Lecture 16: Query Processing & Optimization

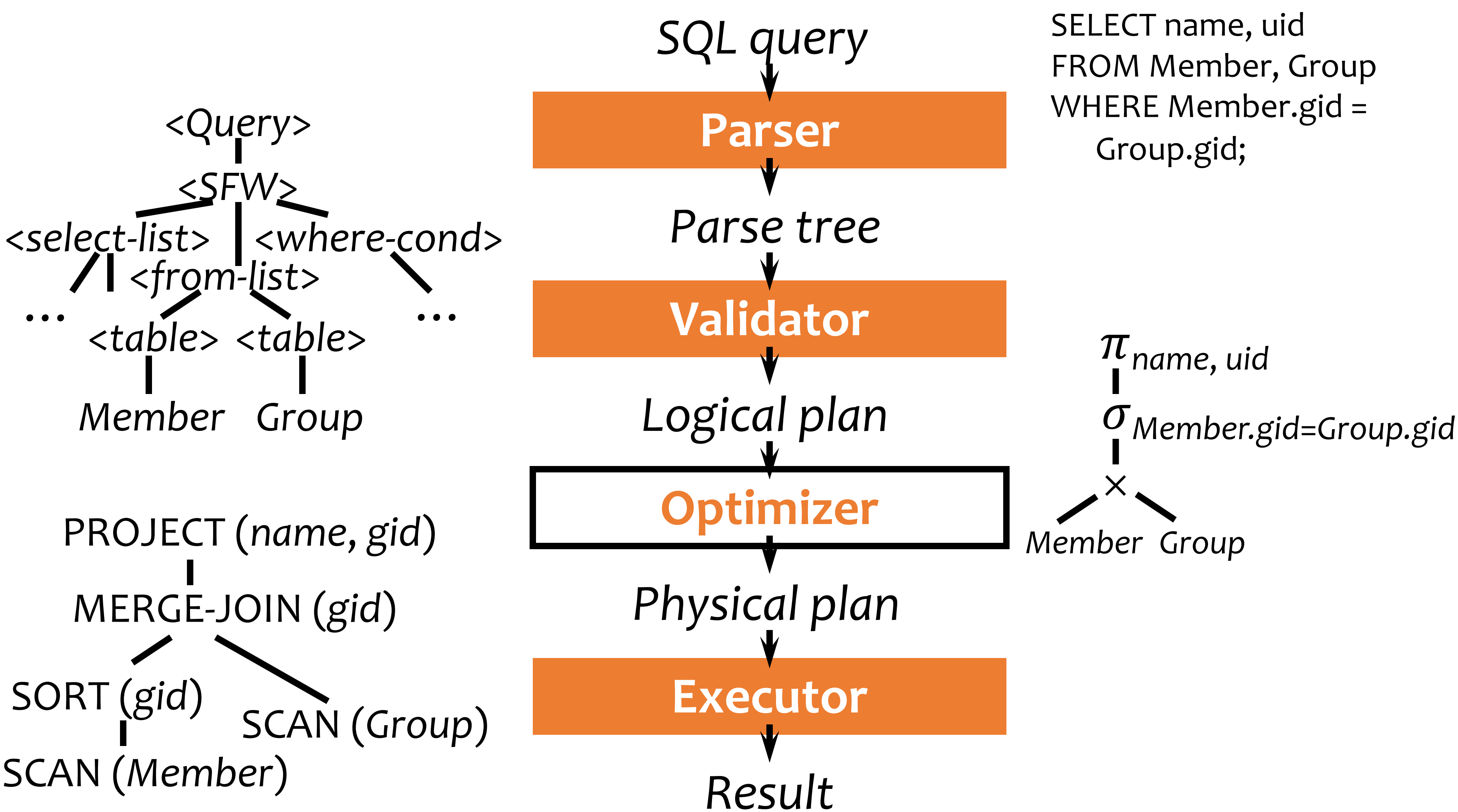
CS348 Spring 2025:
Introduction to Database Management

Instructor: **Xiao Hu**
Sections: 001, 002, 003

Announcements

- Milestone 2 of group project
  - Due today!

A query's trip through the DBMS



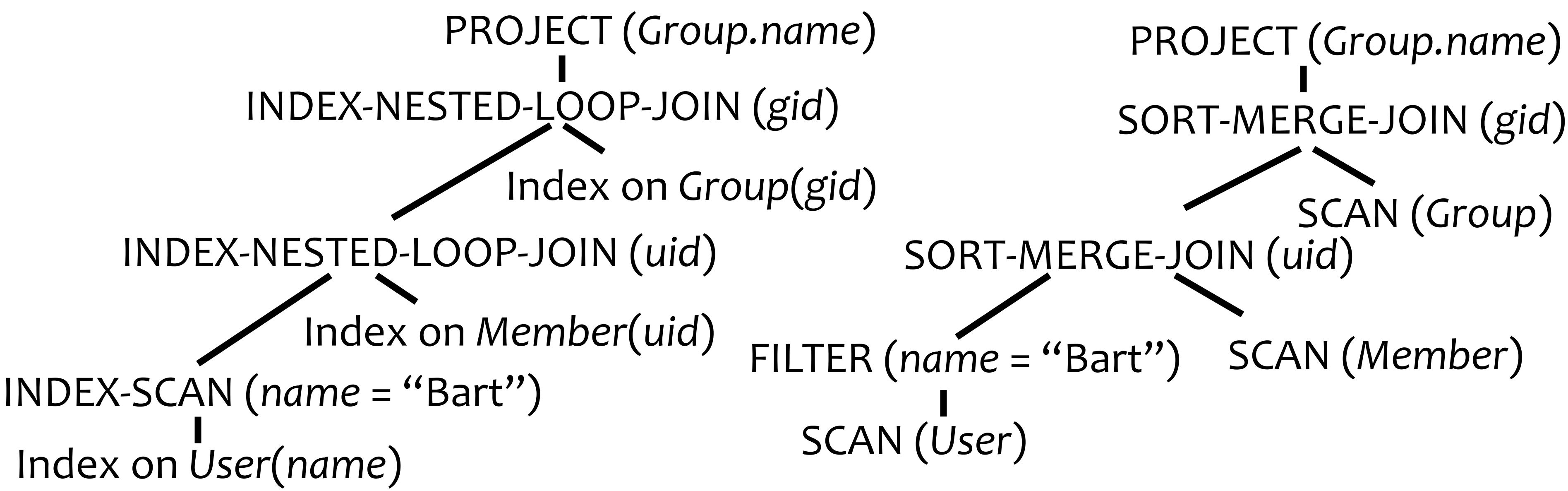
Physical plan

- A complex query may involve multiple tables and various query execution algorithms
  - E.g., table scan, basic & block nested-loop join, index nested-loop join, sort-merge join

- 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 a number of input tables/streams and produces a single output table/stream

(Recap) Physical plans

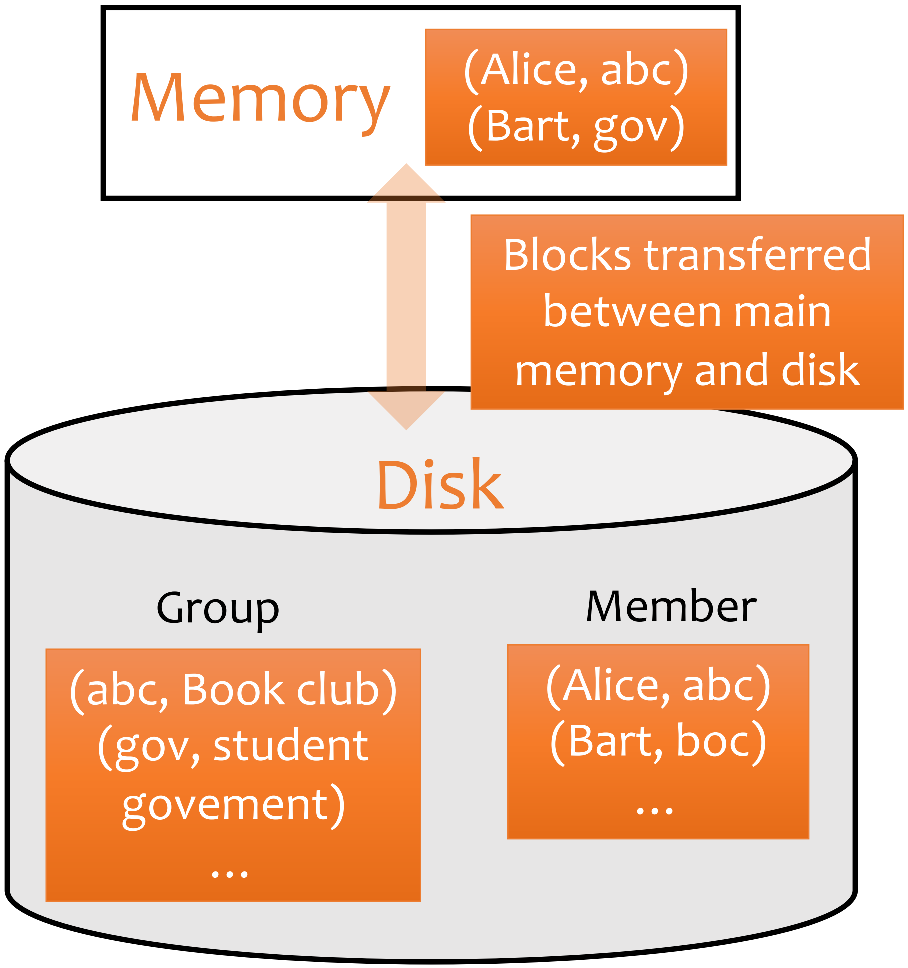
SELECT Group.name
FROM User, Member, Group
WHERE User.name = 'Bart'
AND User.uid = Member.uid AND Member.gid = Group.gid;



- Many physical plans for a single query

Outline

- Scan
  - Table scan
  - Selection, Duplicate-preserving projection
  - Nested-loop join
- Sort
  - External merge sort
  - Duplicate elimination, Grouping and Aggregation
  - Sort-merge join, Union (set), Difference, Intersection
- Hash
- Index



Notation and Assumption

- Relations: R, S
- Tuples: r, s
- Number of tuples: $|R|$, $|S|$
- Number of disk blocks: $B(R)$, $B(S)$
- Number of memory blocks available: M
- Cost metric
  - Number of I/O's
  - Memory requirement
- Not counting the cost of writing the result out
  - Same for any algorithm!
  - Consumed by subsequent operators

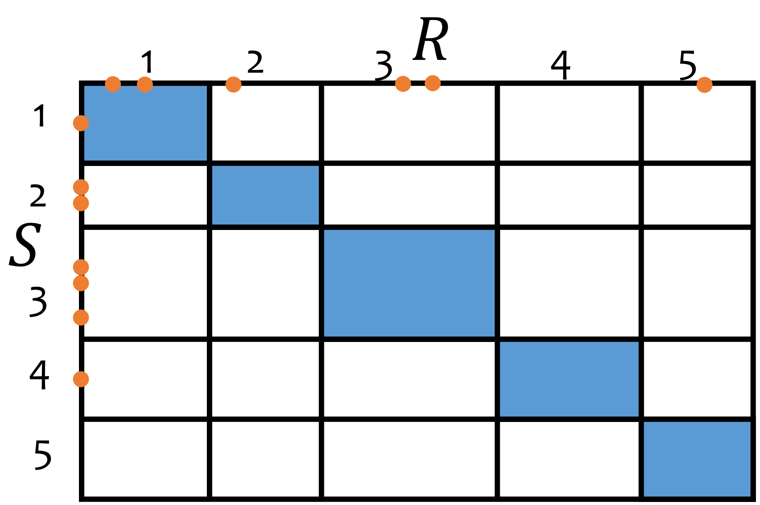
Hashing-based algorithms

http://global.rakuten.com/en/store/citygas/item/041233/

Hash join

$R \bowtie_{R.A=S.B} S$

- Main idea
  - Partition R and S by hashing their join attributes, and then consider corresponding partitions of R and S
  - If $r.A$ and $s.B$ get hashed to different partitions, they don't join

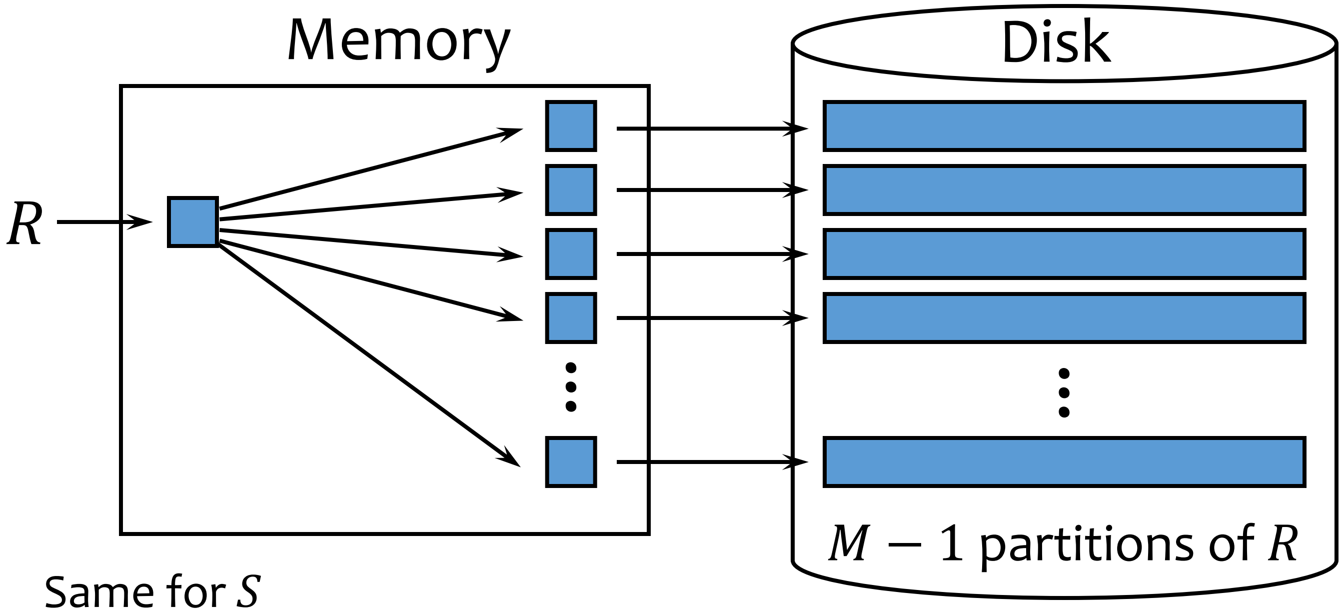


Nested-loop join considers all slots

Hash join considers only those along the diagonal!

Partitioning phase

- Partition R and S according to the same hash function on their join attributes

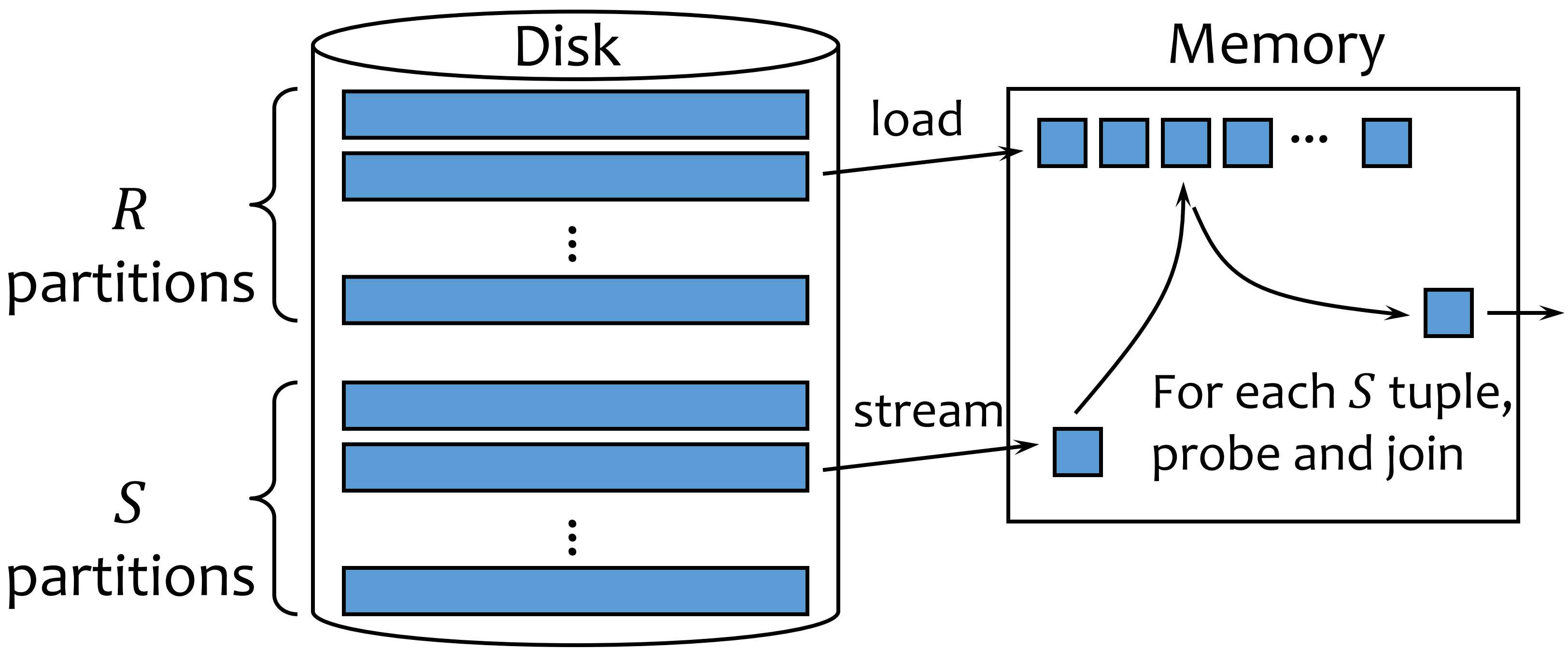


Same for S

Each partition has a size of B(R)/(M-1)

Probing phase

- Read in each partition of R, stream in the corresponding partition of S, join
  - Typically build a hash table for the partition of R
    - Not the same hash function used for partition, of course!



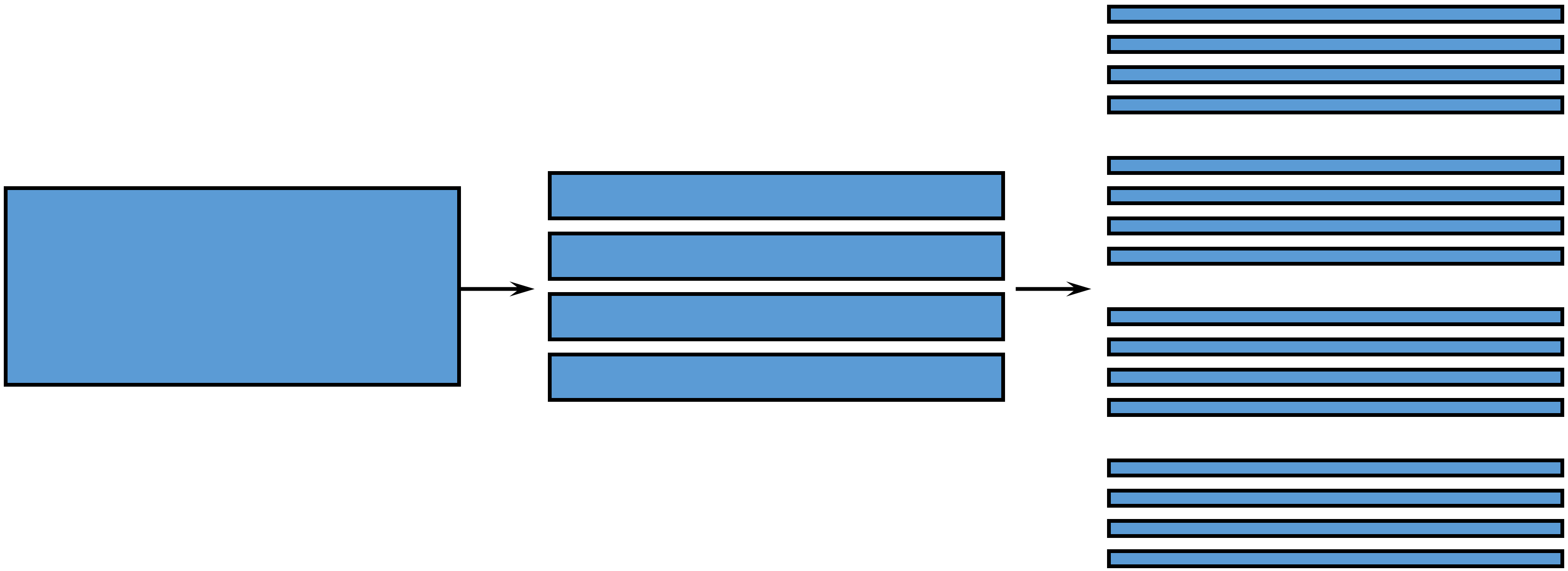
Performance of (two-pass) hash join

- If hash join completes in two phases:
  - I/O's: $3 \cdot \left(B(R) + B(S)\right)$
    - 1st phase: read B(R) + B(S) into memory to partition and write partitioned B(R) +B(S) to disk
    - 2nd phase: read B(R) + B(S) into memory to merge and join
  - Memory requirement:
    - In the probing phase, we should have enough memory to fit one partition of *R*: $M - 1 > \frac{B(R)}{M-1}$
    - $M > \sqrt{B(R)} + 1$
    - We can always pick R to be the smaller relation, so:

$$M > \sqrt{\min\left(B(R), B(S)\right)} + 1$$

Generalizing for larger inputs

- What if a partition is too large for memory?
    - Read it back in and partition it again!
    - Re-partition $O\left(\log_M B(R)\right)$ times



Hash join versus SMJ

(Assuming two-pass)

- I/O's: same
- Memory requirement: hash join is lower
  - $\sqrt{\min\left(B(R), B(S)\right)} + 1 < \sqrt{B(R) + B(S)}$
  - Hash join wins when two relations have very different sizes
- Other factors
  - Hash join performance depends on the quality of the hash
    - Might not get evenly sized buckets
  - SMJ can be adapted for inequality join predicates
  - SMJ wins if R and/or S are already sorted
  - SMJ wins if the result needs to be in sorted order

Hash join vs. SMJ: multi-pass

For both, let I denote "input"

- # passes is $O\left(\log_M\left(\frac{B(I)}{M}\right)\right) = O(\log_M B(I))$
  - Assuming hash function is good enough and there is no severe data skew
- Overall I/Os is $O(B(I) \cdot \log_M B(I))$
  - Assuming no external-memory mini nested loops

Compare with I/O lower bound on external permuting

- Rearranging $B(I)$ elements according to given permutation takes $\Omega(\min(|I|, B(I) \cdot \log_M B(I)))$ I/Os

Duality of sort and hash

- Divide-and-conquer paradigm
  - Sorting: physical division, logical combination
  - Hashing: logical division, physical combination
- Handling very large inputs
  - Sorting: multi-level merge
  - Hashing: recursive partitioning
- I/O patterns
  - Sorting: sequential write, random read (merge)
  - Hashing: random write, sequential read (partition)

Other hash-based algorithms

- Union (set), difference, intersection
  - More or less like hash join
- Duplicate elimination
  - Check for duplicates within each partition/bucket
- Grouping and aggregation
  - Apply the hash functions to the group-by columns
  - Tuples in the same group must end up in the same partition/bucket
  - Keep a running aggregate value for each group
    - Just like in the sorting case, this trick may not always work

Outline

- Scan
  - Table scan
  - Selection, Duplicate-preserving projection
  - Nested-loop join
- Sort
  - External merge sort
  - Duplicate elimination, Grouping and Aggregation
  - Sort-merge join, Union (set), Difference, Intersection
- Hash
  - Hash join, union (set), difference, intersection, duplicate elimination, grouping and aggregation
- Index

Index-based algorithms

http://i1.trekearth.com/photos/28820/p2270994.jpg

Selection using index

- Equality predicate: $\sigma_{A=v}(R)$
  - Use an ISAM, B^+-tree, or hash index on $R(A)$
- Range predicate: $\sigma_{A>v}(R)$
  - Use an ordered index (e.g., ISAM or B^+-tree) on $R(A)$
  - Hash index is not applicable

Index versus table scan

Situations where index clearly wins:

- Index-only queries which do not require retrieving actual tuples
  - Example: $\pi_A\left(\sigma_{A>v}(R)\right)$
- Primary index clustered according to search key
  - One lookup leads to all result tuples in their entirety

Index versus table scan (cont'd)

BUT(!):

- Consider $\sigma_{A>v}(R)$ and a secondary, non-clustered index on $R(A)$
  - Need to follow pointers to get the actual result tuples
  - Say that 20% of R satisfies $A > v$
    - Could happen even for equality predicates
  - I/O's for index-based selection: lookup + 20% $|R|$
  - I/O's for scan-based selection: $B(R)$
  - Table scan wins if a block contains more than 5 tuples!

Index nested-loop join

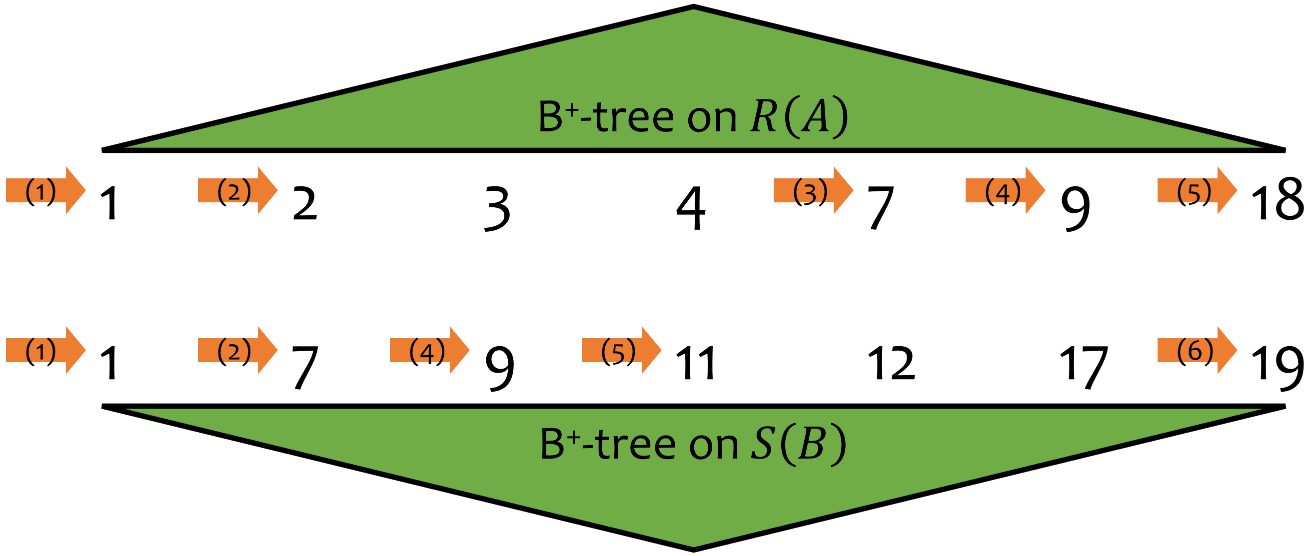
$R \bowtie_{R.A=S.B} S$

- Idea: use a value of $R.A$ to probe the index on $S(B)$
- For each block of R, and for each r in the block:
  Use the index on $S(B)$ to retrieve s with $s.B = r.A$
    Output rs
- I/O's: $B(R) + |R| \cdot (\text{index lookup})$
  - Typically, the cost of an index lookup is 2-4 I/O's
  - Beats other join methods if $|R|$ is not too big
    - And if the index on $S(B)$ is secondary, not too many S rows join with each r
  - Better pick R to be the smaller relation
- Memory requirement: 3

Zig-zag join using ordered indexes

$R \bowtie_{R.A=S.B} S$

- Idea: use the ordering provided by the indexes on $R(A)$ and $S(B)$ to eliminate the sorting step of sort-merge join
- Use the larger key to probe the other index
  - Possibly skipping many keys that don't match



Additional tricks

- Lots of index lookups across the key or address space?
  - "Pre-condition" them to get better caching behavior
    - Recall similar ideas we've seen earlier?

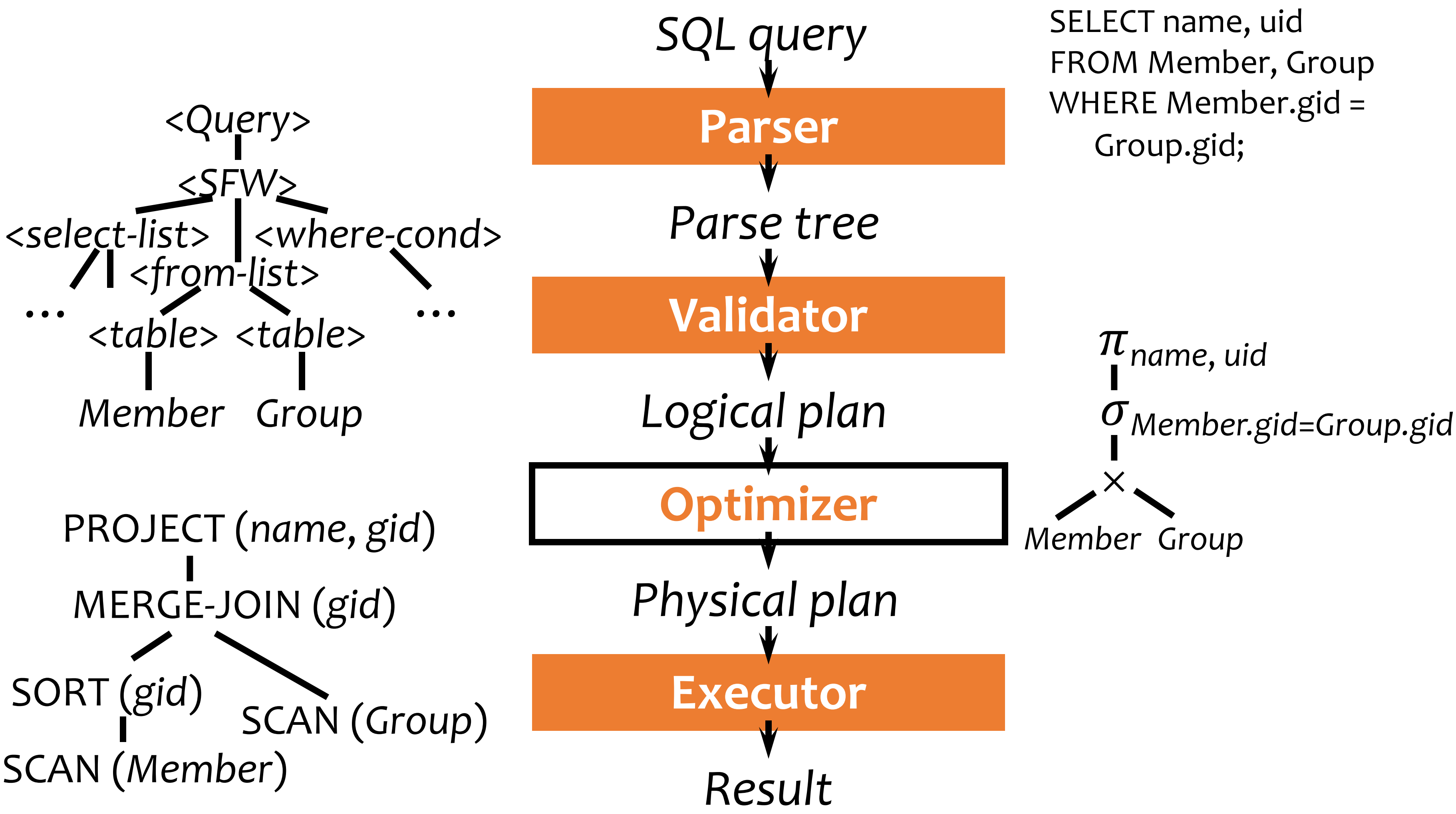
E.g.: $R \bowtie_{R.A=S.B} S$: use index nested-loop with secondary index on $S(B)$

- Sort R by $R.A$ $\Rightarrow$ consecutive index lookups are more likely to share search paths
- Don't fetch joining S rows one at a time; collect a bunch of record ids, and do a "batch" retrieval from data file
  - Option 1: sort record ids by their physical address
  - Option 2 (PostgreSQL "bitmap index scan"): build a bitmap indicating which data blocks hold relevant rows
    - Filter out false positives once tuples are retrieved
  - Both support efficient AND/OR of individually sarg'd conditions

Summary of techniques

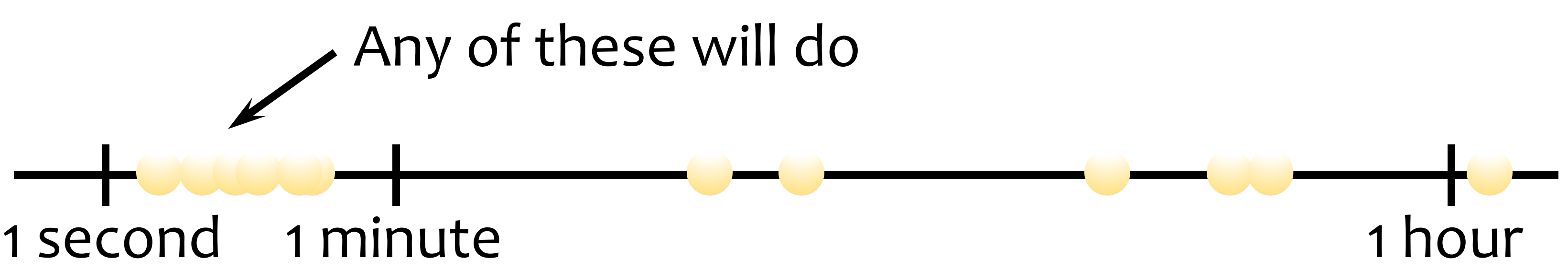
- Scan
  - Table scan
  - Selection, Duplicate-preserving projection
  - Nested-loop join
- Sort
  - External merge sort
  - Duplicate elimination, Grouping and Aggregation
  - Sort-merge join, Union (set), Difference, Intersection
- Hash
  - Hash join, union (set), difference, intersection, duplicate elimination, grouping and aggregation
- Index
  - Selection, index nested-loop join, zig-zag join

Back to the trip



Query optimization

- Why query optimization?
  - Many different ways of processing the same query
    - A query can have multiple logical plans (in RA)
    - A logical plan can have numerous physical plans
      - Scan? Sort? Hash? Index?
  - Different ways make different assumptions about data have different performance
- Often, the goal is not getting the optimum plan, but instead avoiding the horrible ones



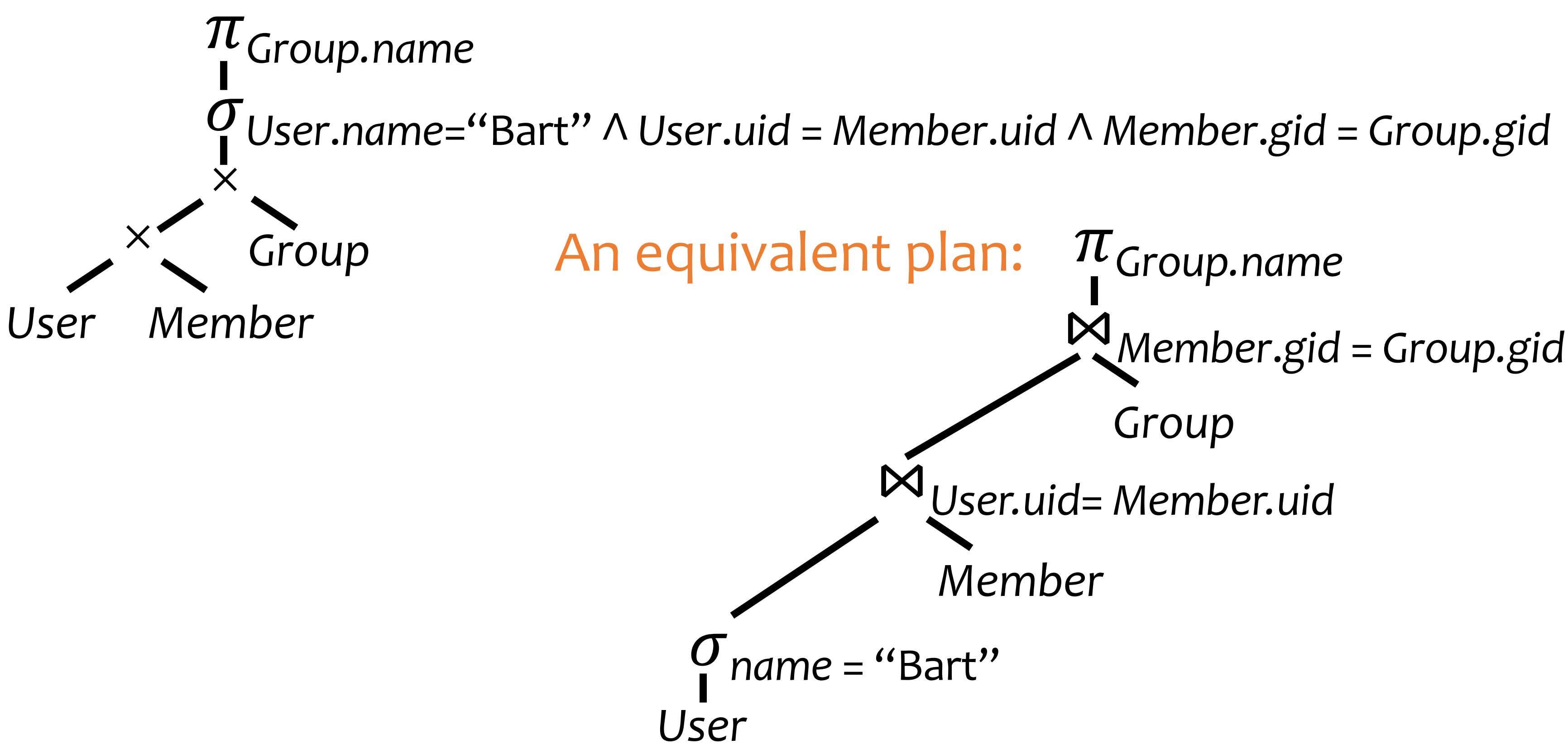
Outline

- Search space
    - What are the possible equivalent logical plans?
    - For each logical plan, what are the possible physical plans? (Lecture 16)

- Search strategy
    - Rule-based strategy
    - Cost-estimation-based strategy

Logical plan

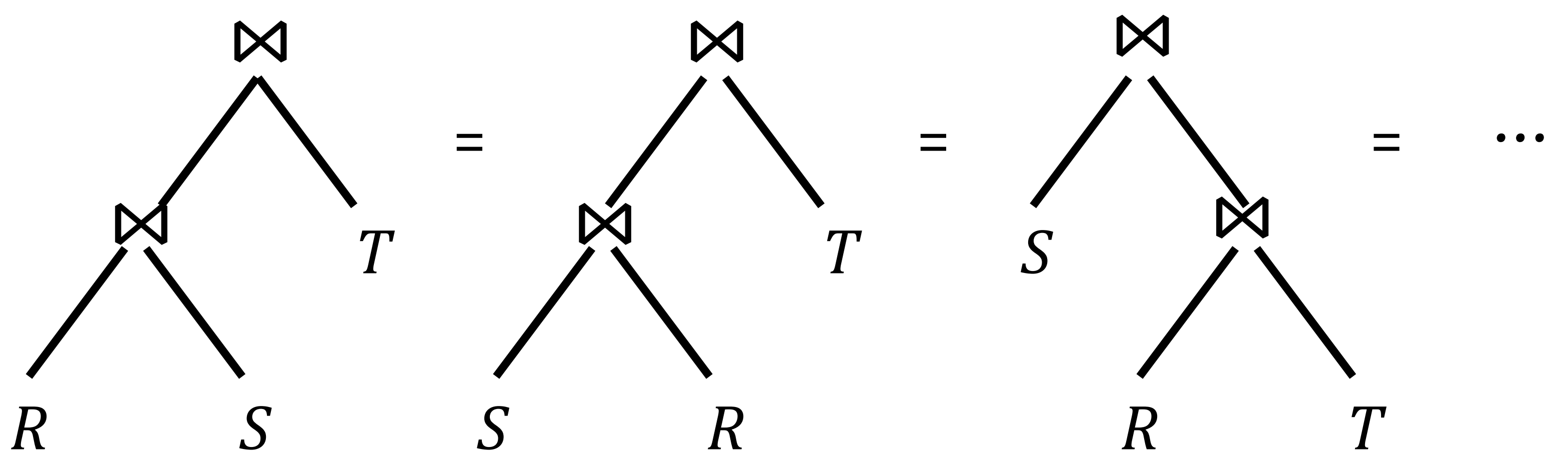
- Nodes are logical operators (often relational algebra operators)
- There are many equivalent logical plans



Algebraic equivalences

- Apply algebraic equivalences in relational and/or algebra to systematically transform a plan to new ones

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



More Algebraic equivalences

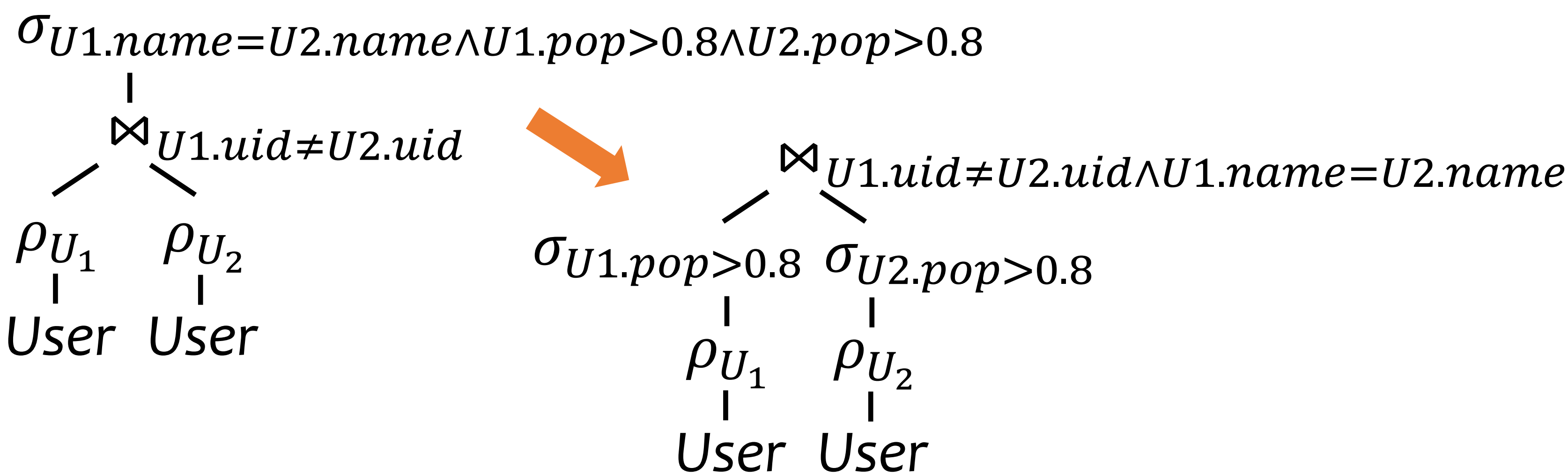
- Convert σ_p-$\times$ to/from $\bowtie_p$: $\sigma_p(R \times S) = R \bowtie_p S$
- Merge/split σ's: $\sigma_{p_1}\left(\sigma_{p_2} R\right) = \sigma_{p_1 \wedge p_2} R$
- Merge/split π's: $\pi_{L_1}\left(\pi_{L_2} R\right) = \pi_{L_1 \cup L_2} R$

More algebraic equivalences

- Push down/pull up σ:

$$\sigma_{p \wedge p_R \wedge p_S}(R \bowtie_{p'} S) = (\sigma_{p_R} R) \bowtie_{p \wedge p'} (\sigma_{p_S} S)$$

  - p_R involves only R;
  - p_S involves only S;
  - p and p' involve both R and S



More algebraic equivalences

- Push down π:

$$\pi_L\left(\sigma_p R\right) = \pi_L\left(\sigma_p\left(\pi_{L \cup L'} R\right)\right)$$

  - L' is the set of columns referenced by p

$\pi_{name,age}$
|
$\sigma_{pop>0.8 \wedge age<18}$
|
$User$

→

$\pi_{name,age}$
|
$\sigma_{pop>0.8 \wedge age<18}$
|
$\pi_{name,age,pop}$
|
$User$

☞Above works under both set and bag semantics

- For bag semantics, π above preserves duplicates