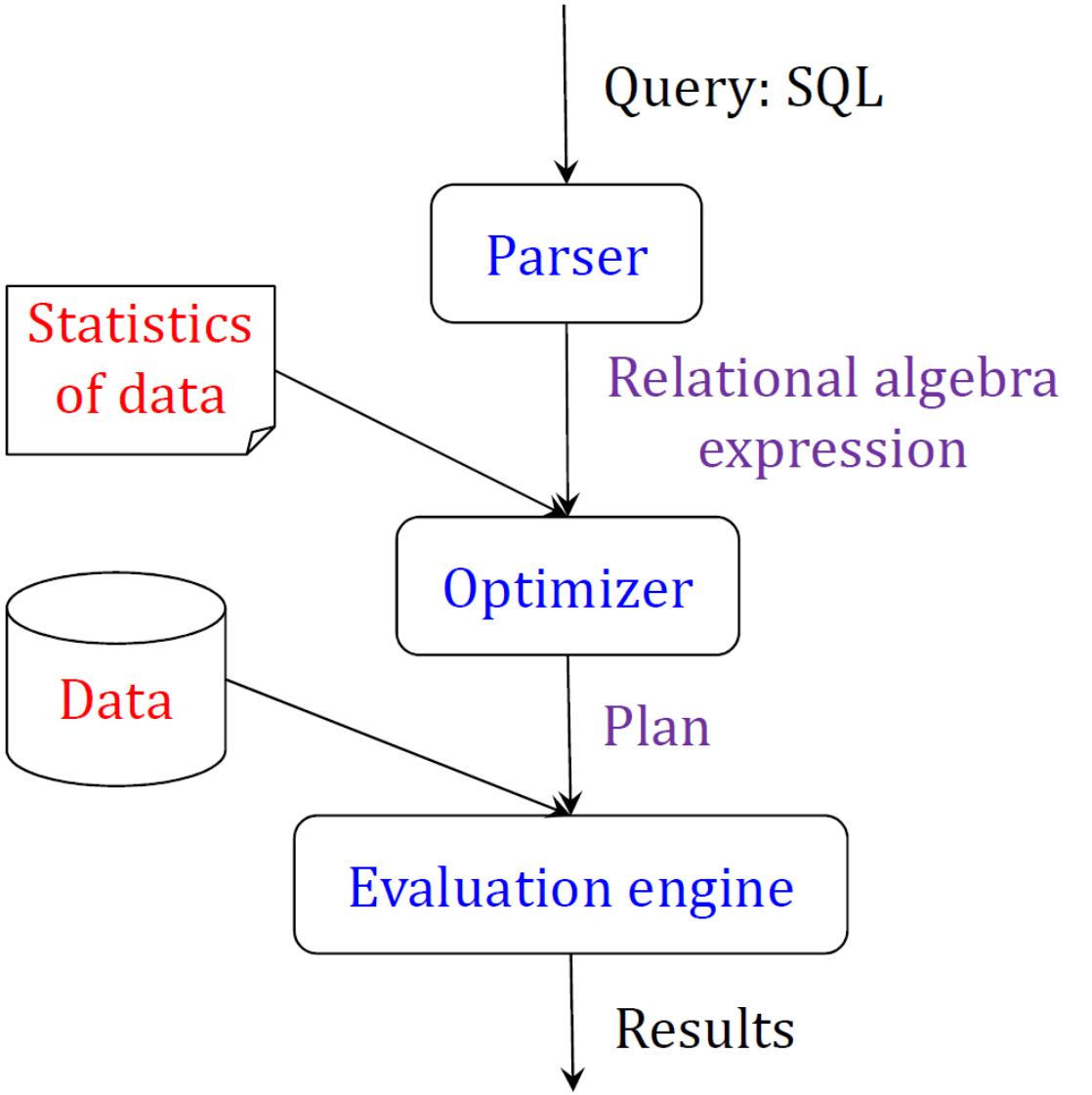
南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

查询处理与查询优化

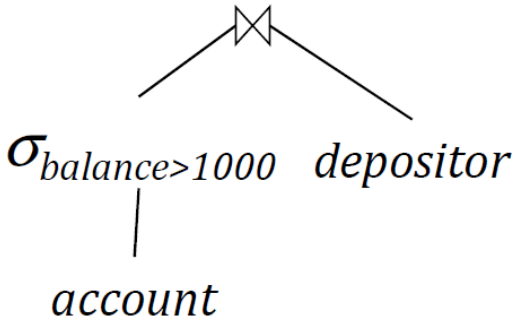
南方科技大学
唐　博
tangb3@sustech.edu.cn

How to process a SQL query

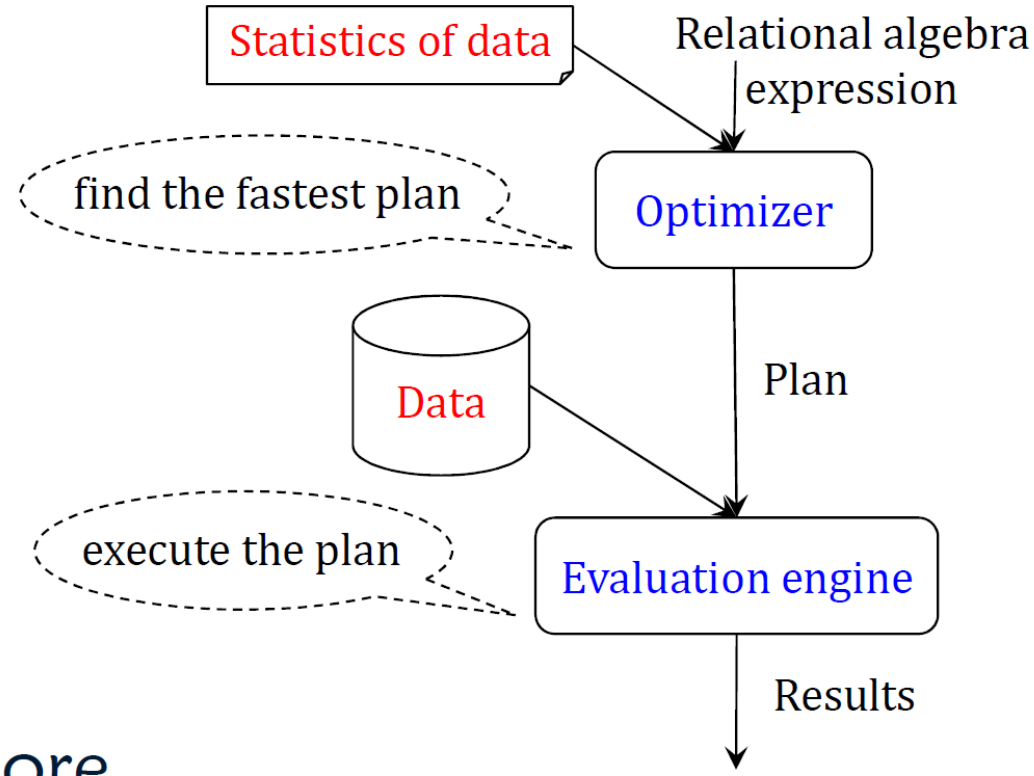


SELECT ...
FROM ...
WHERE ...

$\sigma \ldots (\prod \ldots (\ldots \bowtie \ldots))$



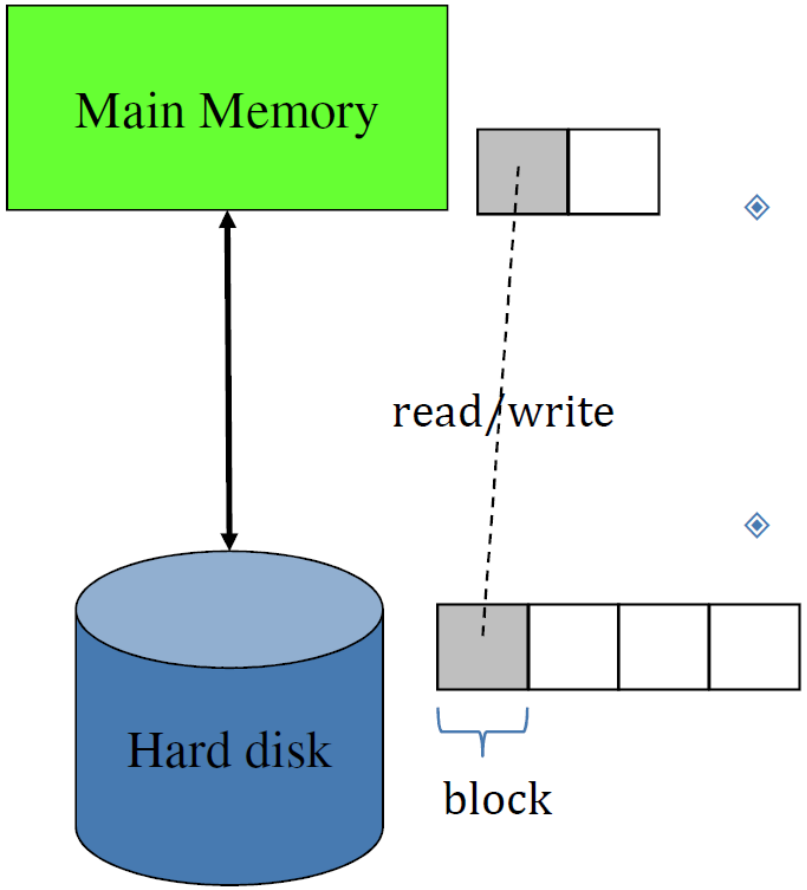
How to process a SQL query



Issues to explore

1. How to store data in the hard disk?
2. Evaluation engine: How to execute a plan?
3. Optimizer: How to find the fastest plan?

Storage



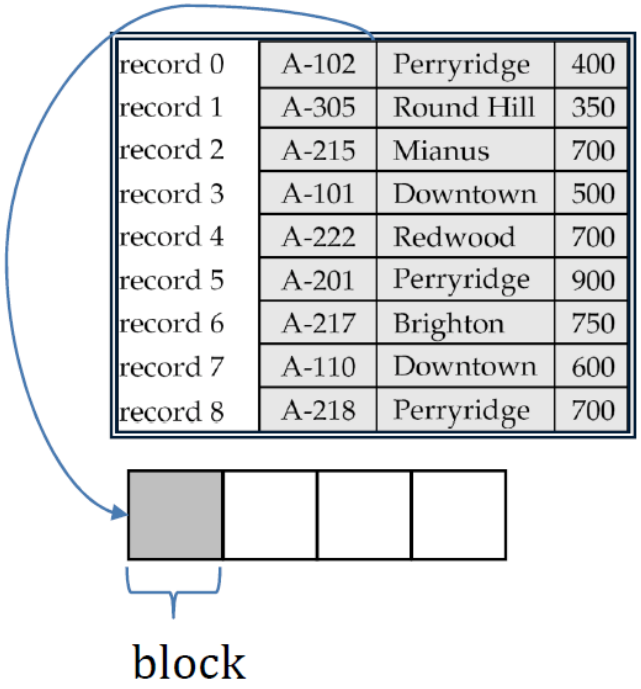
- RDBMS stores data in hard disk because
  - Large capacity (e.g., 10 TB)
  - Non-volatile (Keep data without power)
- **Disk block** – unit of data transfer between disk and main memory
  - Typical block size: 4 kB
    - Much larger than an attribute's size
- Access time = $b * t_T + s * t_S$

  b – the number of disk block transfers
  s – the number of seeks
  t_T – time to transfer one disk block
  t_S – time to seek one disk block
  - Typical values: seek time = 4-10 ms, data transfer rate = 25-100 MB/s

    → much slower than CPU time!

File Organization

- How to store a relation (as a file) in the hard disk?
  - File = a sequence of *records*
  - Record = a sequence of fields (attribute values)
  - Assume fixed-length record



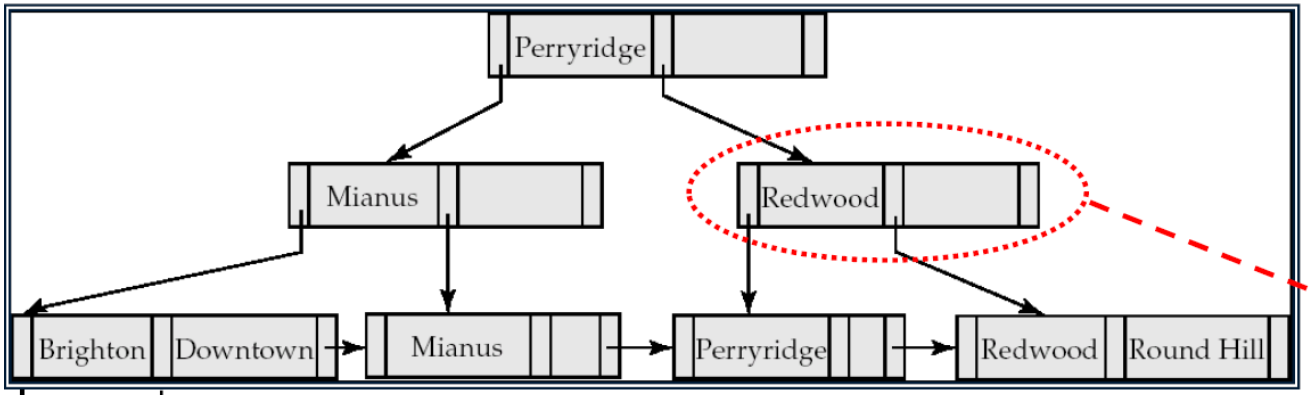
- Notations for a relation r
  - n_r: # of records in r
  - f_r: # of records per disk block
  - b_r: # of blocks

$$b_r = n_r / f_r$$

- How to search records efficiently?

B+-tree Structure

- Except the root node, each node must be at least half full
- Non-leaf nodes vs. leaf nodes
  - Sibling pointer in leaf node is used to process range queries



B+-tree for *account* file

A-212	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
⋮		
account file		

B+-tree node

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n

- Each node occupies a disk **block**
- Search-keys in a node are ordered :

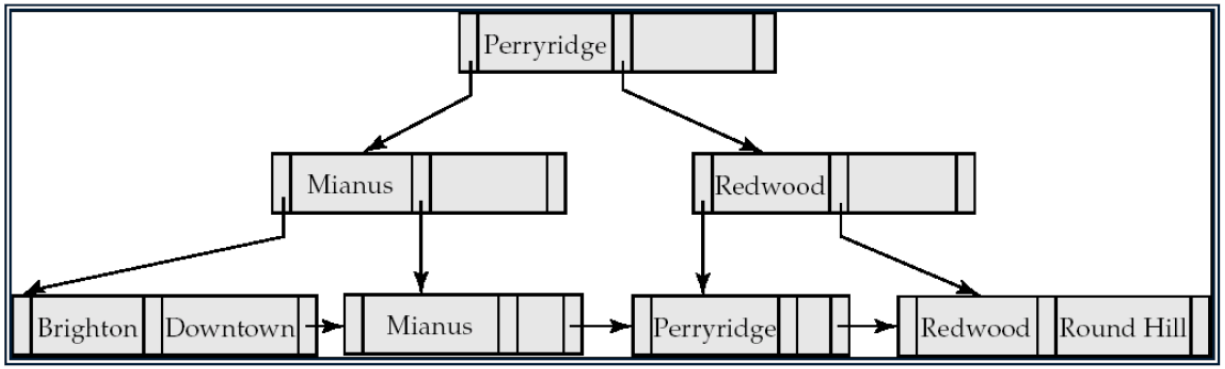
  $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
- Each pointer is a disk block address

Search on B$^+$-tree

- How to find records with a search-key value of k?
  1. Start with the *root* node
  2. Recursively follow a pointer so that it is smallest search-key > k
  3. When at leaf node, find the pointer to the desired record (if exists)

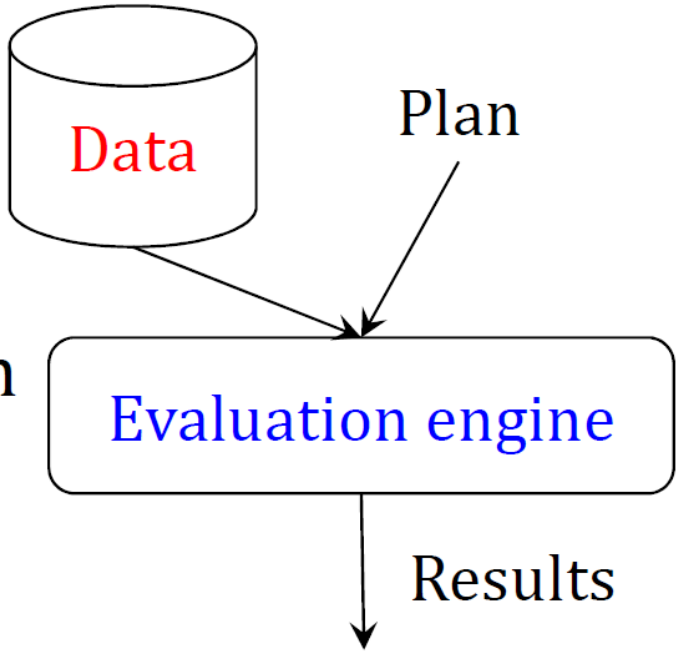
Observations

- Lookup cost = tree height = $\lceil \log_{\lceil f/2 \rceil}(K) \rceil$
  - K: # of search-key values in the file
  - f: maximum # of pointers in a node
- Example: given K = 1 million,
  block size = 4096 bytes,
  key size = 40 bytes
  pointer size = 4 bytes
  - How to calculate f? f = 94
  - Lookup cost
    $= \lceil log_{47}(1{,}000{,}000) \rceil$ = 4 blocks



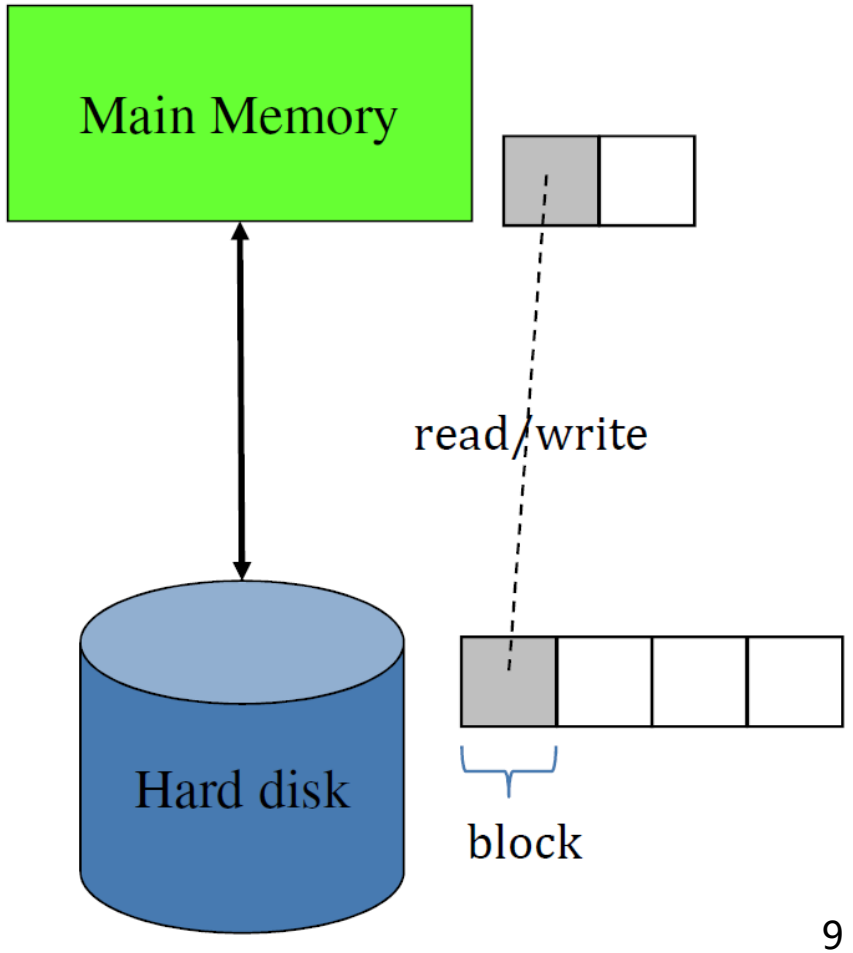
Evaluation Engine

- How to measure the cost of a plan?
- Some methods for executing a selection operator
- Some methods for executing a join operator



Execution Cost

- Cost = the number of disk block transfers
- Assumptions in RDBMS
  - Ignore CPU costs
  - Ignore the cost of writing the final output to disk
- Extra assumption in this course
  - Ignore the disk seek time, because the number of disk block transfers >> the number of disk seeks



Selection Operation

- Example: $\sigma_{\text{balance}<2500}(account)$
- Several different algorithms to implement selections
  - Usually choose the cheapest available one

Algorithm / physical operator	**Cost (# disk blocks)**
Linear search	b_r
Primary index, equality on key	$h_r + 1$
Primary index, equality on non-key	$h_r + b_{results}$
Secondary index, equality	$h_r + n_{results}$

b_r: size of r in blocks

h_r: height of B$^+$-tree on r

$n_{results}$: # of results

$b_{results}$: result size in blocks

For simplicity, we measure the cost as the number of disk block transfers.

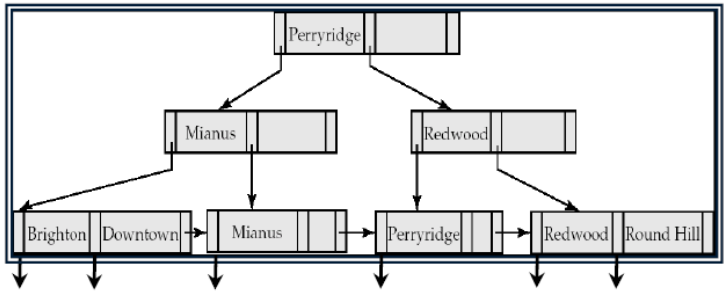
Selection

- *Linear search*:

  Scan each file block and test all tuples
  - Applicable to any type of condition
  - Cost = b_r blocks (number of blocks occupied by relation r)

- *Primary index on candidate key, equality*:

  Retrieve a single tuple that satisfies the equality condition
  - Cost = $h_r + 1$ blocks
    - B$^+$-tree height $h_r = \lceil \log_{\lceil f/2 \rceil} n_r \rceil$



- *Primary index on nonkey, equality:*

  Retrieve multiple (consecutive) tuples that satisfies ……
  - Cost = $h_r + b_{results}$ blocks
    - Let $b_{results}$ = number of blocks containing matching tuples

Join Operation

- Example: $customer \bowtie depositor$
- Several different algorithms to implement joins
  - Usually choose the cheapest available one

Algorithm / physical operator	Cost (# disk blocks)
Nested-loop join	$n_r * b_s + b_r$ [worst case]
Block nested-loop join	$\lceil b_r / (M-2) \rceil * b_s + b_r$
Indexed nested-loop join	$n_r * (h_s + 1) + b_r$
Merge-join	$b_r + b_s$ $+ b_r (2\lceil \log_{M-1}(b_r/M) \rceil + 1)$ $+ b_s (2\lceil \log_{M-1}(b_s/M) \rceil + 1)$
Hash-join	$3(b_r + b_s) + 4\, n_h$ if no recursive partitioning required

For simplicity, we measure the cost as the number of disk block transfers.

Nested-Loop Join

Outer relation

Inner relation

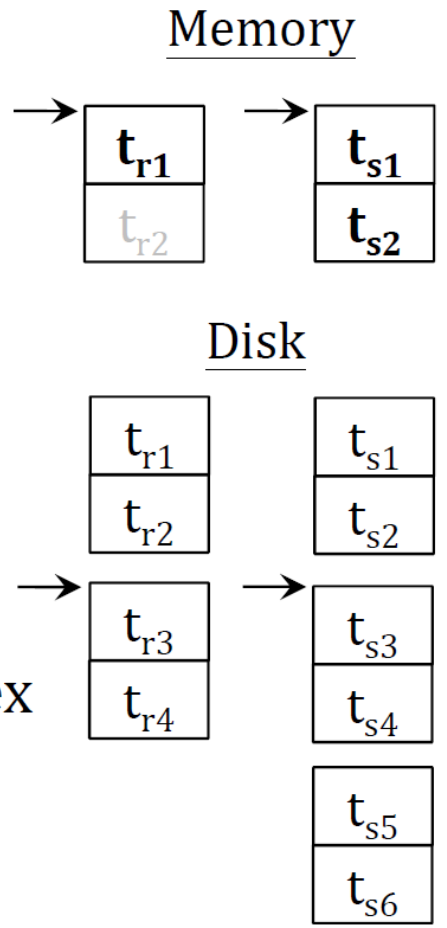
for each tuple t_r **in** r **do**

  for each tuple t_s **in** s **do**

    if pair (t_r, t_s) satisfies the join condition θ

    then add $t_r \bullet t_s$ to the result

$r \bowtie_\theta s$



- Applicable to any join condition, requires no index
- But expensive!
- Improvement: *Index Nested-Loop Join*
  - Use index on inner relation if available

Nested-Loop Join

- Cost of *nested loop join*: $n_r * b_s + b_r$ blocks
  - Assume the worst case: only one memory buffer block for each relation
- Cost of *indexed nested loop join*: $n_r * (h_s + 1) + b_r$ blocks
- **Exercise**: *customer* ⋈ *depositor*
  - Number of tuples: $n_{customer}$=10000 $n_{depositor}$=5000
  - Number of blocks: $b_{customer}$=400 $b_{depositor}$=100
  - Suppose that *customer* has a primary B⁺-tree index on the join attribute *customer-name,* which contains 20 entries per index node.
  - Cost of *nested loop join*

    If outer relation = *depositor* : _____ * _____ + _____ = _____ blocks

    If outer relation = *customer* : _____ * _____ + _____ = _____ blocks
  - Cost of *indexed nested loop join*

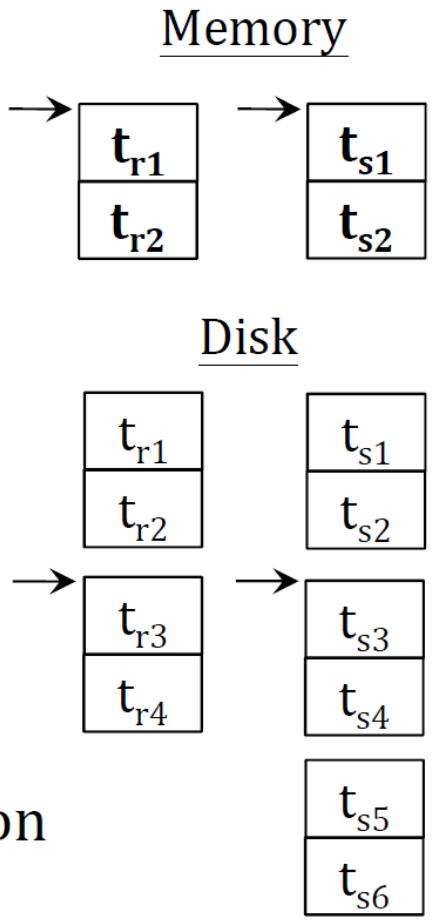
    E.g., the tree height (for *customer*) is 4, so the cost is:

    5000*(4+1)+100 = 25,100 blocks

$$\lceil \log_{\lceil f/2 \rceil}(n_{customer}) \rceil = \lceil \log_{\lceil 20/2 \rceil}(10000) \rceil = 4$$

Block Nested-Loop Join

- Variant of nested-loop join
- Pair every block of the inner relation with every block of the outer relation



for each block B_r **of** r **do**
  for each block B_s **of** s **do**
    for each tuple t_r **in** B_r **do**
      for each tuple t_s **in** B_s **do**
        if pair (t_r, t_s) satisfies the join condition
        then add $t_r \bullet t_s$ to the result

Block Nested-Loop Join

- Worst case cost: $b_r * b_s + b_r$ blocks
  - Assume the worst case: only one memory buffer block for each relation

- If the memory has M blocks,

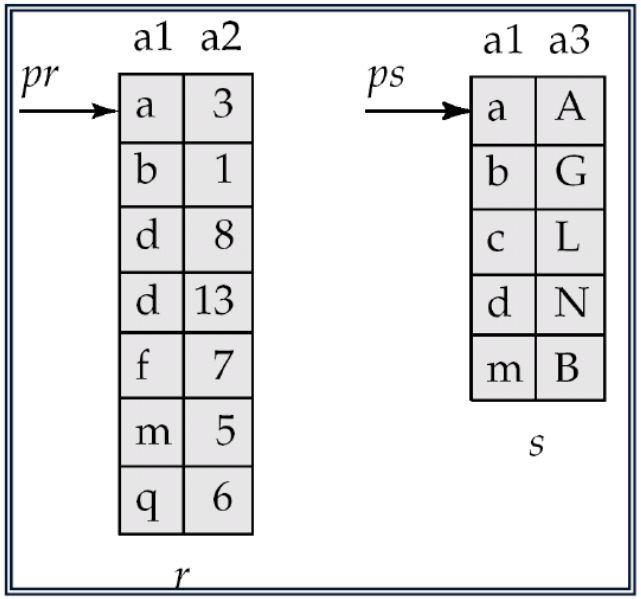
$$\text{Cost} = \lceil b_r / (M-2) \rceil * b_s + b_r \text{ blocks}$$

  - Use the memory blocks as follows:
    - $M - 2$ memory blocks to buffer the outer relation
    - 1 block to buffer the inner relation
    - 1 block to buffer the output

Merge-Join

1. Sort both relations on their join attribute *(to discuss soon)*
2. Merge the sorted relations to join them
   a. This step is similar to the merge stage of the sort-merge algorithm.
   b. The difference is to handle duplicate values in join attribute
      — every pair with same value on join attribute must be matched

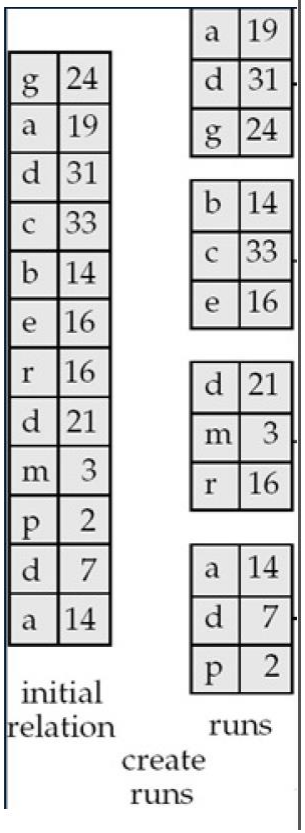
- Applicable to equi-joins and natural joins
- Each block is only read once
  - assuming all tuples for any given value of the join attributes fit in memory
- Cost of merge join =
  $b_r + b_s$ blocks
  + the sorting cost (if relations are unsorted)



External Sort Merge-Join

- Use it when the relation is larger than the main memory, i.e., $b_r > M$
- **External sort-merge** algorithm
  - 1. Create sorted runs
    - Read consecutive M blocks into memory, sort it, then write to a run
  - 2. Merging until only 1 run remains
- Cost: $b_r \ (2\lceil \log_{M-1}(b_r/M) \rceil + 1)$
  - Number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$
  - Block transfers for initial run and in each pass is $2b_r$

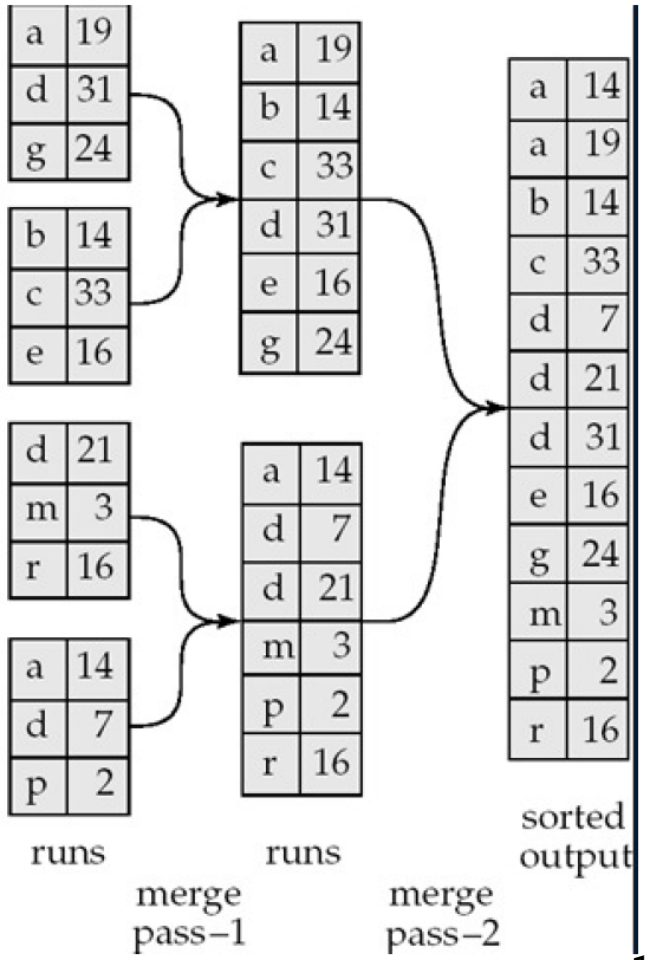
Example: M = 3 memory blocks



External Sort Merge-Join

- How to merge "sorted runs"?
- 2. Merging (for every M–1 runs)
  - Use M–1 memory blocks as input buffers, and 1 memory block as output buffer
  - Move the smallest tuple from its input buffer to the output buffer
  - An input block empty → fill it with the next disk block from its input run
  - An output block full → flush the block to its output run
- Repeat until only 1 run remains

Example: M = 3 memory blocks



Hash Join Algorithm

* Applicable to equi-joins and natural joins

1. Partition the relation *s* using a **hash function** *h* (into n_h partitions)
   * note that n_h must be smaller than M
2. Partition *r* similarly
3. For each partition value *i*

(a) Create another hash function *h'*

(b) Load s_i into memory.

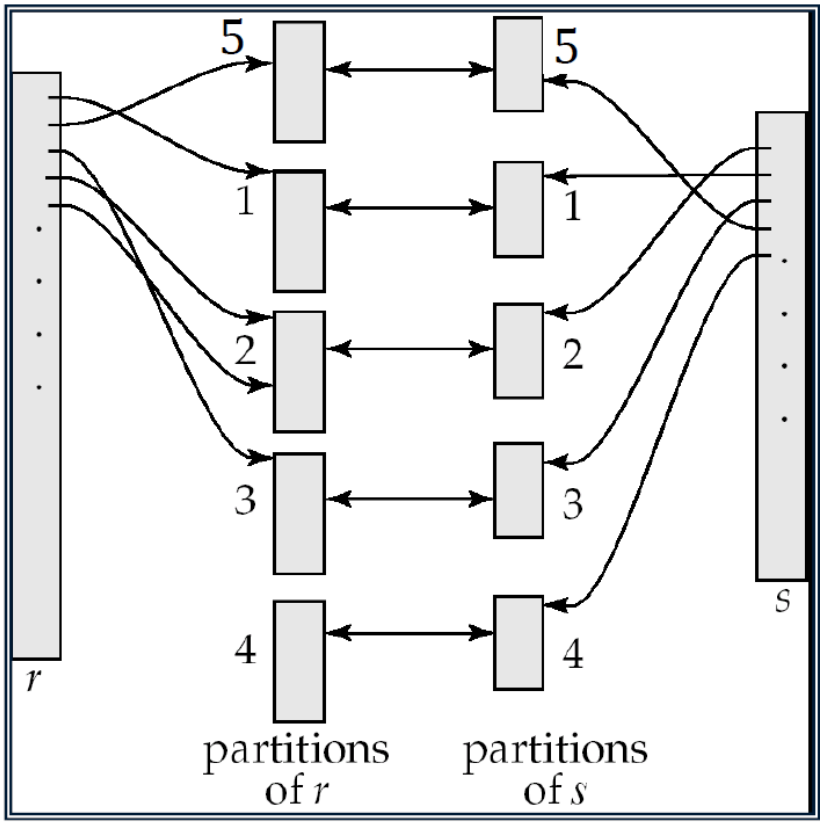
Build an in-memory hash index on it by the join attribute (by using *h'*).

(c) Read the tuples in r_i.

For each tuple t_r, find each matching tuple t_s in s_i from in-memory hash index.

Output the result.

Example: n_h = 5 partitions



r : probe input **s : build input**

Hash Join Algorithm

- Suppose that number of partitions n_h is at most number of memory blocks M
- Cost of hash join: $3(b_r + b_s) + 4\,n_h$
  - Partitioning phase: $2(b_r + b_s)$
    - read and write each relation once
  - Build and probe phase: $b_r + b_s$
  - Partially filled blocks: $2\,(\,2\,n_h\,)$

small, can be ignored

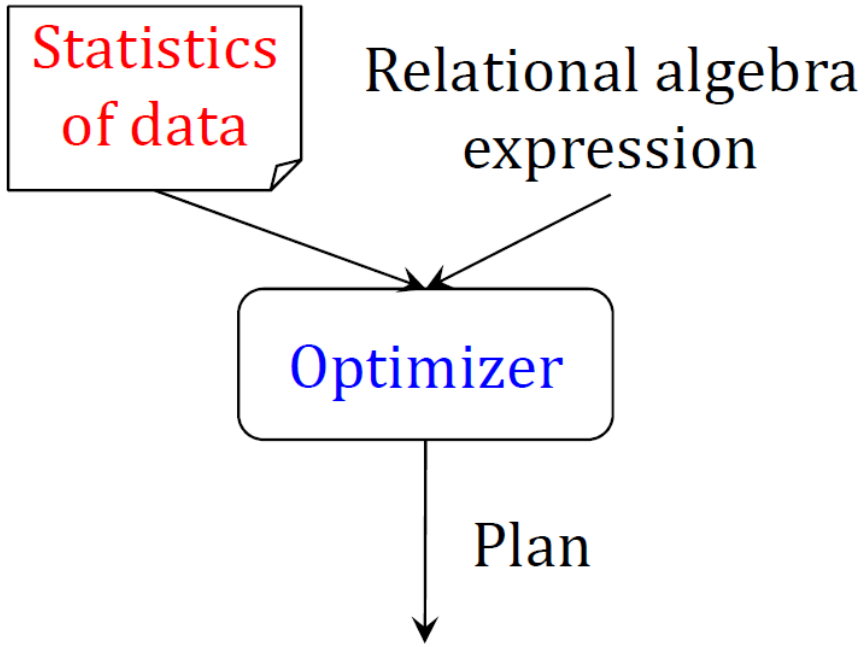
- **Recursive partitioning** is required if $n_h > M$
  - Rarely required: e.g., recursive partitioning not needed for relations of 1GB with memory size of 2MB and block size of 4KB

Cost of Hash Join

$$customer \bowtie depositor$$

- Given that
  - memory size is 20 blocks
  - $b_{depositor}$= 100 and $b_{customer}$ = 400.
- Use the smaller relation (*depositor*) as build input
- How large should a partition be?
  - To make each partition of *depositor* fit in memory (20 blocks), we can partition it into 100/20 = 5 partitions
  - Since 5 < 20, this partitioning can be done in **one pass**
  - Similarly, we partition *customer* into 5 partitions
- Therefore total cost:
  - 3(100 + 400) = 1500 blocks

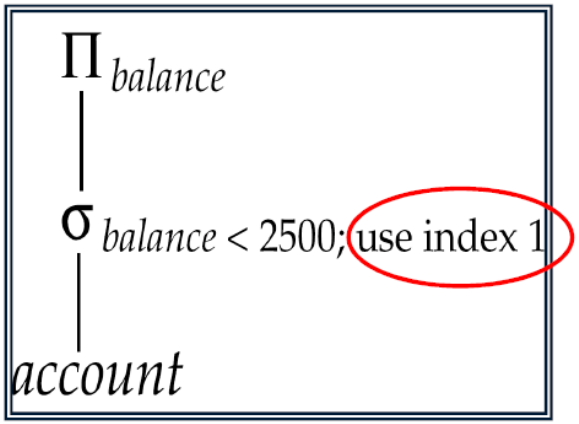
How to find the fastest plan



- Why optimizer?
- Techniques
  - Equivalence rules
  - Join ordering by dynamic programming
  - Cost and size estimation

Query Optimizer

- A query (in relational algebra) has equivalent expressions
  - $\sigma_{balance<2500}(\Pi_{balance}(account))$
  - $\Pi_{balance}(\sigma_{balance<2500}(account))$
- An operation (e.g., selection $\sigma_{balance<2500}$) can be evaluated by different algorithms
  - Scan the whole *balance* relation
  - Search an index on *balance*
- Different plans can have very different costs (e.g. seconds vs. days)



Why Query Optimizer?

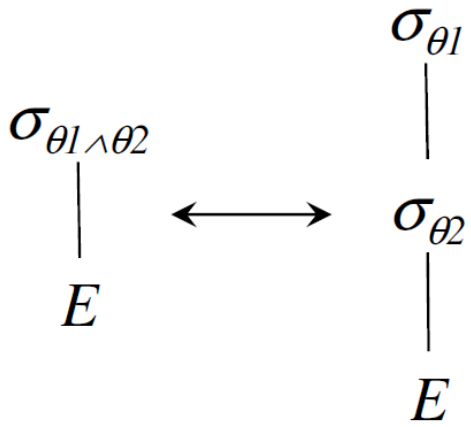
- A query optimizer executes two steps
- 1. Query rewriting
  - Relational algebra expression → logical plan
  - Rewrite a query by using materialized views
  - Obtain a cheaper plan by using **equivalence rules**
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
- 2. Cost-based enumeration
  - Logical plan → Physical plan
  - Enumerate join ordering by using **dynamic programming**
    - Estimate the cost of each candidate plan, then pick the cheapest plan
    - Need to **estimate the cost and output size** of operators

Equivalence Rules

- **Equivalence rule**: two relational algebra expressions output the same set of tuples on every database instance

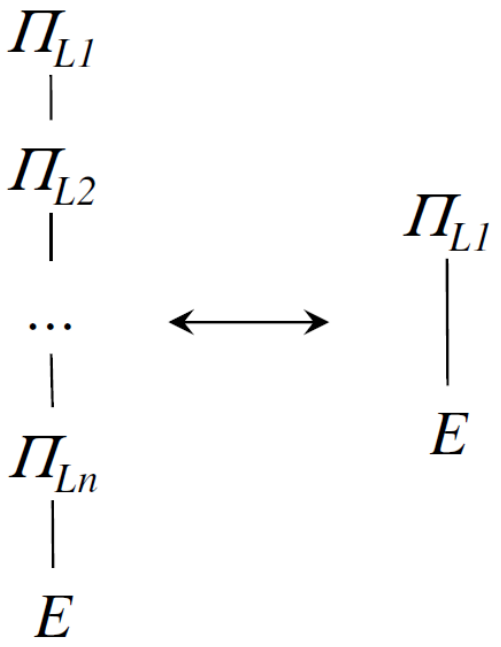
Conjunctive selection operations can be decomposed

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$



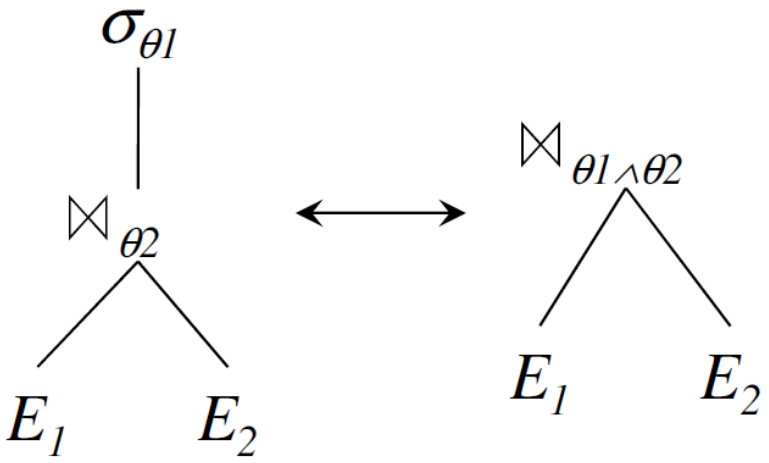
Only need the last project operation

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

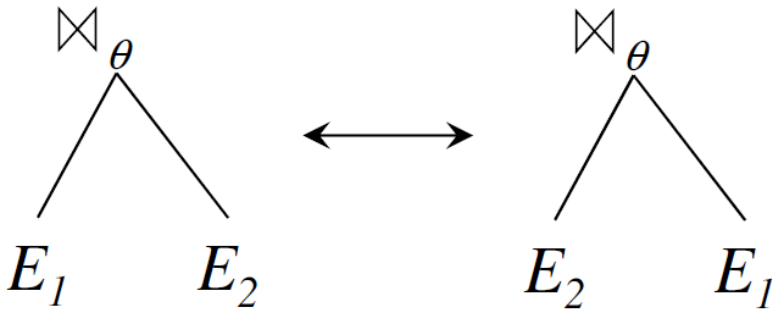


Joins are commutative

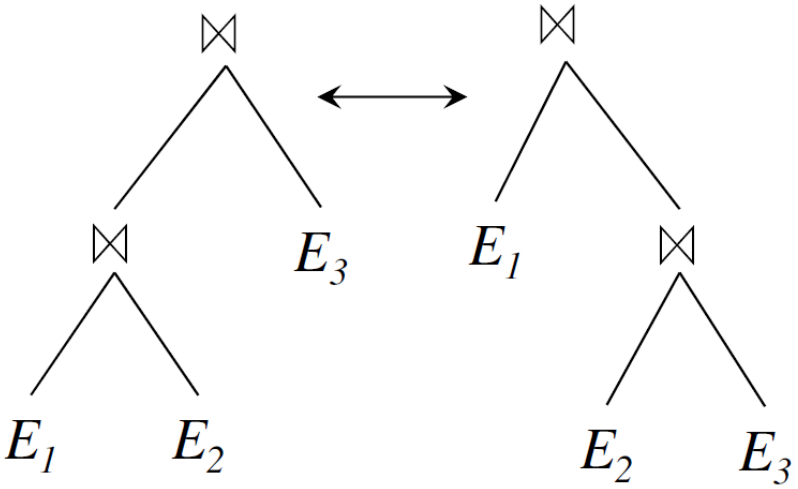
Selections can be combined with joins



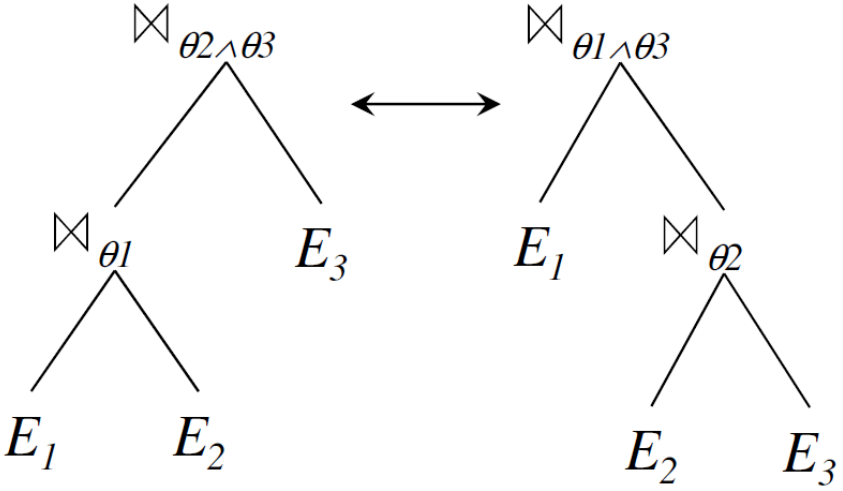
Joins are commutative



Joins are associative

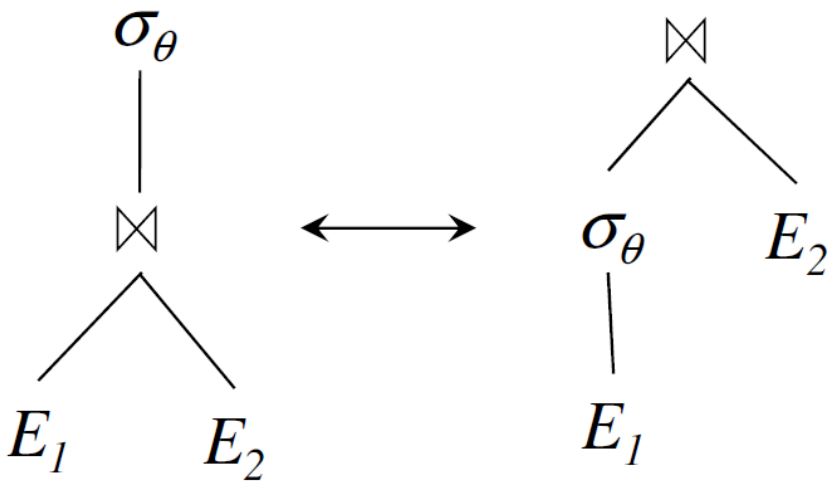


when θ_2 only has attributes of E_2 and E_3

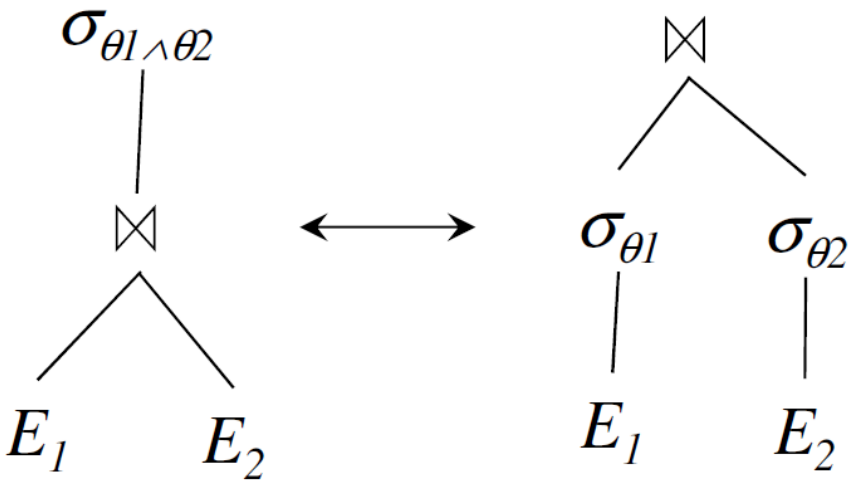


Selection Distributes over Joins

when θ only has attributes of E_1

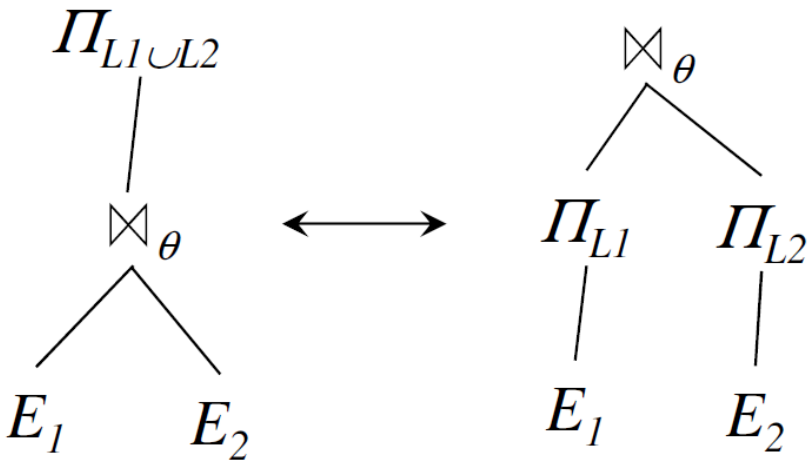


when θ_1 only has attributes of E_1 and θ_2 only has attributes of E_2

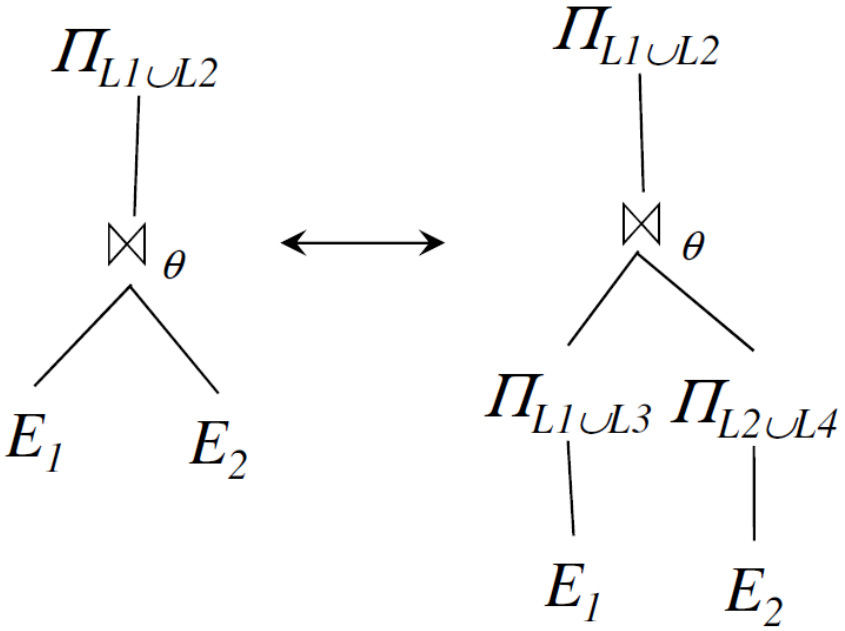


Projection Distributes over Join

if θ only has attributes of $L_1 \cup L_2$



Let L_3 be attributes of E_1 that are in θ, but not in $L_1 \cup L_2$, and
Let L_4 be attributes of E_2 that are in θ, but not in $L_1 \cup L_2$



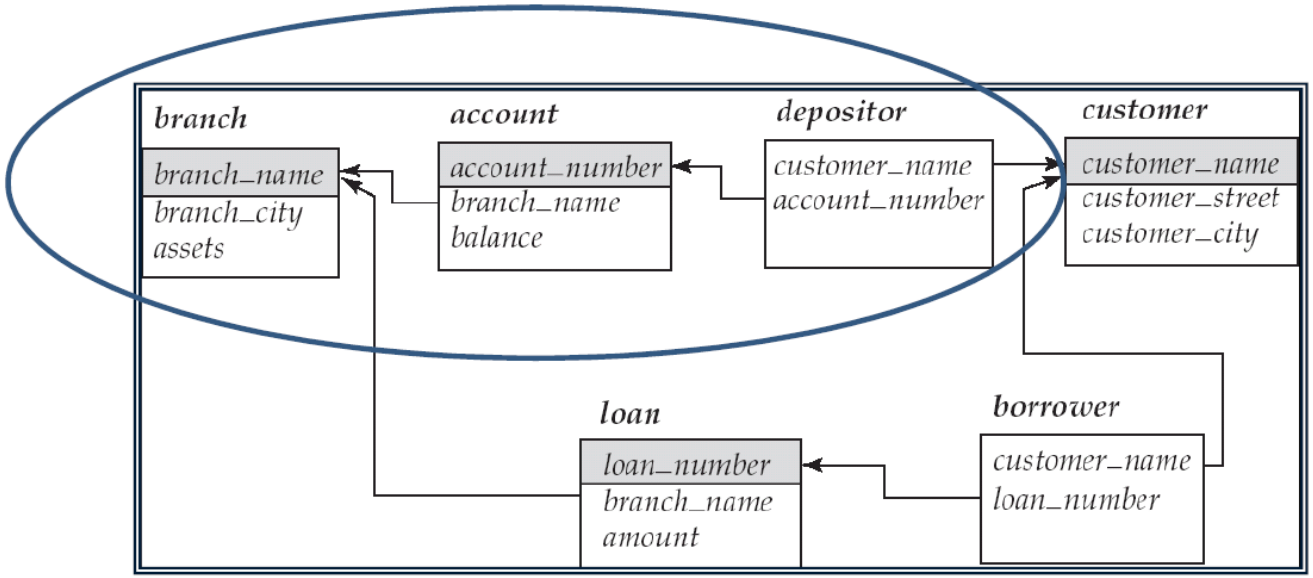
Example

branch (branch_name, branch_city, assets)

account (account_number, branch_name, balance)

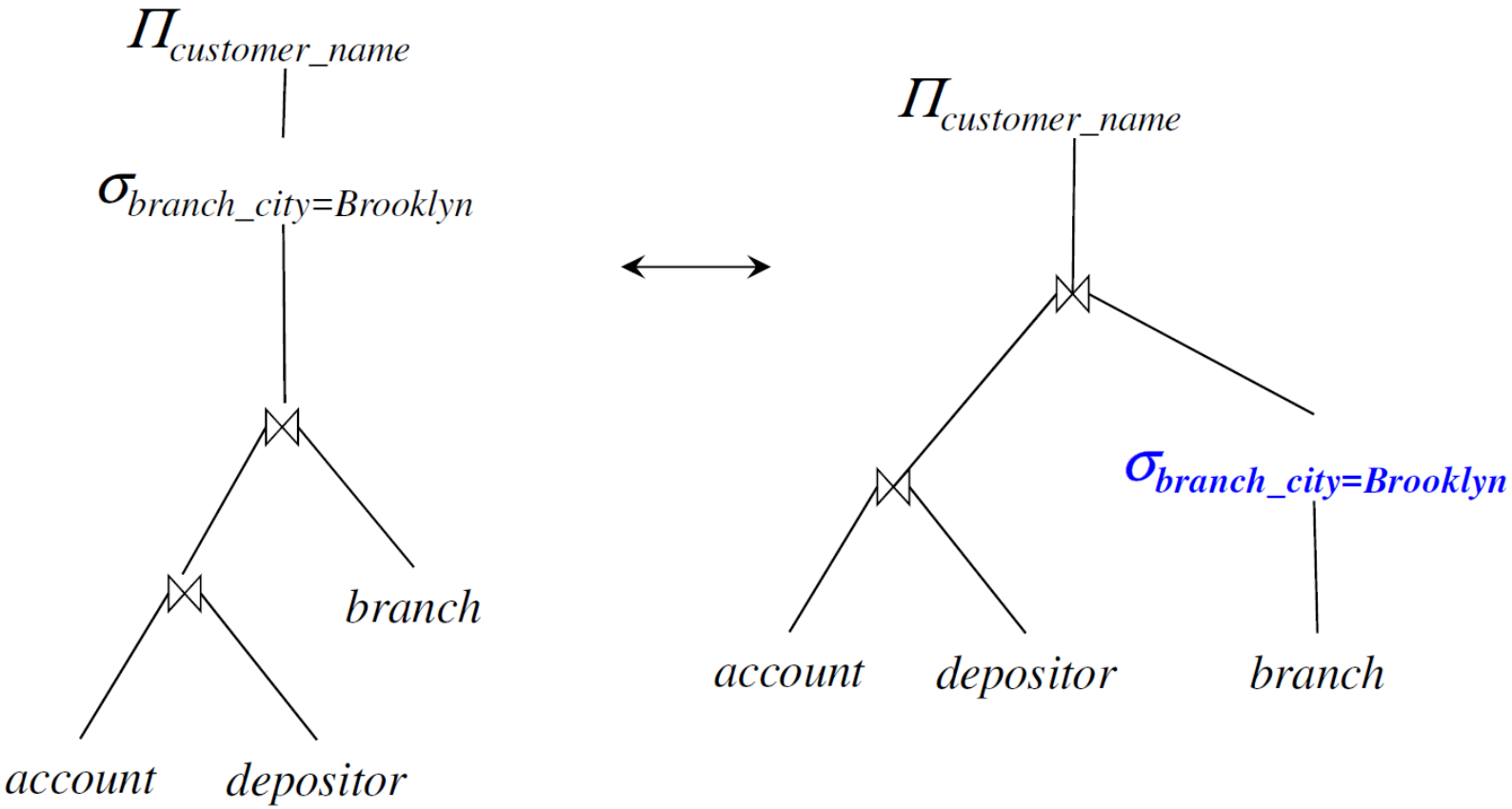
depositor (customer_name, account_number)

.....



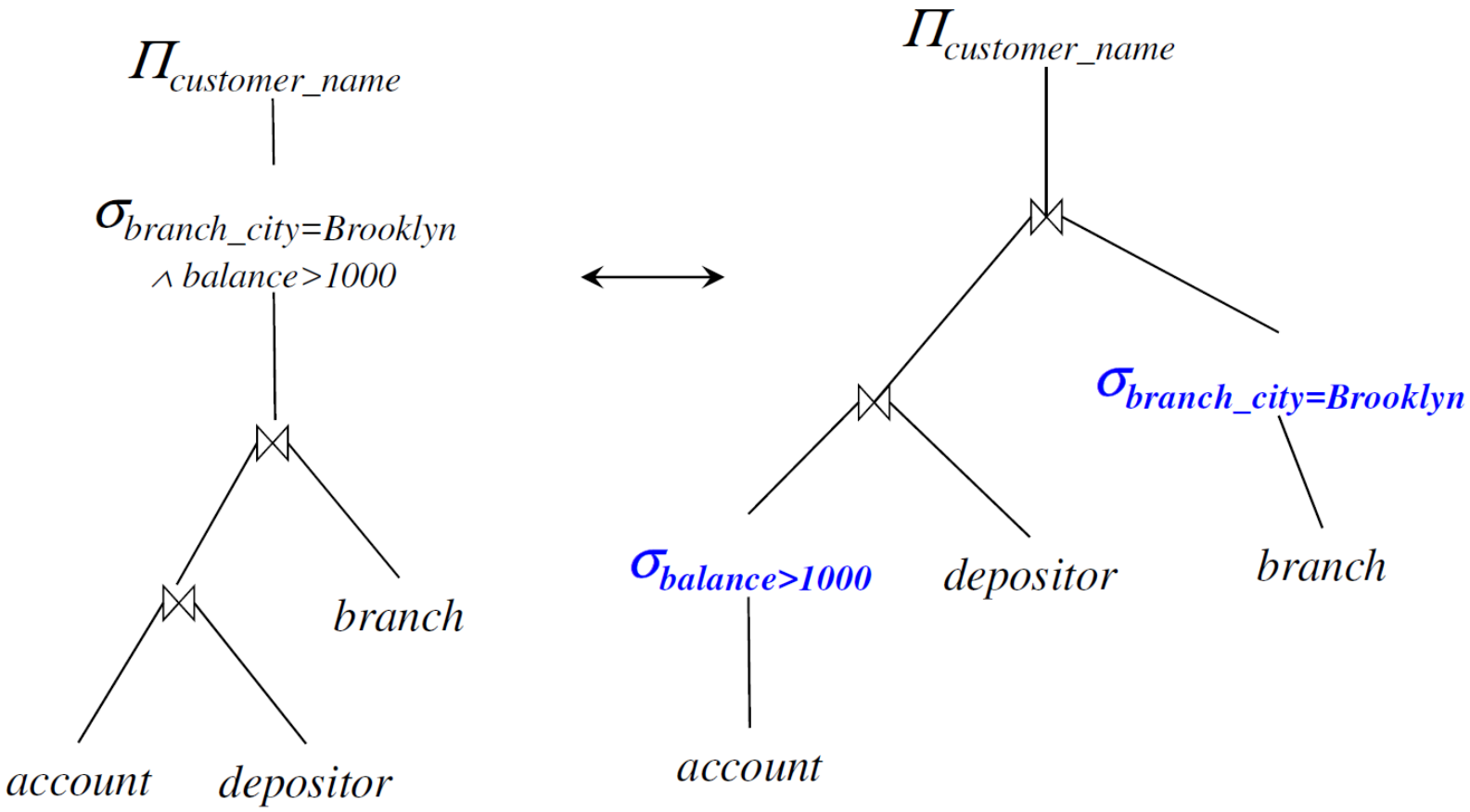
Reducing # of tuples early

- Query: Find the names of all customers who have an account at some branch located in Brooklyn



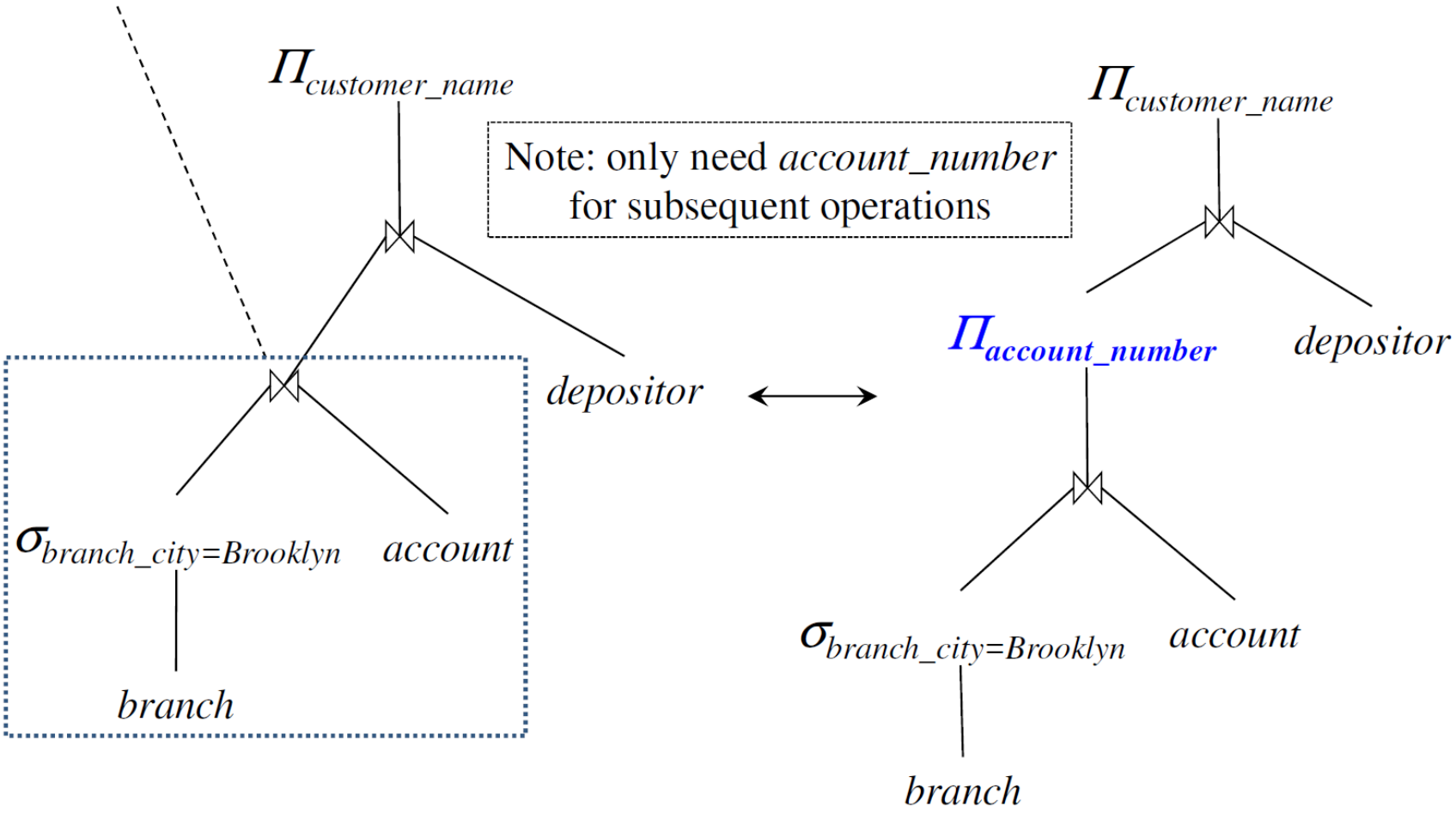
Reducing # of tuples early

- Query: Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000

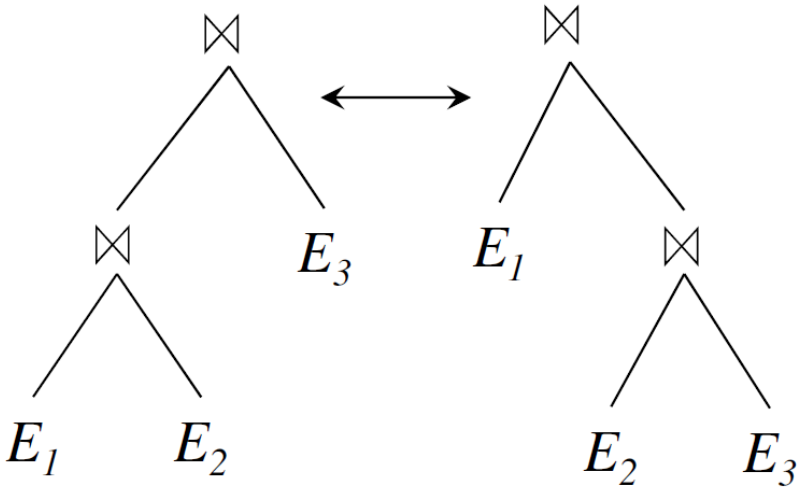


Reducing # of tuples early

the output relation schema here is:
(*branch_name, branch_city, assets, account_number, balance*)



Join Ordering Example

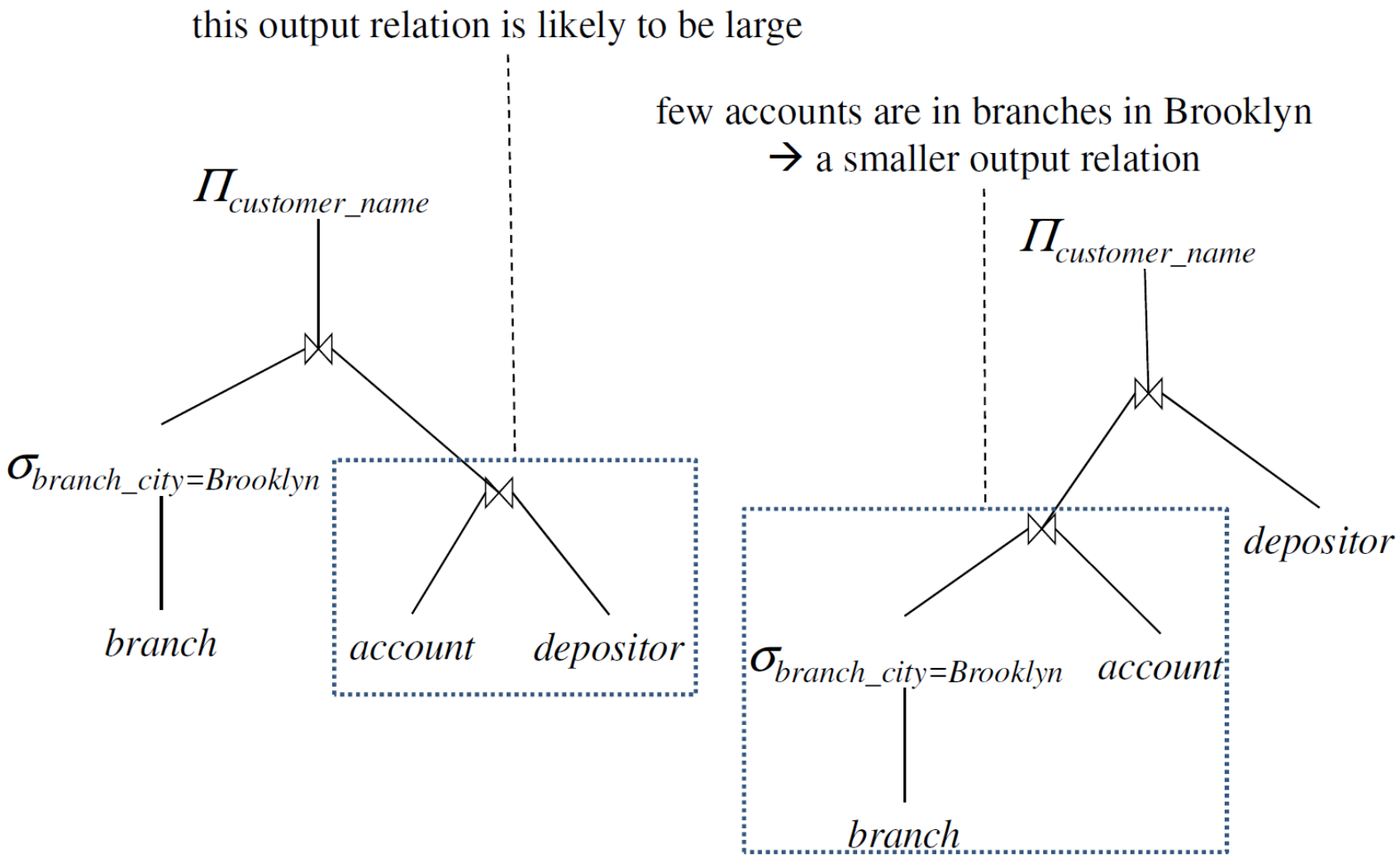


- If $E_2 \bowtie E_3$ is large and $E_1 \bowtie E_2$ is small, we choose

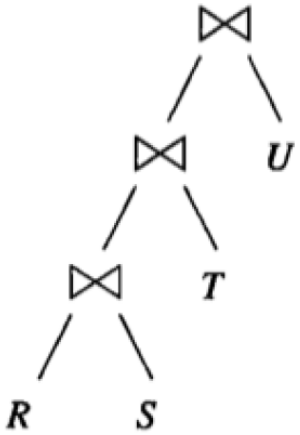
$$(E_1 \bowtie E_2) \bowtie E_3$$

so that we compute a smaller temporary relation

Join Ordering Example

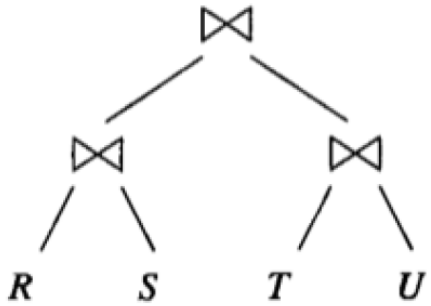


Join Ordering for multiple relations



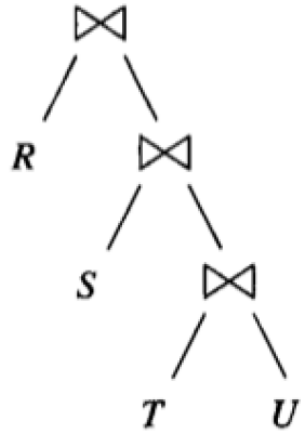
(a)

Left-deep join



(b)

Bushy



(c)

Right-deep

Dynamic Programming

- As a heuristic, optimizers enumerate left-deep plans only
  - Still *n!* possible left-deep plans for *n* relations
- Dynamic programming approach
  - Find the best plans for 2 relations
  - Use them to find the best plans for 3 relations
  - ……
  - Use them to find the best plans for *n* relations

Dynamic Programming Example

- Example from Chapter 16.6.4 of "Database Systems: The Complete Book", 2nd Edition, Prentice Hall, 2009
  - R(a,b) ⋈ S(b,c) ⋈ T(c,d) ⋈ U(d,a)
  - Each relation has 1000 tuples

- Cost estimation
  - For simplicity, assume the cost of a join operator
    = the # of tuples of input *intermediate relation(s)*
    - For accurate estimation, use the cost equations on pages 10, 12
  - Estimate the output result size, between two relations r and s on join attribute x, by:
    $$\frac{n_r * n_s}{\max\{V(r,x), V(s,x)\}}$$
    - For accurate estimation, use the equations on page 45

Example

- 6 possible combinations: {R,S}, {R,T}, {R,U}, {S,T}, {S,U}, {T,U}
- Consider the combination {R,S}
  - Estimate its output size R ⋈ S

    = 1000*1000/max(200,100)

    = 5000
  - Cost = 0 as input relations are not intermediate

$R(a,b)$	$S(b,c)$	$T(c,d)$	$U(d,a)$
$V(R,a) = 100$			$V(U,a) = 50$
$V(R,b) = 200$	$V(S,b) = 100$		
	$V(S,c) = 500$	$V(T,c) = 20$	
		$V(T,d) = 50$	$V(U,d) = 1000$

	$\{R,S\}$	$\{R,T\}$	$\{R,U\}$	$\{S,T\}$	$\{S,U\}$	$\{T,U\}$
Size	5000	1,000,000	10,000	2000	1,000,000	1000
Cost	0	0	0	0	0	0
Best plan	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

Example

$R(a,b)$	$S(b,c)$	$T(c,d)$	$U(d,a)$
$V(R,a)=100$			$V(U,a)=50$
$V(R,b)=200$	$V(S,b)=100$		
	$V(S,c)=500$	$V(T,c)=20$	
		$V(T,d)=50$	$V(U,d)=1000$

- 4 possible combinations: {R,S,T}, {R,S,U}, {R,T,U}, {S,T,U}
- Consider the combination {R,S,T}
  - Estimate its output size: 5000*1000/max(500,20)=10000
  - Cost of (R⋈S)⋈T = 5000
  - Cost of (S⋈T)⋈R = 2000
  - Best plan: (S⋈T)⋈R

	$\{R,S\}$	$\{R,T\}$	$\{R,U\}$	$\{S,T\}$	$\{S,U\}$	$\{T,U\}$
Size	5000	1,000,000	10,000	2000	1,000,000	1000
Cost	0	0	0	0	0	0
Best plan	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

	$\{R,S,T\}$	$\{R,S,U\}$	$\{R,T,U\}$	$\{S,T,U\}$
Size	10,000	50,000	10,000	2,000
Cost	2,000	5,000	1,000	1,000
Best plan	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

Example

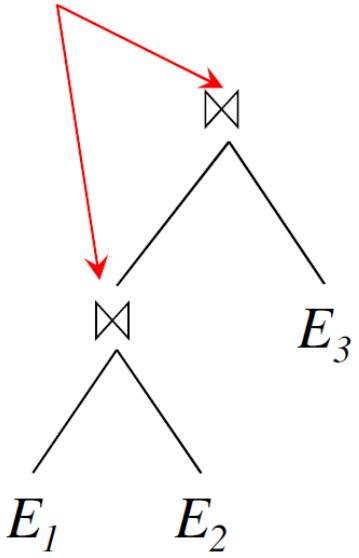
- Only 1 combination: {R,S,T,U}
  - The output size is the same for all cases below:
  - Cost of ((S⋈ T)⋈ R)⋈ U = 2000 + 10000 = 12000
  - Cost of ((R⋈ S)⋈ U)⋈ T = 5000 + 50000 = 55000
  - Cost of ((T⋈ U)⋈ R)⋈ S = 1000 + 10000 = 11000
  - Cost of ((T⋈ U)⋈ S)⋈ R = 1000 + 2000 = 3000
  - Best plan: ((T⋈ U)⋈ S)⋈ R

	$\{R,S,T\}$	$\{R,S,U\}$	$\{R,T,U\}$	$\{S,T,U\}$
Size	40,000	50,000	10,000	2,000
Cost	2,000	5,000	1,000	1,000
Best plan	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

Grouping	Cost
$((S \bowtie T) \bowtie R) \bowtie U$	12,000
$((R \bowtie S) \bowtie U) \bowtie T$	55,000
$((T \bowtie U) \bowtie R) \bowtie S$	11,000
$((T \bowtie U) \bowtie S) \bowtie R$	3,000

Example

- Cost of a plan = the sum of the cost of each operator
- Cost of each operator (see earlier slides)
  - Need statistics of input relations
    - E.g. number of tuples, sizes of tuples
  - Some input can be another operator's output
    - Need to estimate the size of such input
    - To do so, we need additional statistics, e.g., number of distinct values for an attribute



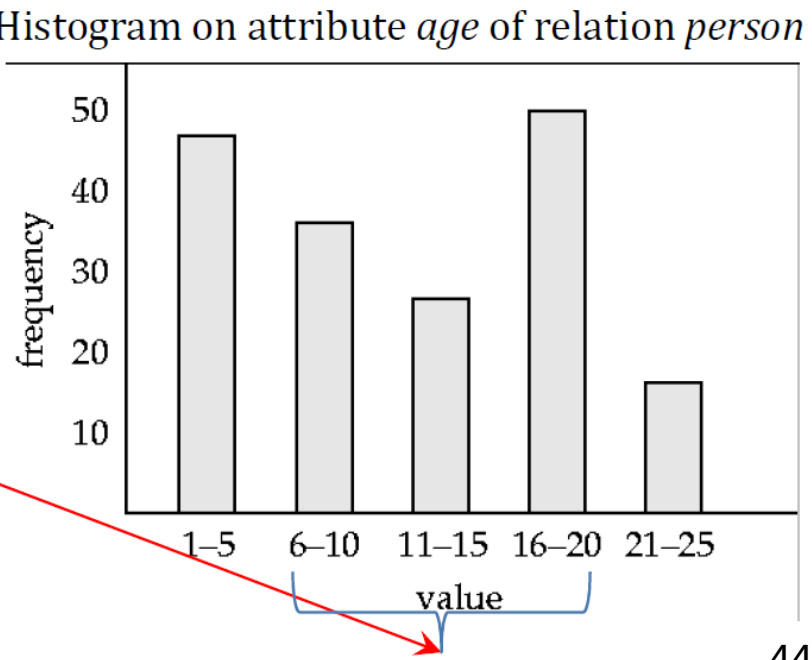
- Notations
  - n_r: number of tuples in a relation r
  - f_r: blocking factor, i.e., number of tuples of r per block
  - b_r: number of blocks in r
    - $b_r = n_r / f_r$ if tuples are stored together physically in a file
  - $V(r, A)$: number of distinct values for attribute A in r

Estimate Selection Size

- Let *size* be the estimated number of tuples satisfying the condition
- Estimate the size of $\sigma_{A=v}(r)$
  - *size* = 1 if *A* is the primary key of *r*
  - *size* = n_r / *V(r,A)* : otherwise
- Estimate the size of $\sigma_{v1 \le A \le v2}(r)$
  - If min(r,A) and max(r,A) are available in the database catalog

$$size = n_r . \frac{v_2 - v_1}{\max(r, A) - \min(r, A)}$$

- If histograms available,
  we can estimate the size more accurately
  - E.g., estimate $\sigma_{8 \le age \le 16}(person)$
    = 35*(3/5) + 25 + 50*(1/5)
    = 21 + 25 + 10
    = 56



Estimate Join Size

R: the schema of relation r

S: the schema of relation s

Cases of attributes	***Size of $r \bowtie s$***	***Remarks***
$R \cap S = \varnothing$	n_r x n_s	
$R \cap S$ is a key for r	at most n_s	a tuple of s joins with at most one tuple from r
$R \cap S$ is a foreign key in s referencing r	exactly n_s	
$R \cap S = \{A\}$ is not a key for r nor s	$\frac{n_r * n_s}{\max\{V(r,A), V(s,A)\}}$	the lower estimate, assume that every tuple t in r (*or* s) produces tuples in r $\bowtie s$

Note: symmetric cases can be obtained by swapping two relations in the above table

- n_r: number of tuples in relation r
- $V(r,A)$: number of distinct values for attribute A in relation r

Example

Given information in the database catalog:

- $n_{customer} = 10000$ $f_{customer} = 25$
  - → $b_{customer} = 10000/25 = 400$
- $n_{depositor} = 5000$ $f_{depositor} = 50$
  - → $b_{depositor} = 5000/50 = 100$

- $V(\ depositor, customer\_name\ ) = 2500$
  - → each customer has two accounts on average
- *customer_name* in *depositor* is a foreign key on *customer*
- *customer_name* in customer is a primary key
  - → $V(\ customer, customer\_name\ ) = 10000$

Example

- Using foreign keys
  - *customer_name* in *depositor* is a foreign key on *customer*
  - Estimated result size

  $= n_{depositor}$

  = 5000 tuples

- Without using information about foreign keys:
  - *V(depositor, customer_name)* = 2500, and
    V(customer, customer_name) = 10000
  - Estimated size

  = 5000 * 10000/max(2500,1000)

  = 5000 * 10000/10000

  = 5000 tuples

Other factors

- Must consider the interaction of **physical operators** in a plan
  - Choosing the cheapest *physical operator* for each operation independently may not yield the best plan
- Examples
  - *Merge-join* may be more expensive than hash-join,
    but provides a *sorted output*
    which may reduce the cost for an *outer level aggregation*
  - *Nested-loop join* may enable *pipelining* (i.e., on-the-fly processing) which avoids writing intermediate relations on the disk

Summary

- Query processing
  - Query cost
  - Operations: selection, join
- Query optimization
  - Equivalence rules
  - Join ordering by dynamic programming
  - Cost and size estimation
- Read this paper (available in Blackboard):
  - "An overview of Query Optimization in Relational Systems"

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

谢谢！

DBGroup @ SUSTech

Dr. Bo Tang (唐博)

tangb3@sustech.edu.cn