_{\text{age}}(\text{User})$

# More relational algebra equivalences

- Push down/pull up  $\sigma$ :

$$\sigma_{p \wedge p_r \wedge p_s} (R \bowtie_{p'} S) = (\sigma_{p_r} R) \bowtie_{p \wedge p'} (\sigma_{p_s} S), \text{ where}$$

- $p_r$  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
- Example

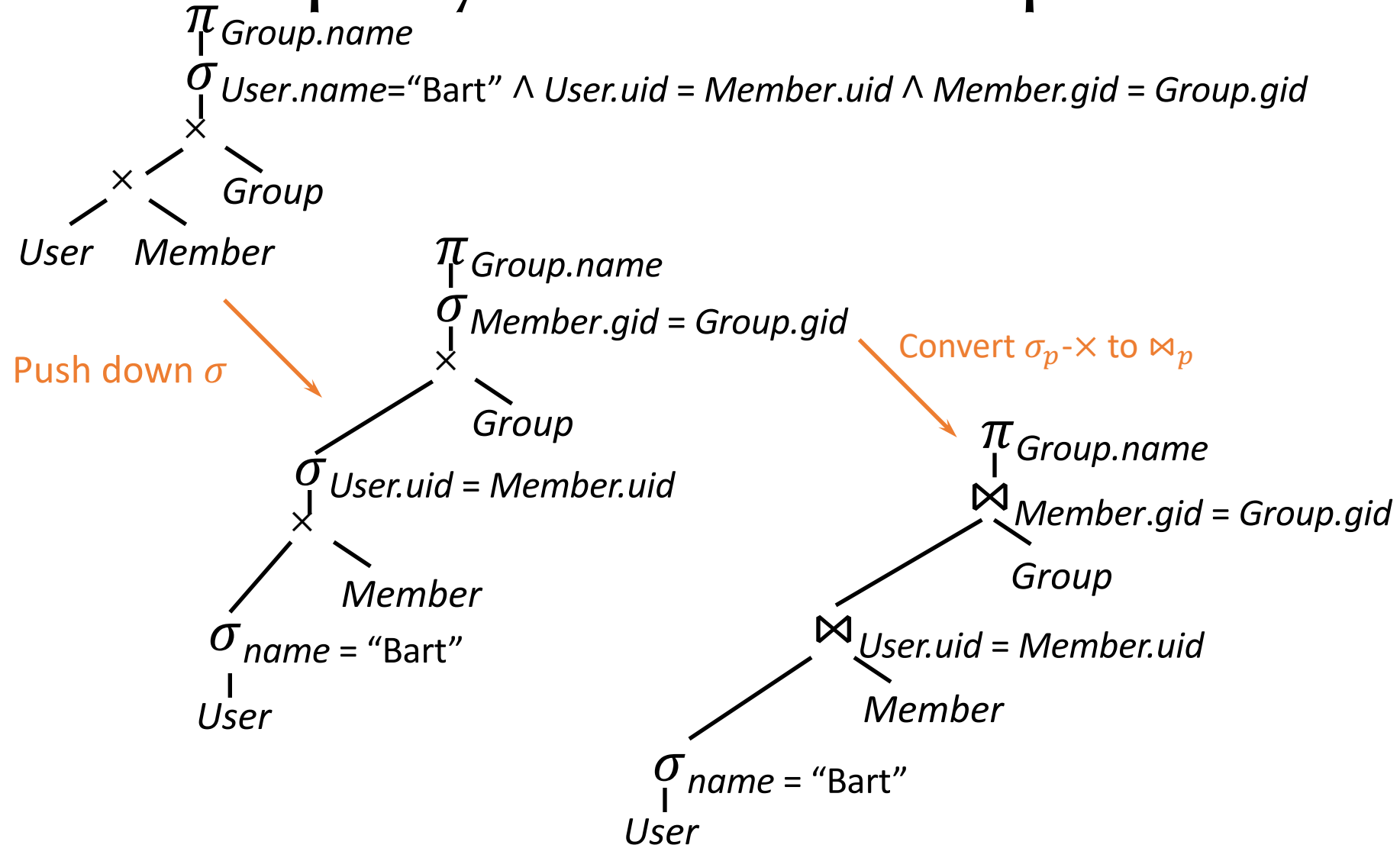
$$\begin{aligned} & \sigma_{U1.name=U2.name \wedge U1.pop>0.5 \wedge U2.pop > 0.5} (\rho_{U1} User \bowtie_{U1.uid \neq U2.uid} \rho_{U2} User) \\ &= ( \sigma_{pop>0.5} (\rho_{U1} User) ) \bowtie_{U1.uid \neq U2.uid \wedge U1.name=U2.name} ( \sigma_{pop>0.5} (\rho_{U2} User) ) \end{aligned}$$

# More relational algebra equivalences

- Push down  $\pi$ :  $\pi_L(\sigma_p R) = \pi_L(\sigma_p(\pi_{LL'} R))$ , where
  - $L'$  is the set of columns referenced by  $p$  that are not in  $L$
  - Example:
    - $\pi_{age}(\sigma_{pop>0.5} User) = \pi_{age}(\sigma_{pop>0.5}(\pi_{age,pop} User))$
- 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? Reduce the size of intermediate results
    - Why not?
  - Join smaller relations first, and avoid cross product
    - Why? Reduce the size of intermediate results
    - Why not?
  - Convert the transformed logical plan to a physical plan (by choosing appropriate physical operators)
- If you need to store the intermediate results, than plans which store large numbers of intermediate results should be avoided

# 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  
Each block may be optimized, but there may be some operations at the very top which can be pushed down to reduce the size of the intermediate results.
    - Even if the plan is optimal for each block, it may not be optimal for the entire query  
Ask professor for an example of this
  - Unnest query: convert subqueries/views to joins
- ☞ We can then deal with each select-project-join(-aggregation) block
- 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;`
  - Wrong—
- `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%';
```
- New subquery is inefficient (it computes the size for every group)
- Suppose a group is empty?

# 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

# Summary

- System view of query processing
  - Logical plan and physical plan
- Cost calculation of the physical plan
  - Cardinality estimation
- Search space and search strategy
  - Transformation rules
  - Other heuristics