NYU TANDON SCHOOL OF ENGINEERING

CS6083: Database Systems

Join Processing in Databases

Torsten Suel
CSE Department
NYU Tandon School of Engineering

Query processing

SQL query

↓ **Parse & Translate**

Relational Algebra Expression

↓ **Optimization**

Query execution plan

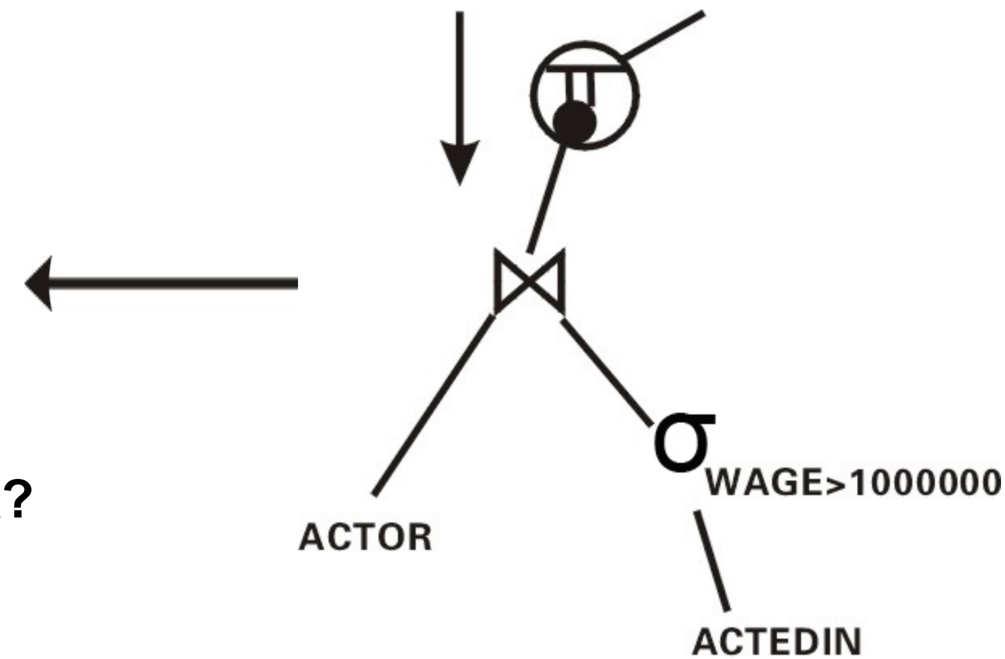
Issues:

- How to execute (or implement) an operator:
  - Select
  - Join
  - Project
  - Group by
  - …
- How to optimize networks of operators (plans)
  - Rule based
  - Cost based
    - Disk seeks
    - Block read/write
    - CPU
    - Space

SELECT NAME
FROM ACTOR, ACTEDIN
WHERE AGE > 1000000 AND ACTOR.AID = ACTEDIN.AID;

Execution plan:

- What order?
- What Join?
- What select? index?



Join Implementation

- Many algorithms
- Trivial algorithm:
  - “Nested loop join”
- Optimized algorithms:
  - Blocked nested-loop join
  - Index nested-loop join
  - Sorted merge join (equality joins only)
  - Hash join (equality joins only)
- Block I/O model for simplicity

Examples:

- 1. SELECT C.NAME FROM C, A WHERE C.CID = A.CID AND A.NUM = 1245;
  - "First select on account #"
- 2. SELECT C.NAME FROM C, A WHERE C.CID = A.CID AND A.NUM = 2 * C.ZIPCODE;
  - "Cannot select on account # first"
- 3. SELECT C.NAME FROM C, A WHERE C.CID = A.CID;
  - "Big join": All customers who have an account
- 4. SELECT C.NAME FROM C, A WHERE C.CID > A.CID;
  - "Semi-join" (Bad example?)
- 5. WHERE COND(C.CID, A.CID) e.g.: WHERE 7 * C.CID > √A.CID
  - "General" (Silly example)

Examples

- Join large and small relation ➔
  - Does small one fit in memory ?
  - Does index exist on large one ?

- Join two large relations ➔

- Both large/small and large/large case important
- But small/small does not matter
- Equality joins (almost all of the time)

Setup:

- Relation R, S   (R < S)
- Equi-join: R JOIN S ON R.SID = S.SID
- Inequality-join: R JOIN S ON R.GPA < S.GPA
- More general ?

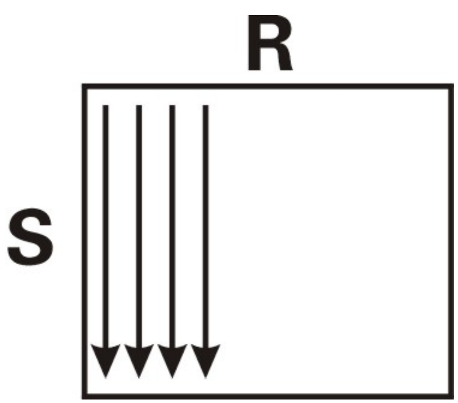
- **<u>(Tuple-Oriented) Nested Loop Join:</u>** **relations R, S**
  - **For each tuple r in R**
    - **For each tuple s in S**
      - **If condition (r, s) true**
        - **add (r, s) to result**

- **<u>In external memory (on disk):</u>**
  - **For each tuple r in R**
    - **Scan the entire relation S, and check condition (r, s) for s $\in$ S**
  - **➔ S is scanned many times**
    - **Cost : M + P_R * N * M block I/Os**
    - **M = # pages in R N = # pages in S**
    - **P_R = # tuples per page of R**

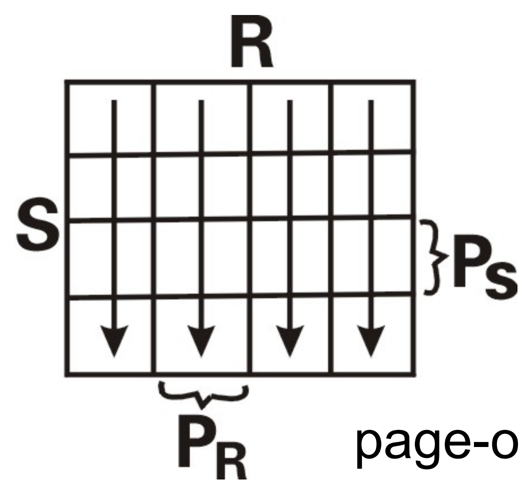
Page-oriented Nested Loop Join

- For each page P_R of R
  - For each page P_S of S
    - Check join condition for all r ∈ P_R and all s ∈ P_S

- Cost: M + M * N block I/Os

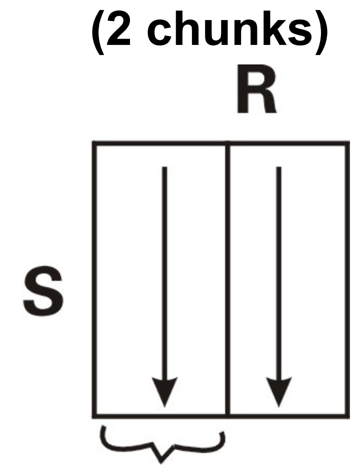


tuple-oriented



page-oriented

Blocked-Nested Loop Join



Partition R into large chunks M_R

for each M_R, scan S

$$M * N / M_R$$

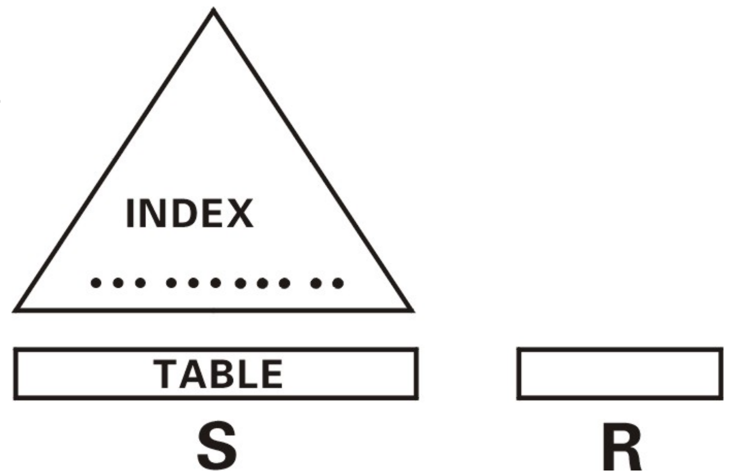
\# Pages that fit in buffer = M_R

Need to divide buffer space between R and S

Index-Based Joins

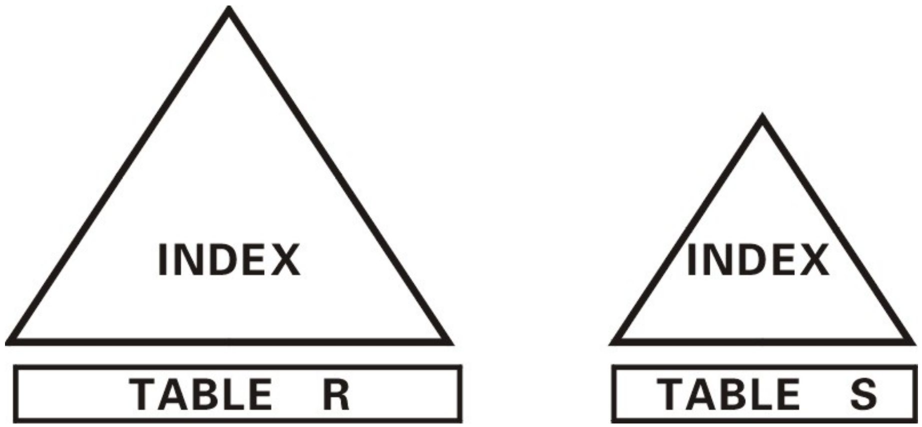
Suppose you have index on S

- For each tuple in R, look in index for matching tuples in S



- Need index on attribute used in join
- Cost: $|R| * (HT + 1) \approx |R| * \lceil \log_d (|S|) \rceil$
  - does it matter if index is clustered?
  - does it matter if index is sparse or dense?

Index on R vs. Index on S



- Lookup in R or in S? (which one is outer relation?)
- Use index into large table
  - $S.lg_d(|R|) < R * \log_d(|S|)$
- More complicated if clustered vs. Non-clustered

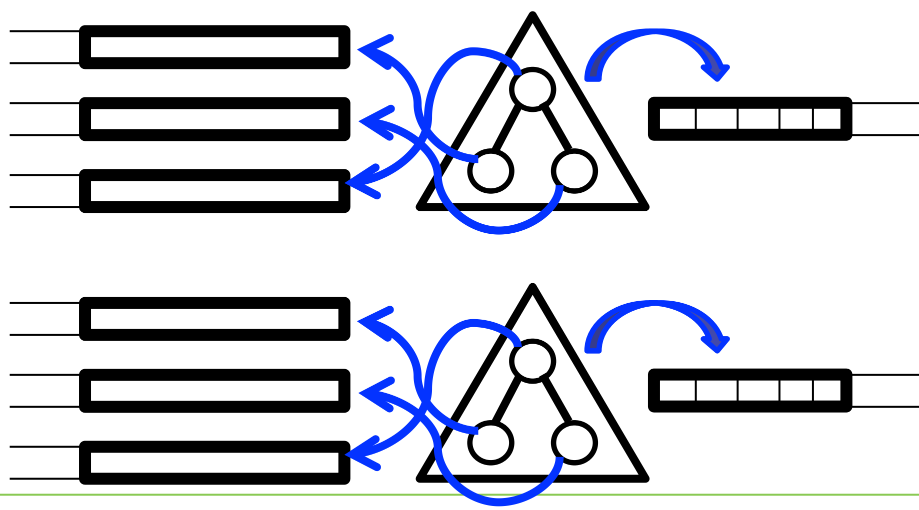
Sort-Merge Join

- Exactly what you would expect
- If not sorted, sort table
- Then scan tables to find matches between tuples

NYU | TANDON SCHOOL OF ENGINEERING

Sort-Merge Join

- Exactly what you would expect
- If not sorted, sort table
- Then scan tables to find matches between tuples
- Perform scans right after merge heap (no extra i/o)



"perform scan inside output buffer"

Hash Join

- Throw both relations into a common hash table
- Or into two hash tables with same hash function
- Examine corresponding buckets of the hash tables
- Tuples that join are in same bucket (equality only)
- Very efficient if each bucket fits in memory

Join Algorithms

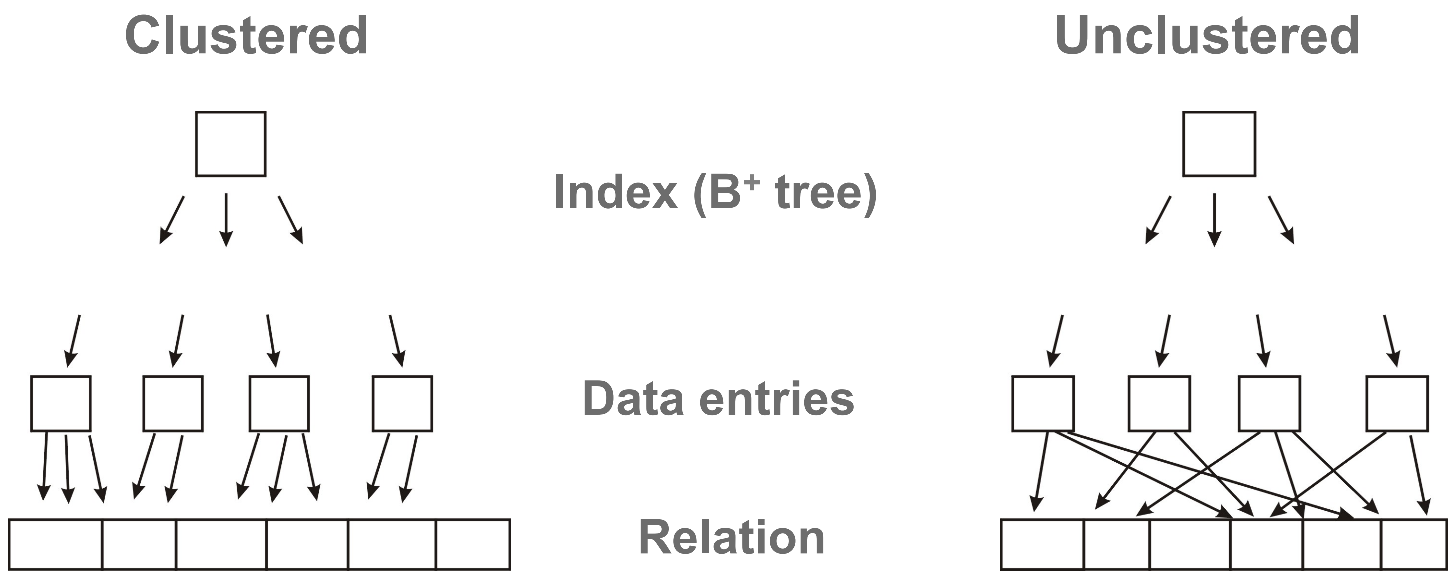
- Blocked Nested-Loop Join
- Index-Based Join
- Sort-Based Join
- Hash-Based Join

- Most common: index-based, along foreign keys
- Next: blocked nested-loop join
- Rare: hash- and sort-based; for very large joins
- Most joins have one very small input relation

Access Path:

- "A way of retrieving records from a relation"
- Scan (all tuples and then filter)

  or
- Use index (for selected tuples)



Join Optimization

- How to best do joins
- Many different solutions
- No “best” solution for all cases
- Optimizer should choose best one for current situation

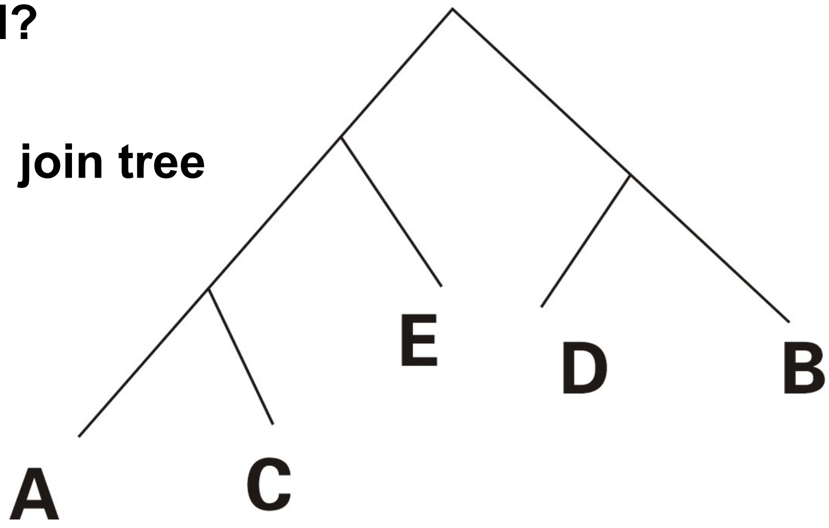
Join Size Estimation

- How large is result of A join B?
  - At most |A| * |B|, usually less
  - Semi-join ≤, ≥ : ~ ½ |A| * |B|
  - General join: ?
  - Equi-join:
    - - Suppose join on key in B: Result ≤ |B|
    - - one-to-one, one-to many, many-to-many

Join Order Optimization

SELECT A.NAME, SUM(AI.WAGE)
FROM A, AI, M
WHERE A.AID = AI.AID AND AI.MID = M.MID
GROUP BY A.NAME

➔ First join A and AI, or AI and M?



Query Evaluation:

SQL query

↓ How to transform

Query plan ("RA")

↓ How to optimize

Optimized query plan

↕ How to put operators together

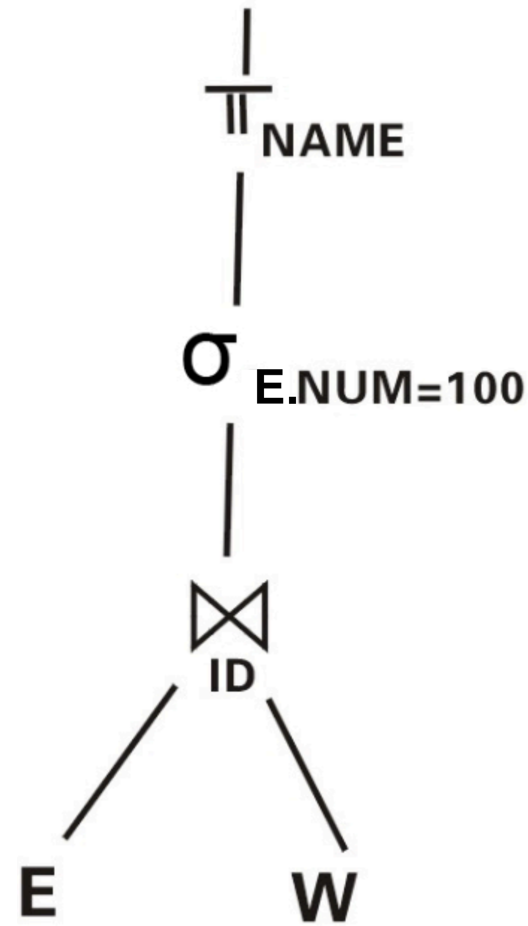
Operator (join, select)

Example:

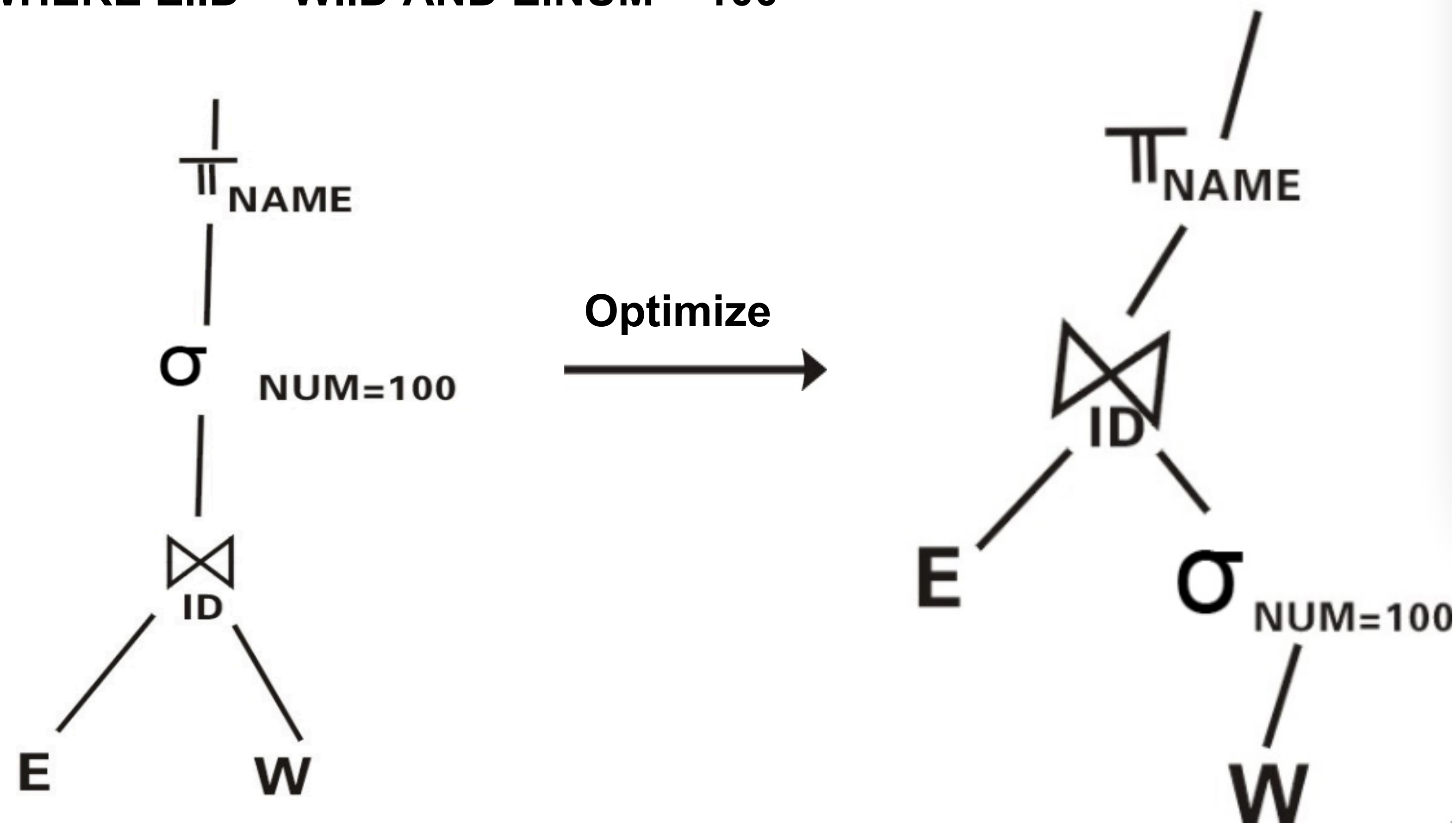
"Names of employees from Dept. 100"

SELECT EMPLOYEE.NAME
FROM EMPLOYEE E, WORKS_IN W
WHERE E.ID = W.ID AND E.NUM = 100

$\longrightarrow$

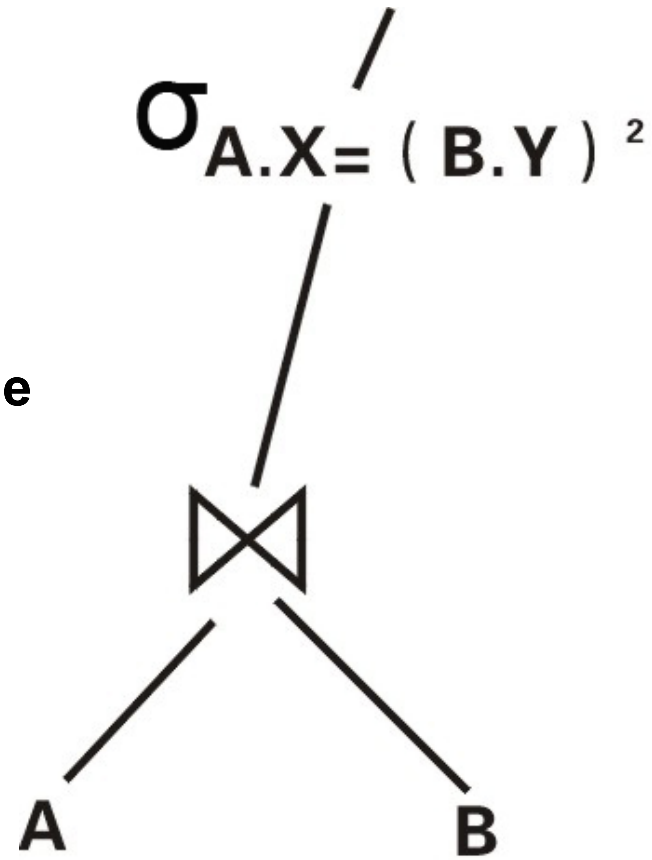


- SELECT EMPLOYEE.NAME FROM EMPLOYEE, WORKS_IN W WHERE E.ID = W.ID AND E.NUM = 100



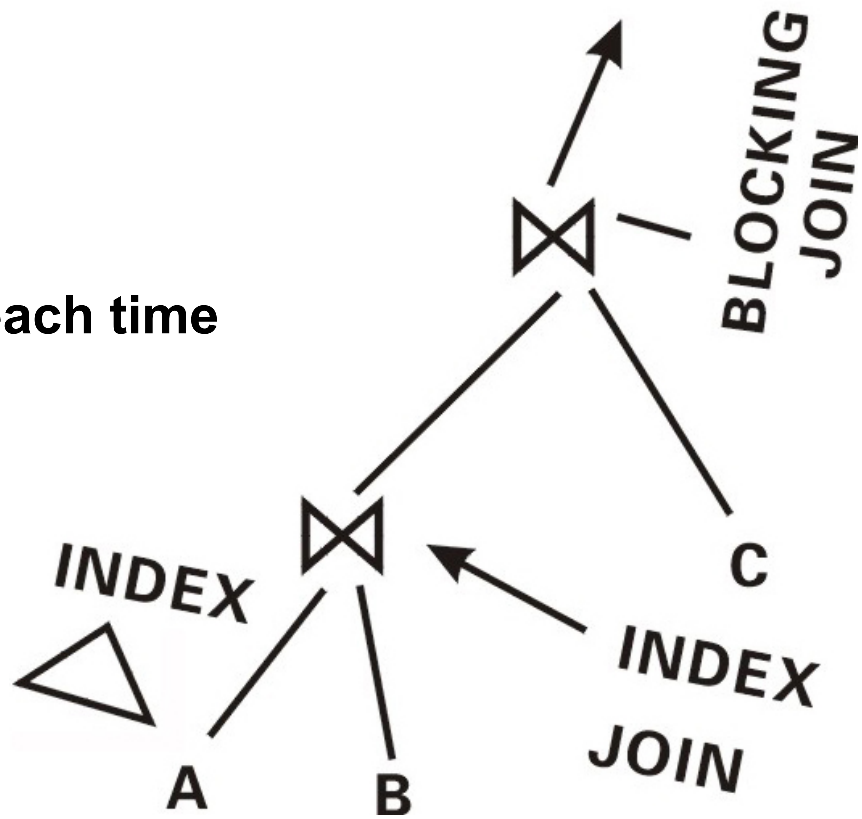
Pipelined Query Execution

- Faster
- Elegant implementation
- Several operators active at each time



Pipelined Query Execution

- Faster
- Elegant implementation
- Several operators active at each time

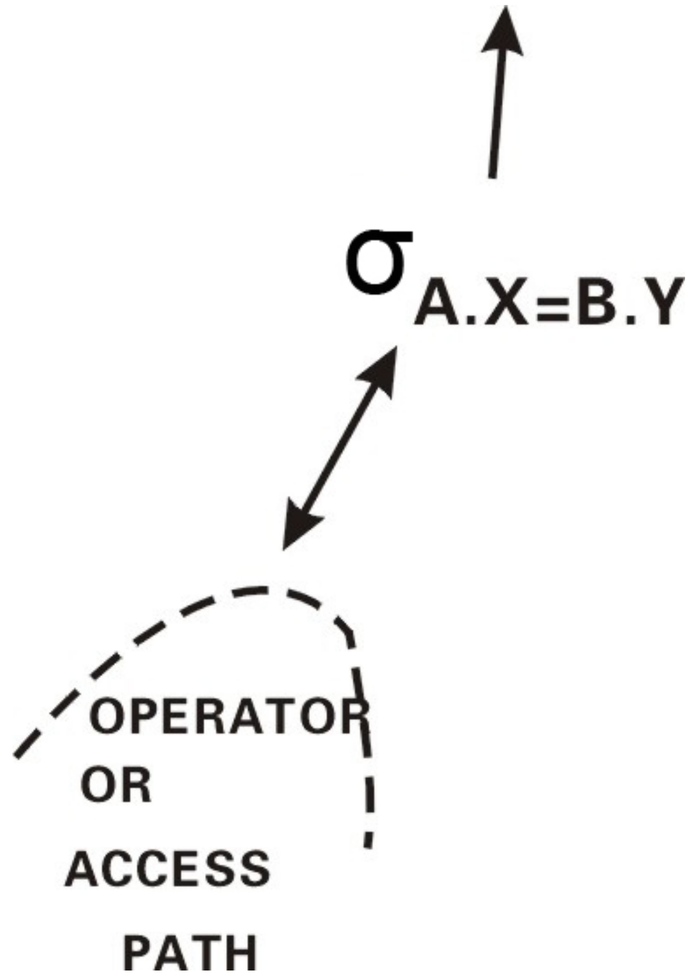


Iterators:

open()

close()

get_next()



- Pipelining / blocking nodes
- "Materializing" data
- Volcano style (GRAEFE)