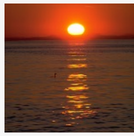
# Chapter 13: Query Processing

**Database System Concepts, 5th Ed.**

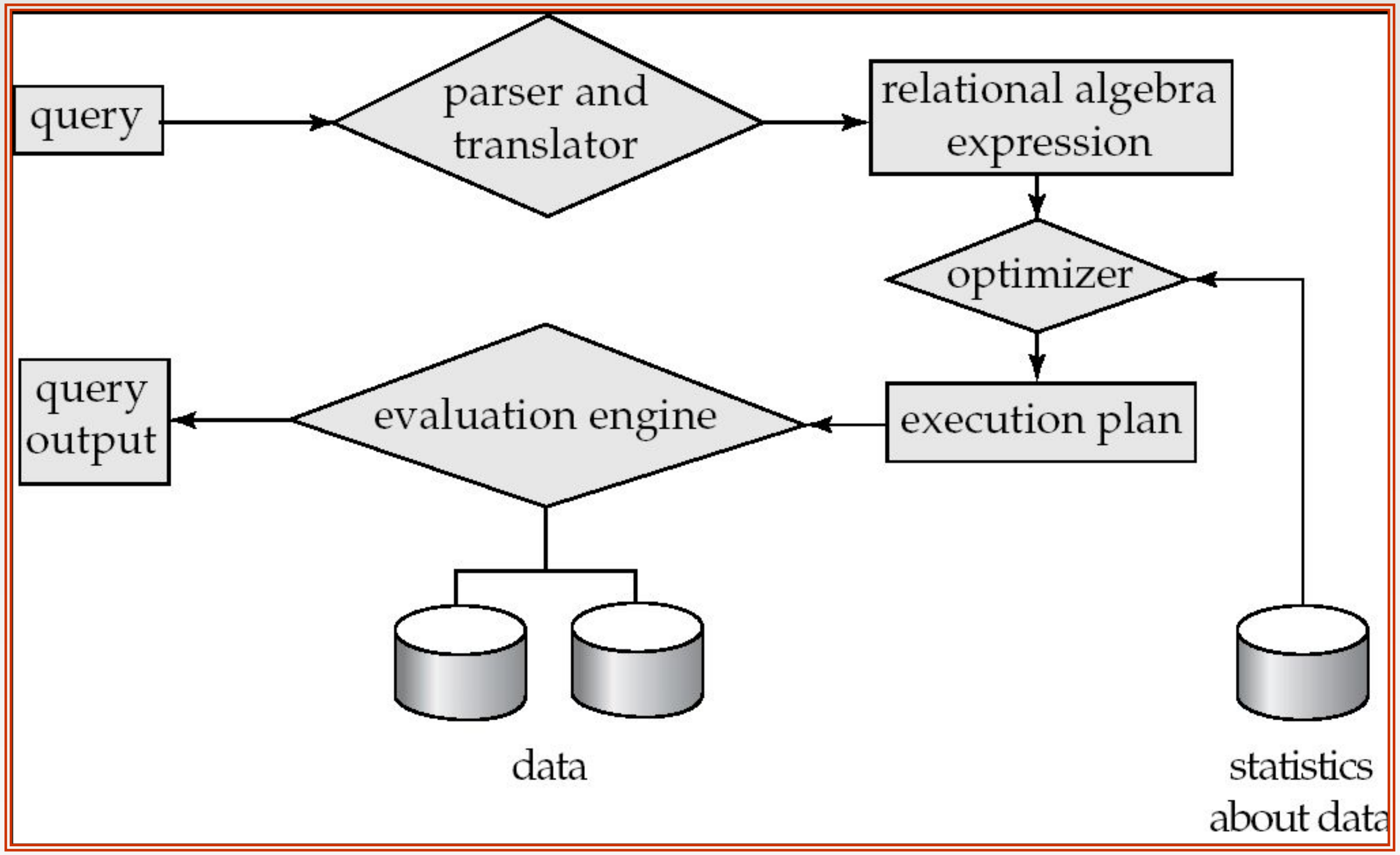


# Chapter 13: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions

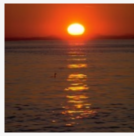
## Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



## Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

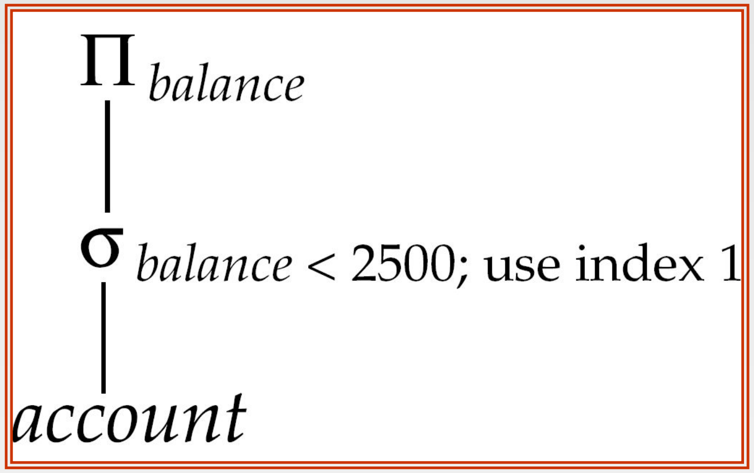


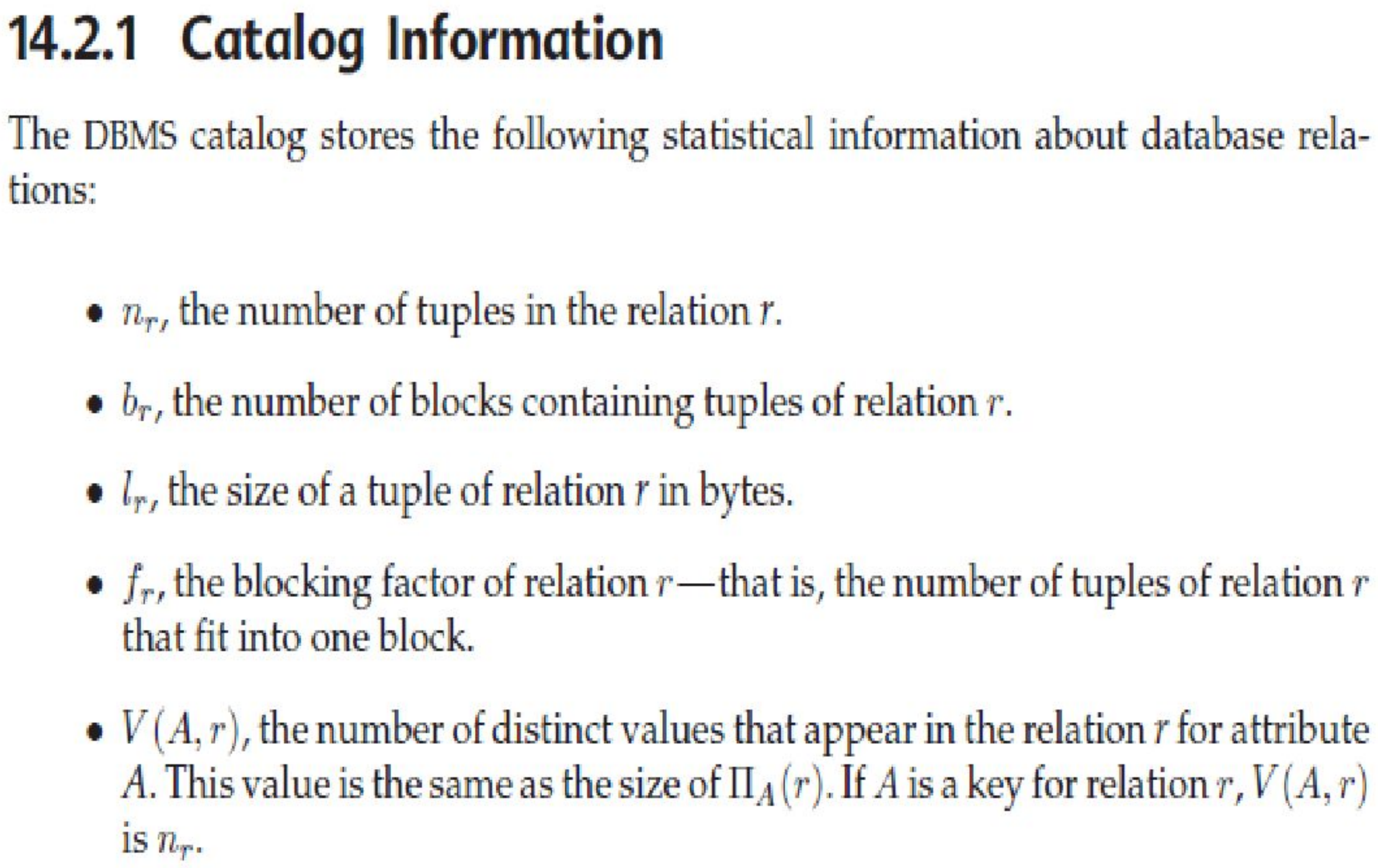
## Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$ is equivalent to
  -

    $\Pi_{\text{balance}}(\sigma_{\text{balance}<2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *balance* to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance ≥ 2500

## Figure 13.2

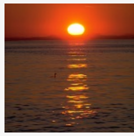




## 14.2.1 Catalog Information

The DBMS catalog stores the following statistical information about database relations:

- $n_r$, the number of tuples in the relation $r$.
- $b_r$, the number of blocks containing tuples of relation $r$.
- $l_r$, the size of a tuple of relation $r$ in bytes.
- $f_r$, the blocking factor of relation $r$—that is, the number of tuples of relation $r$ that fit into one block.
- $V(A, r)$, the number of distinct values that appear in the relation $r$ for attribute $A$. This value is the same as the size of $\Pi_A(r)$. If $A$ is a key for relation $r$, $V(A, r)$ is $n_r$.



# Metadata

- Given a relation r, DBMS likely maintains the following metadata:

1. Size (# of tuples) $\mathbf{n_r}$
2. Size (# of blocks) $\mathbf{b_r}$
3. Block size (#tuples) $\mathbf{f_r}$

   (typically $\mathbf{b_r} = \lceil \mathbf{n_r} / \mathbf{f_r} \rceil$)
4. Tuple size (in bytes) $\mathbf{s_r}$
5. Attribute Variance (for each attribute r, # of different values) **V(att, r)**
6. Selection Cardinality (for each attribute in r, *expected* size of a selection: $\sigma_{att = K}$ (r ) ) **SC(att, r)**

## 14.4.2 Cost-Based Optimization

A cost-based optimizer generates a range of query-evaluation plans from the given query by using the equivalence rules, and chooses the one with the least cost. For a complex query, the number of different query plans that are equivalent to a given plan can be large. As an illustration, consider the expression

$$r_1 \bowtie r_2 \bowtie \cdots \bowtie r_n$$

where the joins are expressed without any ordering. With $n = 3$, there are 12 different join orderings:

$$\begin{array}{llll}
r_1 \bowtie (r_2 \bowtie r_3) & r_1 \bowtie (r_3 \bowtie r_2) & (r_2 \bowtie r_3) \bowtie r_1 & (r_3 \bowtie r_2) \bowtie r_1 \\
r_2 \bowtie (r_1 \bowtie r_3) & r_2 \bowtie (r_3 \bowtie r_1) & (r_1 \bowtie r_3) \bowtie r_2 & (r_3 \bowtie r_1) \bowtie r_2 \\
r_3 \bowtie (r_1 \bowtie r_2) & r_3 \bowtie (r_2 \bowtie r_1) & (r_1 \bowtie r_2) \bowtie r_3 & (r_2 \bowtie r_1) \bowtie r_3
\end{array}$$

In general, with $n$ relations, there are $(2(n-1))!/(n-1)!$ different join orders. (We leave the computation of this expression for you to do in Exercise 14.10.) For joins involving small numbers of relations, this number is acceptable; for example, with $n = 5$, the number is 1680. However, as $n$ increases, this number rises quickly. With $n = 7$, the number is 665280; with $n = 10$, the number is greater than 17.6 billion!

Luckily, it is not necessary to generate all the expressions equivalent to a given expression. For example, suppose we want to find the best join order of the form

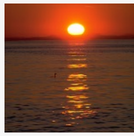
$$(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$$

## Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - 4 e.g. number of tuples in each relation, size of tuples, etc.
- In this chapter we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression
- In Chapter 14
  - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

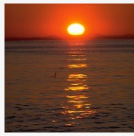
# Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful



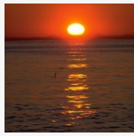
# Measures of Query Cost (Cont.)

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - $t_T$ – time to transfer one block
  - $t_S$ – time for one seek
  - Cost for b block transfers plus S seeks
    $b * t_T + S * t_S$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation



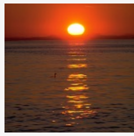
# Selection Operation

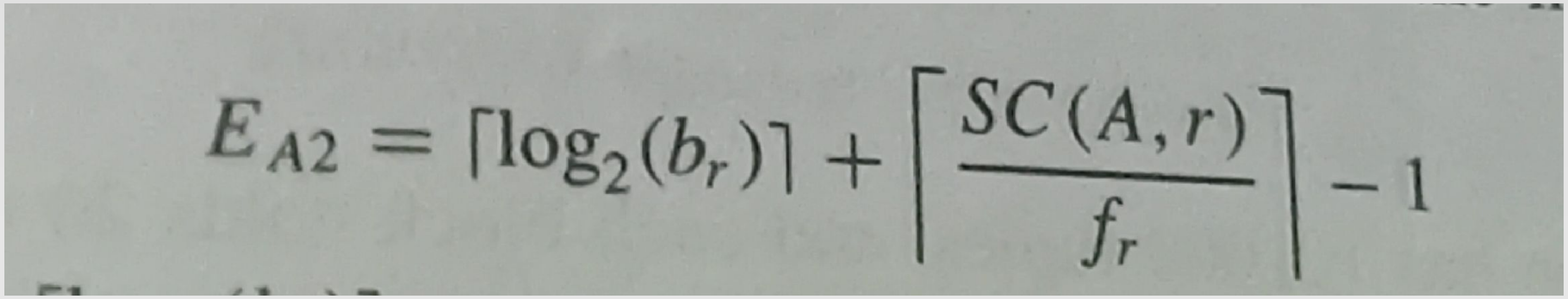
- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- Algorithm **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = $b_r$ block transfers + 1 seek
    - $b_r$ denotes number of blocks containing records from relation $r$
  - If selection is on a key attribute, can stop on finding record
    - cost = $(b_r/2)$ block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices



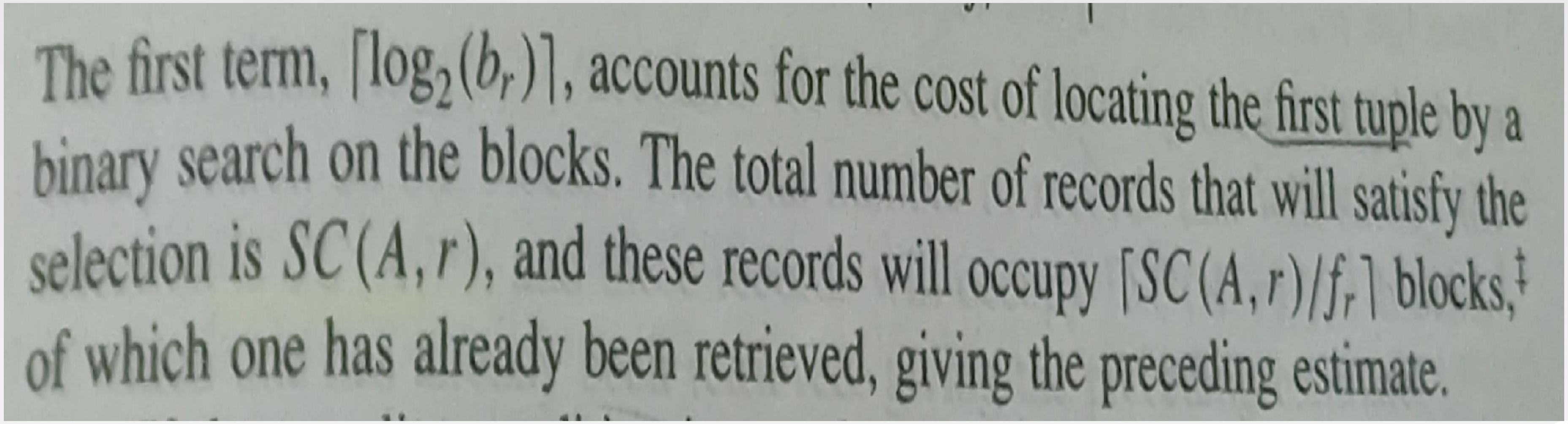
## Selection Operation (Cont.)

- **A2** *(binary search).* Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks
      - $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost in Chapter 14



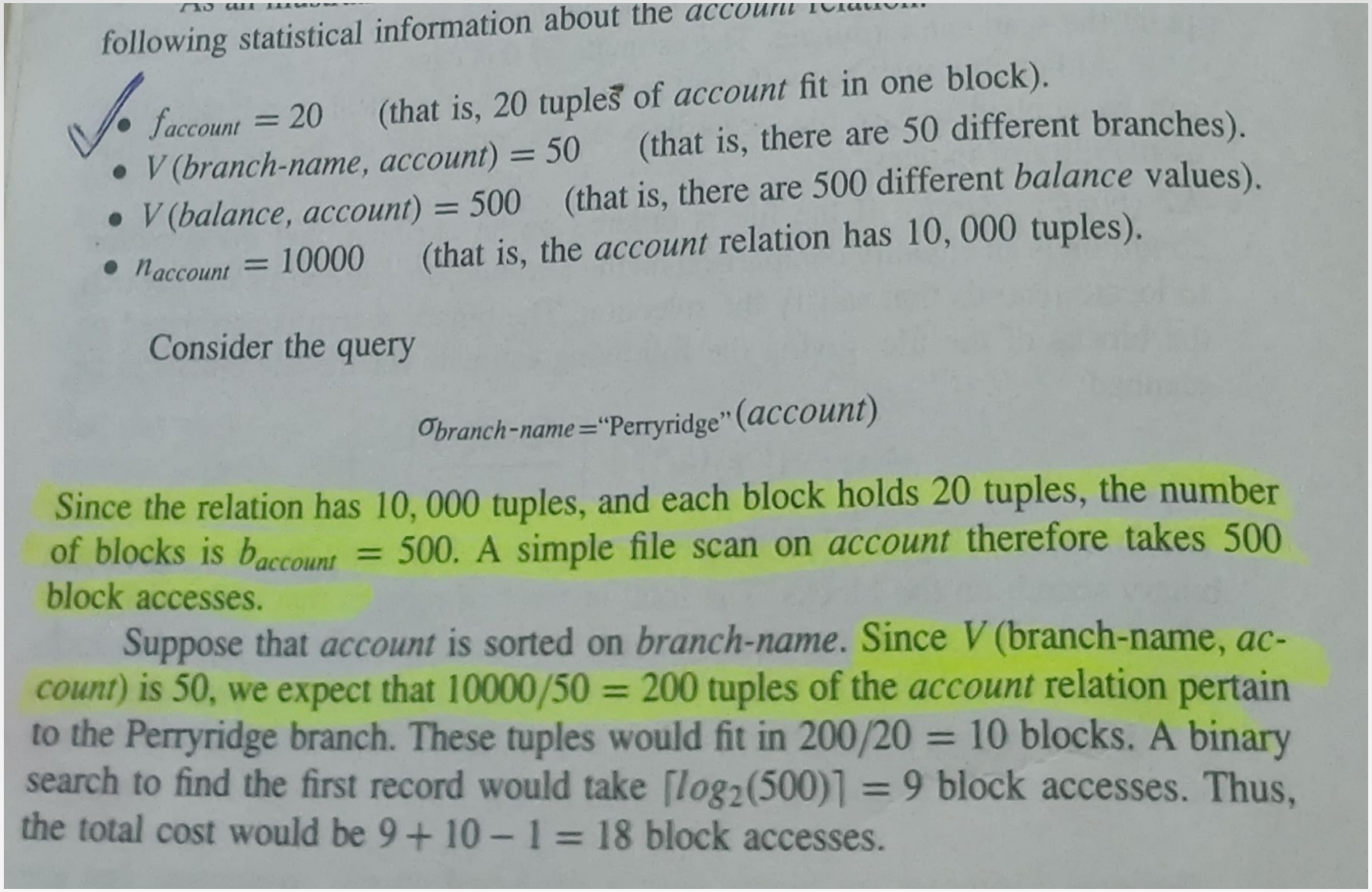


$$E_{A2} = \lceil \log_2(b_r) \rceil + \left\lceil \frac{SC(A,r)}{f_r} \right\rceil - 1$$



The first term, $\lceil \log_2(b_r) \rceil$, accounts for the cost of locating the first tuple by a binary search on the blocks. The total number of records that will satisfy the selection is $SC(A,r)$, and these records will occupy $\lceil SC(A,r)/f_r \rceil$ blocks,‡ of which one has already been retrieved, giving the preceding estimate.

# Statistics about account relation  Linear Vs Binary



following statistical information about the *account* relation:

- ✓ $f_{account} = 20$ (that is, 20 tuples of *account* fit in one block).
- $V(branch\text{-}name, account) = 50$ (that is, there are 50 different branches).
- $V(balance, account) = 500$ (that is, there are 500 different *balance* values).
- $n_{account} = 10000$ (that is, the *account* relation has 10,000 tuples).

Consider the query

$$\sigma_{branch\text{-}name = \text{"Perryridge"}}(account)$$

Since the relation has 10,000 tuples, and each block holds 20 tuples, the number of blocks is $b_{account} = 500$. A simple file scan on *account* therefore takes 500 block accesses.

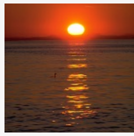
Suppose that *account* is sorted on *branch-name*. Since $V(\text{branch-name}, account)$ is 50, we expect that $10000/50 = 200$ tuples of the *account* relation pertain to the Perryridge branch. These tuples would fit in $200/20 = 10$ blocks. A binary search to find the first record would take $\lceil log_2(500) \rceil = 9$ block accesses. Thus, the total cost would be $9 + 10 - 1 = 18$ block accesses.

# Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A3** (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
- **A4** (*primary index on nonkey, equality)* Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - $Cost = h_i * (t_T + t_S) + t_S + t_T * \mathrm{b}$
- **A5** (*equality on search-key of secondary index).*
  - Retrieve a single record if the search-key is a candidate key
    - $Cost = (h_i + 1) * (t_T + t_S)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - Cost = $(h_i + n) * (t_T + t_S)$
      - Can be very expensive!

# Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:
- **A6** (*primary index, comparison).* (Relation is sorted on A)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A7** (*secondary index, comparison*).
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper



## Cardinality of Joins in General

Assume join: $R \bowtie S$

1. If R, S have no common attributes: $n_r * n_s$
2. If R,S have attribute A in common:

$$n_r \frac{n_s}{V(A,s)} \text{ (take min) or } n_s \frac{n_r}{V(A,r)}$$

3. If R, S have attribute A in common and:
   1. A is a candidate key for R: $\leq n_s$
   2. A is candidate key in R and candidate key in S : $\leq \min(n_r, n_s)$
   3. A is a key for R, foreign key for S: $= n_s$

# Join Operation

- Size and plans for join operation

depositor(cname, acct_no)
customer(cname, cstreet, ccity)

- Running example: let us compute: depositor $\bowtie$ customer

Metadata:

$n_{customer} = 10{,}000$ $n_{depositor} = 5000$

$f_{customer} = 25$ $f_{depositor} = 50$

$b_{customer} = 400$ $b_{depositor} = 100$

V(cname, depositor) = 2500 (each customer has on average 2 accts)

cname in depositor is foreign key

## Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

```
for each tuple $t_r$ in $r$ do begin
  for each tuple $t_s$ in $s$ do begin
        test pair $(t_r, t_s)$ to see if they satisfy the join condition $\theta$
        if they do, add $t_r \cdot t_s$ to the result.
  end
end
```

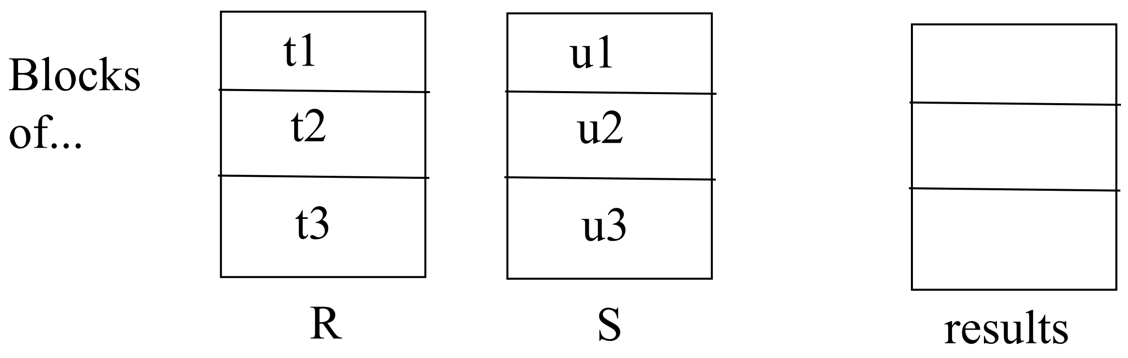
- *r* is called the **outer relation** and *s* the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

# Nested-Loop Join

Query: $R \bowtie_{\theta} S$

Algorithm 1: Nested Loop Join

**Idea:**



Compare: (t1, u1), (t1, u2), (t1, u3) .....

Then: GET NEXT BLOCK OF S

Repeat: **for EVERY tuple of R**

# Nested-Loop Join

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

  $$\mathbf{n_r * b_s + b_r}$$ block transfers, plus

  $n_r + b_r$ seeks
- If the smaller relation fits entirely in memory, use that as the inner relation.
  - ★ Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Assuming worst case memory availability cost estimate is
  - ★ with *depositor* as outer relation:
    - 5000 * 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - ★ with *customer* as the outer relation
    - 10000 * 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*depositor)* fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.

# Block Nested-Loop Join

- Block Nested Loop Join

```
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 begin
                Check if $(t_r, t_s)$ satisfy the join condition
                if they do ("match"), add $t_r \bullet t_s$ to the result.
```

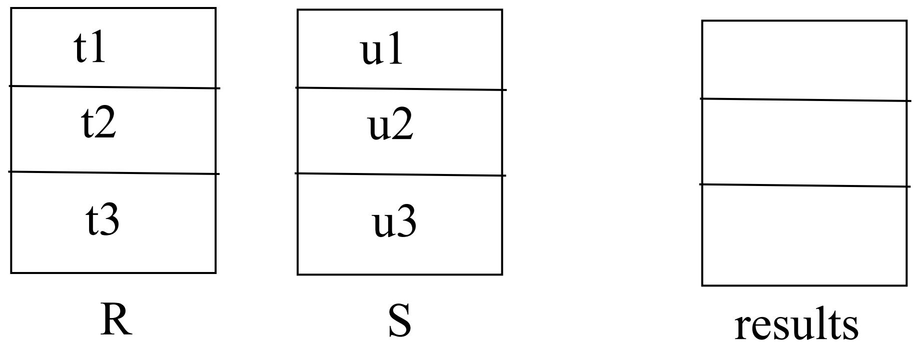
# Block Nested-Loop Join

Query: R $\bowtie_\theta$ S

Algorithm 2: Block Nested Loop Join

**Idea:**

Blocks of...



Compare: (t1, u1), (t1, u2), (t1, u3)
(t2, u1), (t2, u2), (t2, u3)
(t3, u1), (t3, u2), (t3, u3)

Then: GET NEXT BLOCK OF S

Repeat: **for EVERY BLOCK of R**

# Block Nested-Loop Join (Cont.)

Cost:

- Worst case estimate: $b_r * b_s + b_r$ **block accesses.**

# Block Nested Loop Join example

Alternative 1: Block Nested Loop

1a: customer = OUTER relation

depositor = INNER relation

cost: $b_{customer} + b_{depositor} * b_{customer} = 400 + (400 * 100) = 40{,}400$

1b: customer = INNER relation

depositor = OUTER relation

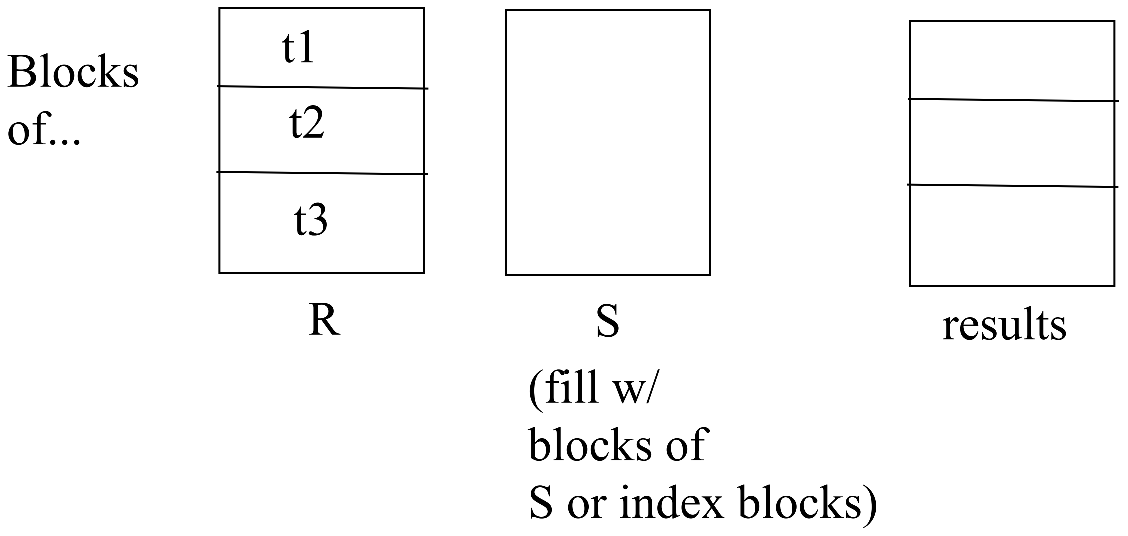
cost: $b_{depositor} + b_{depositor} * b_{customer} = 100 + (400 * 100) = 40{,}100$

# Indexed Nested-Loop Join

Query: $R \bowtie S$

Algorithm 3: Indexed Nested Loop Join

**Idea:**



For each tuple $t_i$ of R

if ti.A = K (A is the attribute R,S have in common)

then use the index to compute $\sigma_{att = K}(S)$

Demands: index on A for S

# Indexed Nested-Loop Join

Indexed Nested Loop Join

- For each tuple $t_R$ in the outer relation *R,* use the index to look up tuples in *S* that satisfy the join condition with tuple $t_R$.
- Worst case: buffer has space for only one page of *R*, and, for each tuple in *R*, we perform an index lookup on *s*.
- Cost of the join**:** $\mathbf{b_r + n_r * c}$
  - ★ Where *c* is the cost of traversing the index and fetching all matching *s* tuples for one tuple from *r*
  - ★ ***c* can be estimated as cost of a single selection on *s* using the join condition.**
- If indices are available on join attributes of both *R* and *S,* use the relation with fewer tuples as the outer relation.

# Example of Indexed Nested-Loop Join

Query: *depositor* $\bowtie$ *customer*

(cname, acct_no) (cname, ccity, cstreet)

Metadata:

customer: $n_{customer} = 10{,}000$ (Inner Relation with larger tuples)

$f_{customer} = 25$ $b_{customer} = 400$

depositor: $n_{depositor} = 5000$(outer Relation with smaller tuples)

$f_{depositor} = 50$ $b_{depositor} = 100$

V (cname, depositor) = 2500

i a primary index on cname (dense) for customer ($f_i = 20$)

Minimal buffer

# Indexed Nested-Loop Join

Alternative 2: Indexed Nested Loop

We have index on cname for customer. Depositor is the outer relation

Cost:

$b_{depositor} + n_{depositor} * c = 100 + (5000 * c)$ , c is the cost of evaluating a selection cname=K using index.

What is c? Primary index on cname, cname a key for customer

$$c = HT_i + 1$$

# Indexed Nested-Loop Join

What is $HT_i$ ?

cname a key for customer. V(cname, customer) = 10,000

$f_i$ = 20, i is dense

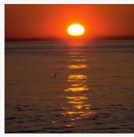
$LB_i = \lceil 10{,}000/20 \rceil = 500$

$HT_i \sim \lceil \log_{fi}(LB_i) \rceil + 1 = \lceil \log_{20} 500 \rceil + 1 = 4$

Cost of index nested loop is:
= 100 + (5000 * (4+1)) = 25,100 Block transfers (cheaper than Nested Loop Join)

## Sorting

- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, **external**
  **sort-merge** is a good choice.



## External Sort-Merge

Let $M$ denote memory size (in pages).

1. **Create sorted runs**. Let $i$ be 0 initially.

   Repeatedly do the following till the end of the relation:

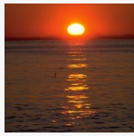
   (a) Read $M$ blocks of relation into memory

   (b) Sort the in-memory blocks

   (c) Write sorted data to run $R_i$; increment $i$.

   Let the final value of $i$ be $N$

2. *Merge the runs (next slide).....*

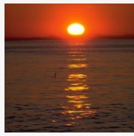


## External Sort-Merge (Cont.)

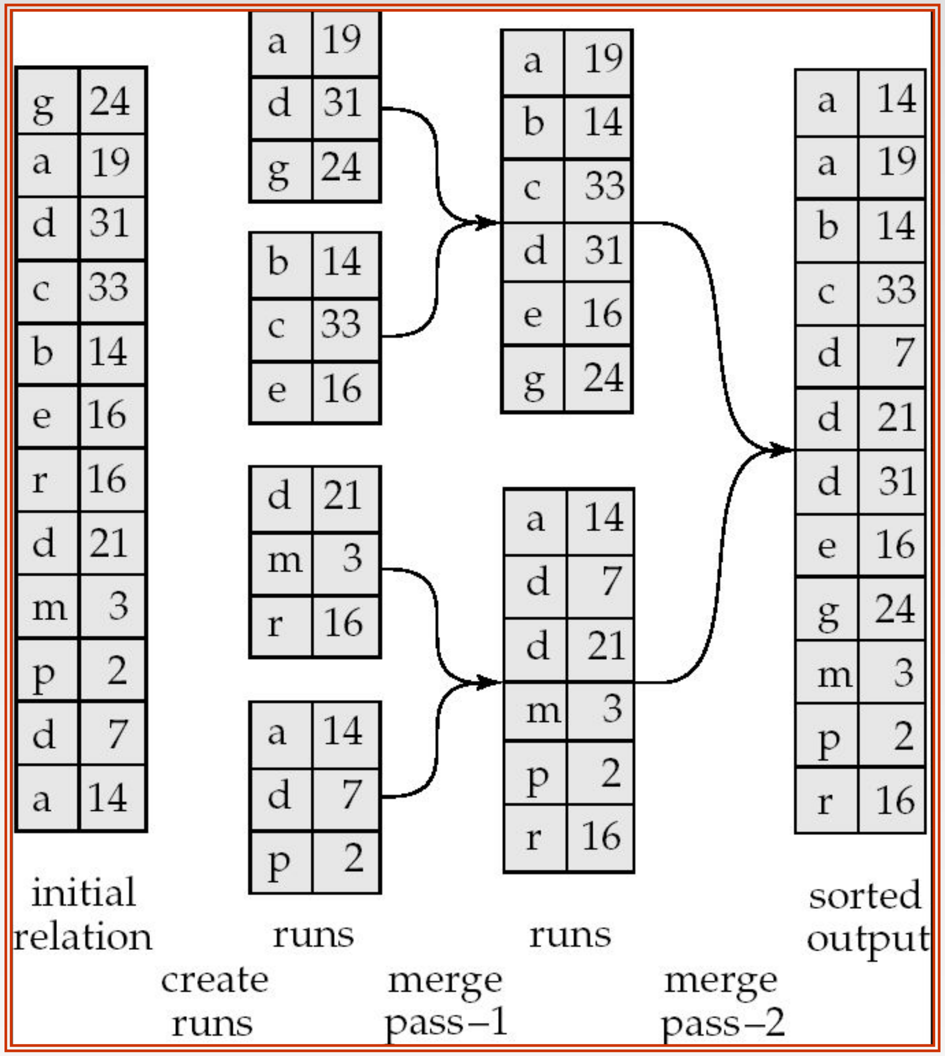
2. **Merge the runs (N-way merge)**. We assume (for now) that $N < M$.
   1. Use $N$ blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
   2. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      2. Write the record to the output buffer.  If the output buffer is full write it to disk.
      3. Delete the record from its input buffer page.
         **If** the buffer page becomes empty **then**
           read the next block (if any) of the run into the buffer.
   3. **until** all input buffer pages are empty:

## External Sort-Merge (Cont.)

- If $N \geq M$, several merge *passes* are required.
  - In each pass, contiguous groups of $M$ - 1 runs are merged.
  - A pass reduces the number of runs by a factor of $M$ -1, and creates runs longer by the same factor.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one.

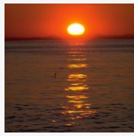


## Example: External Sorting Using Sort-Merge



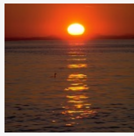
# External Merge Sort (Cont.)

- Cost analysis:
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Block transfers for initial run creation as well as in each pass is $2b_r$
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
    - Thus total number of block transfers for external sorting:
      $b_r ( 2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$
  - Seeks: next slide

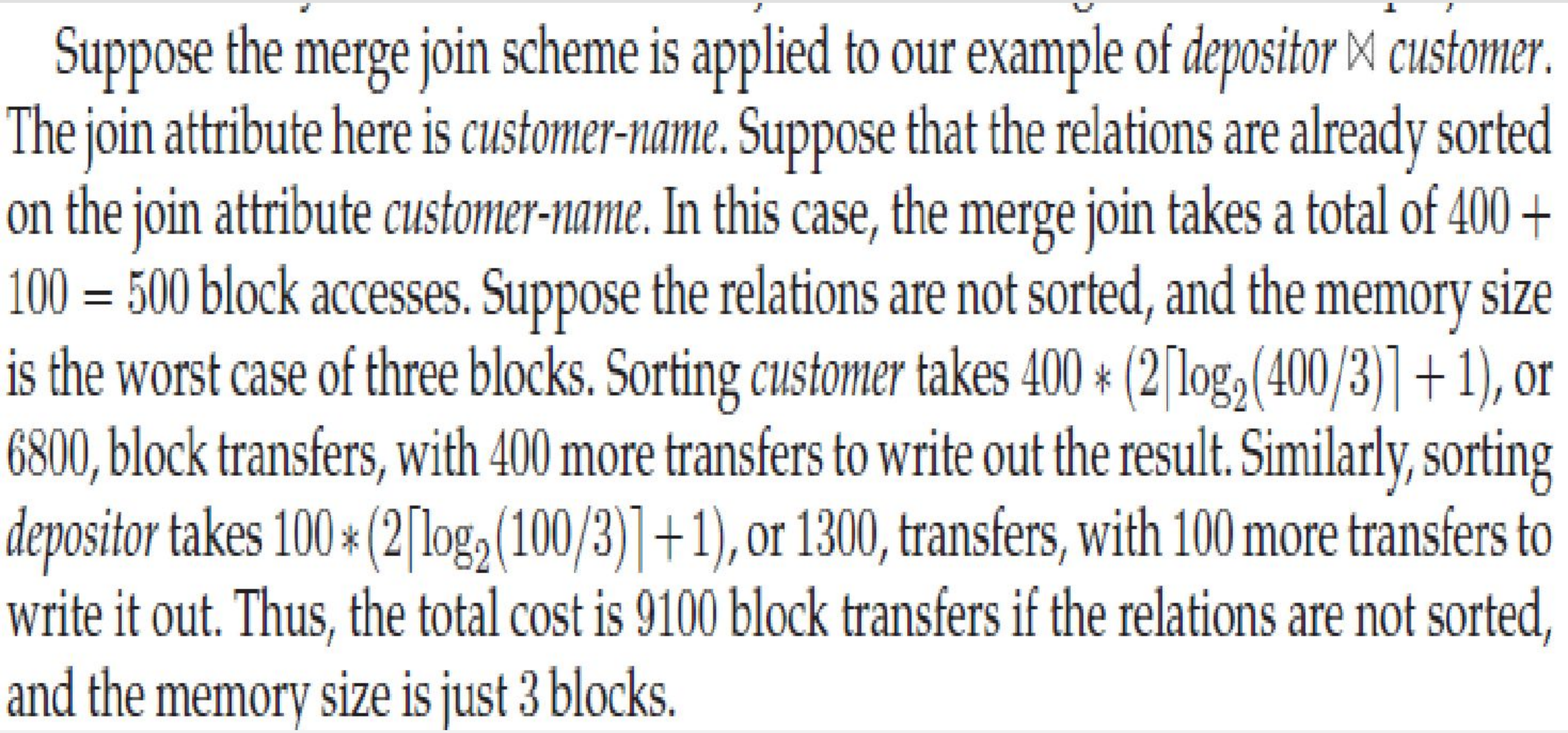


# External Merge Sort (Cont.)

- Cost of seeks
  - During run generation: one seek to read each run and one seek to write each run
    - $2 \lceil b_r / M \rceil$
  - During the merge phase
    - Buffer size: $b_b$ (read/write $b_b$ blocks at a time)
    - Need $2 \lceil b_r / b_b \rceil$ seeks for each merge pass
      - except the final one which does not require a write
    - Total number of seeks:
      $2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{M-1}(b_r / M) \rceil - 1)$



## Apply External Sort Merge on depositor join customer



Suppose the merge join scheme is applied to our example of *depositor* ⋈ *customer*. The join attribute here is *customer-name*. Suppose that the relations are already sorted on the join attribute *customer-name*. In this case, the merge join takes a total of 400 + 100 = 500 block accesses. Suppose the relations are not sorted, and the memory size is the worst case of three blocks. Sorting *customer* takes $400 * (2\lceil \log_2(400/3) \rceil + 1)$, or 6800, block transfers, with 400 more transfers to write out the result. Similarly, sorting *depositor* takes $100 * (2\lceil \log_2(100/3) \rceil + 1)$, or 1300, transfers, with 100 more transfers to write it out. Thus, the total cost is 9100 block transfers if the relations are not sorted, and the memory size is just 3 blocks.

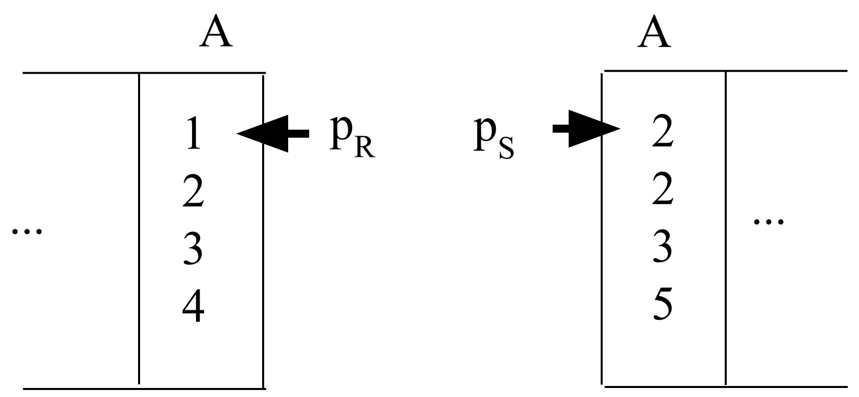
Sorting Customer relation =7200 Block Transfer [6800+400]
Sorting Depositor relation =1400 Block Transfer [1300+100]
Additional (400+100 Block Transfer are needed to write the result =500 Block Transfer
==□ 9100 Block Transfer )

# Merge Join Strategy

Query: R $\bowtie$ S

Algorithm: Merge Join

Idea: suppose R, S are both sorted on A (A is the common attribute)



Compare:

(1, 2) advance $p_R$
(2, 2) match, advance $p_S$ □ add to result
(2, 2) match, advance $p_S$ □ add to result
(2, 3) advance $p_R$
(3, 3) match, advance $p_S$ □ add to result
(3, 5) advance $p_R$
(4, 5) read next block of R

# Merge-Join

GIVEN R, S both sorted on A

1. Initialization
   - Reserve blocks of R, S into buffer reserving one block for result
   - Pr= 1, Ps =1
2. Join (assuming no duplicate values on A in R)

```
WHILE  !EOF( R) && !EOF(S) DO
 if  $B_R$[Pr].A == $B_S$[Ps].A then
            output to result; Ps++
      else if $B_R$[Pr].A < $B_S$[Ps].A then
            Pr++
 else (same for Ps)
 if Pr or Ps point past end of block,
     read next block and set Pr(Ps) to 1
```

## Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:

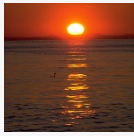
  $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks

  - + the cost of sorting if relations are unsorted.

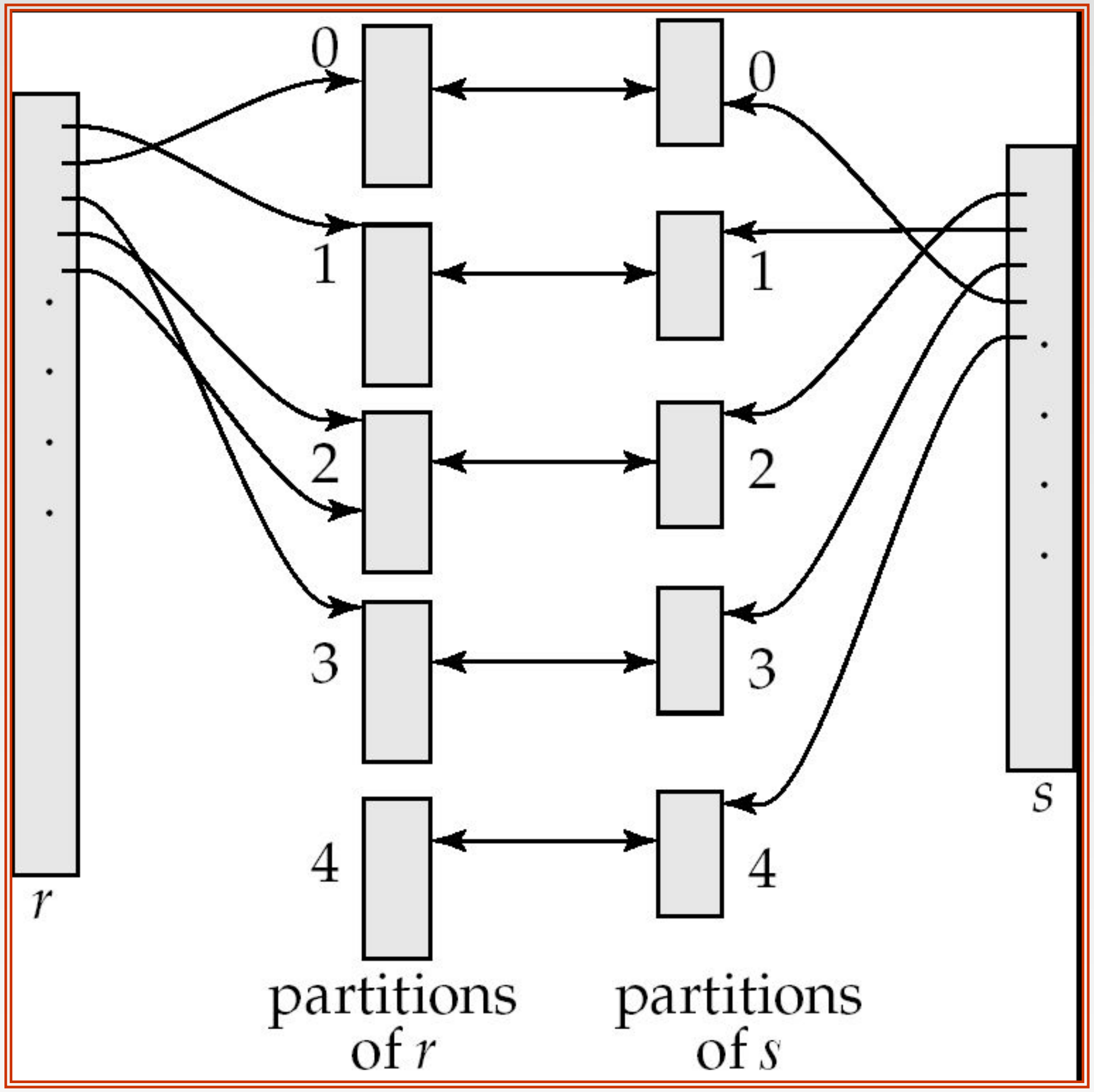
# Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function *h* is used to partition tuples of both relations
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}, where *JoinAttrs* denotes the common attributes of *r* and *s* used in the natural join.
  - **$r_0, r_1, \ldots, r_n$ denote partitions of *r* tuples**
    - 4 **Each tuple $t_r \in r$ is put in partition $r_i$ where $i = h(t_r[JoinAttrs])$.**
  - **$s_0, s_1, \ldots, s_n$ denotes partitions of *s* tuples**
    - 4 **Each tuple $t_s \in s$ is put in partition $s_i$, where $i = h(t_s[JoinAttrs])$.**

- *Note:* In book, $r_i$ is denoted as $H_{ri,}$ $s_i$ is denoted as $H_{si}$ and $n$ is denoted as $n_{h.}$



# Hash-Join (Cont.)



# Hash-Join (Cont.)

- *r* tuples in $r_i$ need only to be compared with *s* tuples in $s_i$ Need not be compared with *s* tuples in any other partition, since:
  - an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the *r* tuple has to be in $r_i$ and the *s* tuple in $s_i$.

# Hash-Join Algorithm

The hash-join of *r* and *s* is computed as follows.

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i:*

   (a) Load $s_i$ into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h*.

   (b) Read the tuples in $r_i$ from the disk one by one. For each tuple $t_r$ locate each matching tuple $t_s$ in $s_i$ using the in-memory hash index. Output the concatenation of their attributes.

Relation *s* is called the **build input** and *r* is called the **probe input.**

## Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is

  $3(b_r + b_s) + 4 * n_h$ block transfers +

  $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks

## Example of Cost of Hash-Join

$$customer \bowtie depositor$$

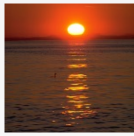
- Assume that memory size is 20 blocks
- $b_{depositor} = 100$ and $b_{customer} = 400$.
- *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *customer* into five partitions,each of size 80. This is also done in one pass.
- Therefore total cost, ignoring cost of writing partially filled blocks:
  - $3(100 + 400) = 1500$ block transfers +
    $2( \lceil 100/3\rceil + \lceil 400/3\rceil) = 336$ seeks

# Summary of all Formulas R ⋈ $_\theta$ S

| Sr.No | Name of Join | No. of Block transfers Needed. | Remarks |
|---|---|---|---|
| 1 | Nested Loop Join | **nr * bs + br block transfers** | No index Required |
| 2 | Block Nested Loop Join | **$b_r * b_s + b_r$ block transfers** | No index Required |
| 3 | Indexed Nested Loop Join | **br + nr * c block transfers** | Index required. Keep relation with larger number of tuples as inner relation, and fewer tuples as Outer relation. |
| 4 | Merge Join | **br + bs block transfers** | External sorting is needed to sort relation.Appiled only on sorted relations. |
| 5 | Hash Join | **3(br+ bs) block transfers** | Hashing is used. |

# Query Optimization

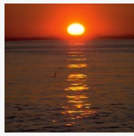
Database System Concepts 5th Ed.



## Equivalent Expressions in mathematics

- 2+3=3+2
- 3*2=2*3
  - Similar concept can be

    Applied to relational algebra……..expressions……..



## Transformation Example: Pushing Selections

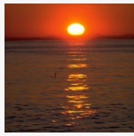
- Query: Find the names of all customers who have an account at some branch located in Brooklyn.

  $\Pi_{customer\_name}(\sigma_{branch\_city = \text{"Brooklyn"}}(branch \bowtie (account \bowtie depositor)))$

- Transformation using rule 7a.

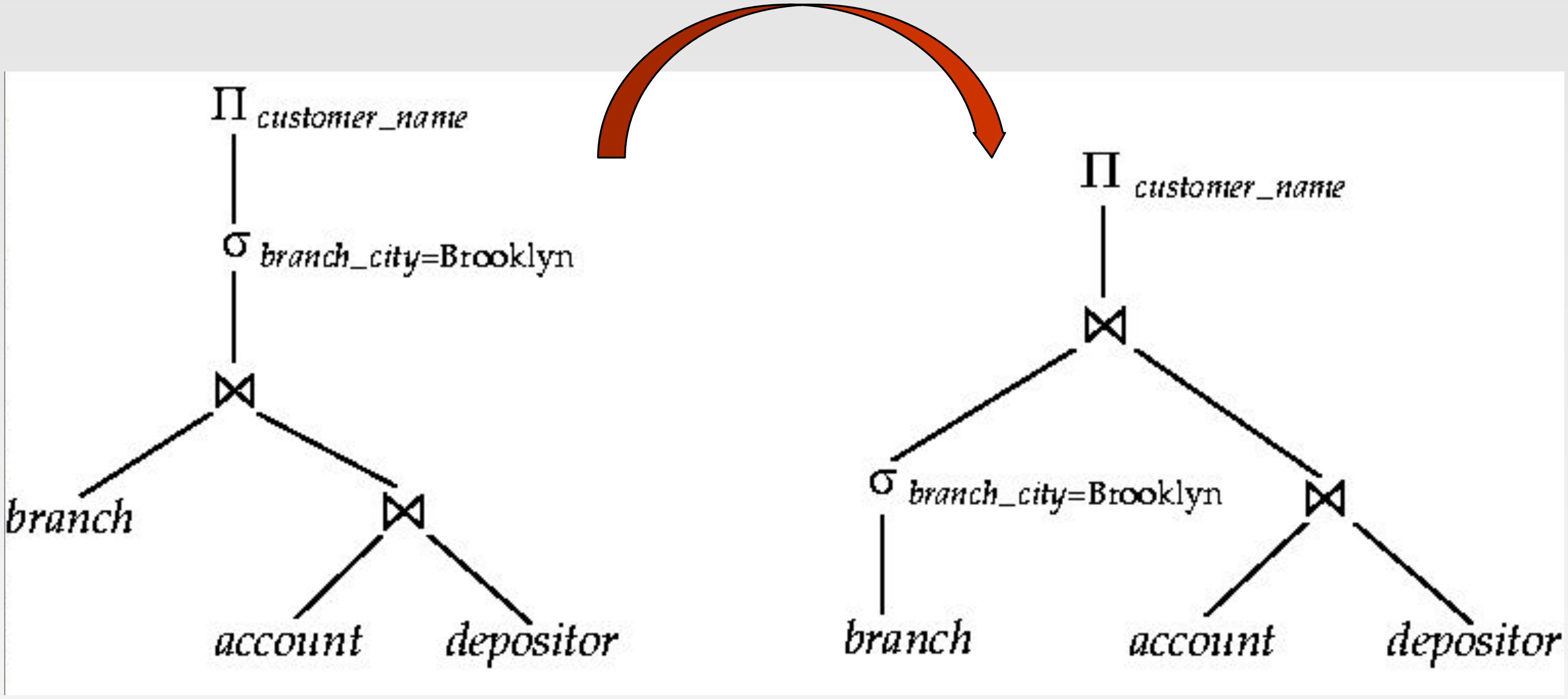
  $\Pi_{customer\_name}((\sigma_{branch\_city = \text{"Brooklyn"}}(branch)) \bowtie (account \bowtie depositor))$

- Performing the selection as early as possible reduces the size of the relation to be joined.



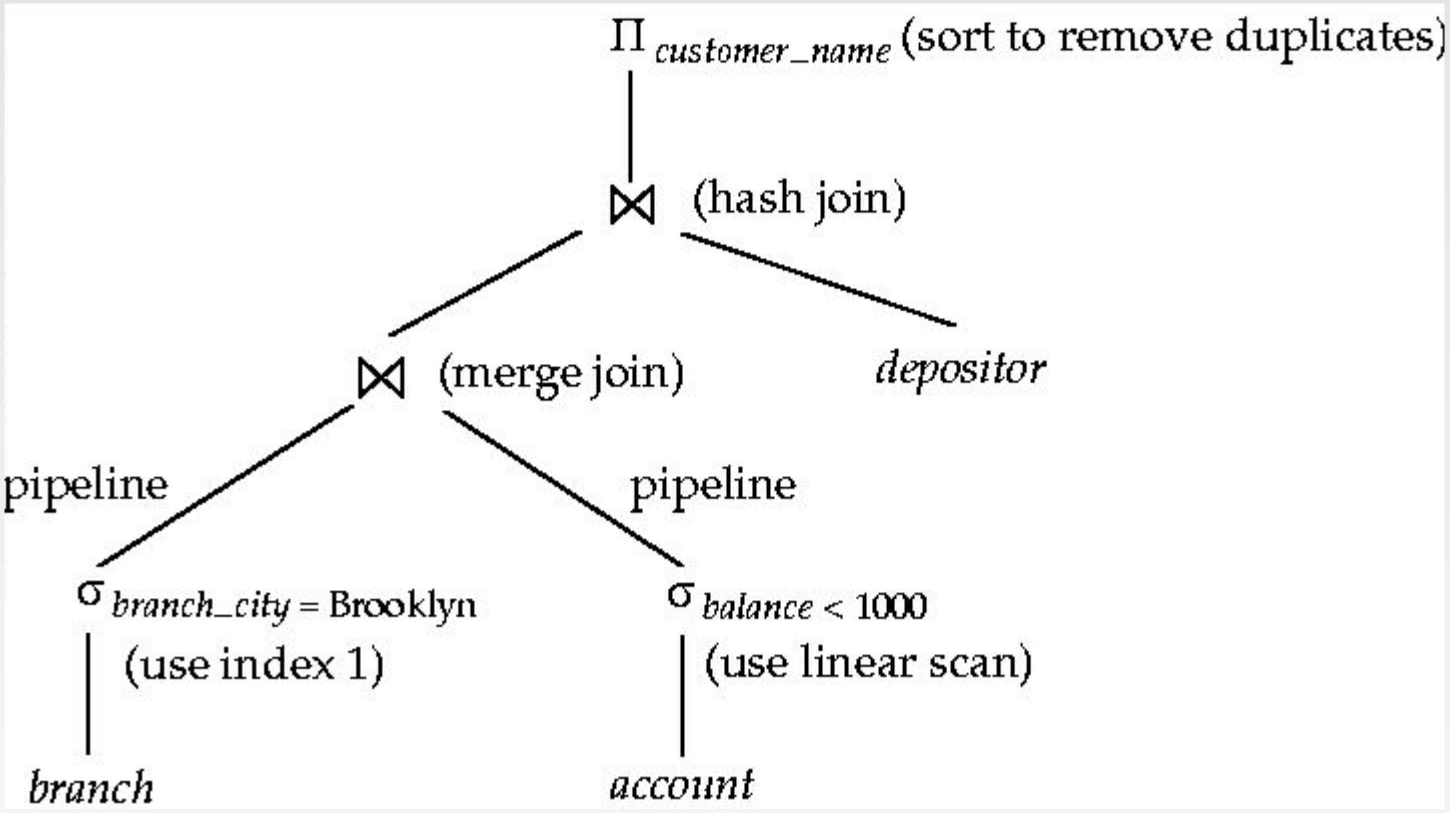
# Introduction

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation



# Introduction (Cont.)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.



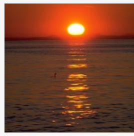
# Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

## Generating Equivalent Expressions (V.IMP)

# Transformation of Relational Expressions

- **Two relational algebra expressions are said to be equivalent** if the two expressions generate the same set of tuples on every legal database instance
  - Note: order of tuples is irrelevant
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa



# Equivalence Rules(Used in optimization)

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

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

2. Selection operations are commutative.

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

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

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

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \text{ X } E_2) = E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

# Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

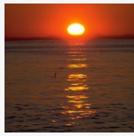
6. (a) Natural join operations are associative:

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

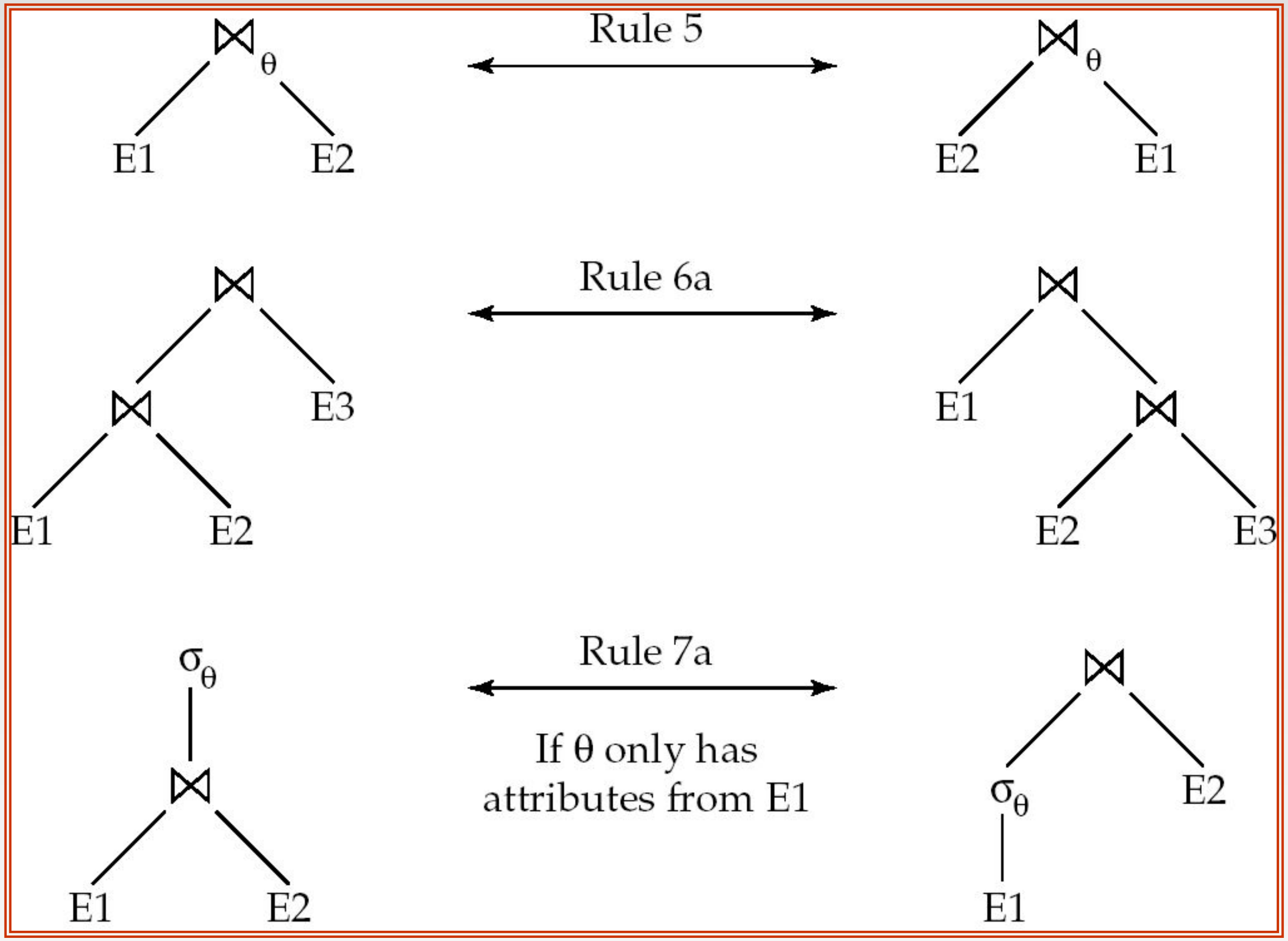
(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where $\theta_2$ involves attributes from only $E_2$ and $E_3$.



# Pictorial Depiction of Equivalence Rules



## Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in $\theta_0$ involve only the attributes of one of the expressions ($E_1$) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

   (b) When $\theta_1$ involves only the attributes of $E_1$ and $\theta_2$ involves only the attributes of $E_2$.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$$

# Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

   (a) if θ involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let $L_1$ and $L_2$ be sets of attributes from $E_1$ and $E_2$, respectively.
   - Let $L_3$ be attributes of $E_1$ that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - let $L_4$ be attributes of $E_2$ that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

## Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative
   $E_1 \cup E_2 = E_2 \cup E_1$
   $E_1 \cap E_2 = E_2 \cap E_1$
   - (set difference is not commutative).
10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

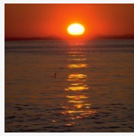
and similarly for $\cup$ and $\cap$ in place of $-$

Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$



## Example with Multiple Transformations

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

  $\Pi_{customer\_name}(\sigma_{branch\_city = \text{"Brooklyn"} \wedge balance > 1000}$
  $(branch \bowtie (account \bowtie depositor)))$

- Transformation using join associatively (Rule 6a):

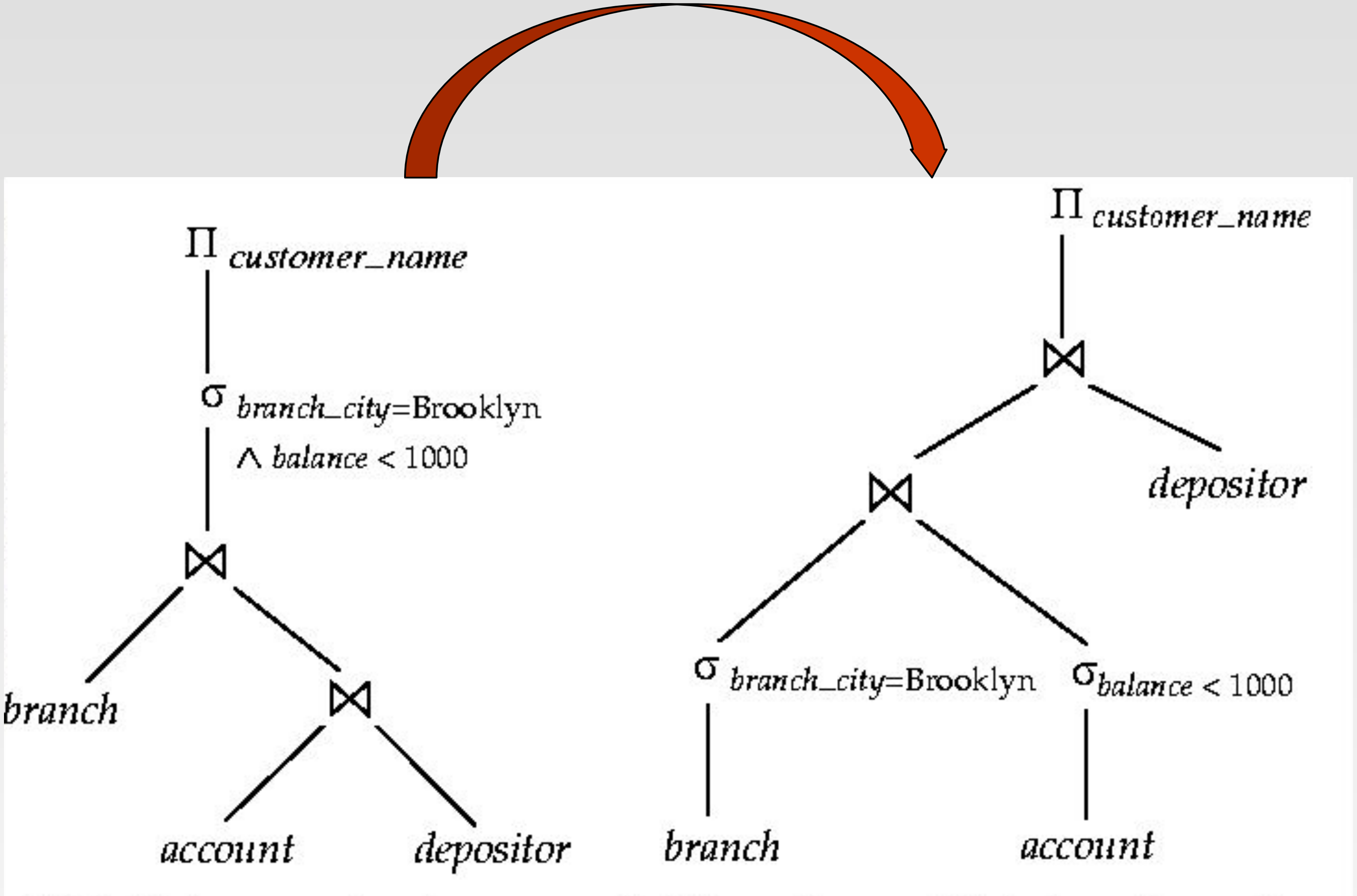
  $\Pi_{customer\_name}((\sigma_{branch\_city = \text{"Brooklyn"} \wedge balance > 1000}$
  $(branch \bowtie account)) \bowtie depositor)$

- Second form provides an opportunity to apply the "perform selections early" rule, resulting in the subexpression

  $\sigma_{branch\_city = \text{"Brooklyn"}}(branch) \bowtie \sigma_{balance > 1000}(account)$

- Thus a sequence of transformations can be useful

# Multiple Transformations (Cont.)

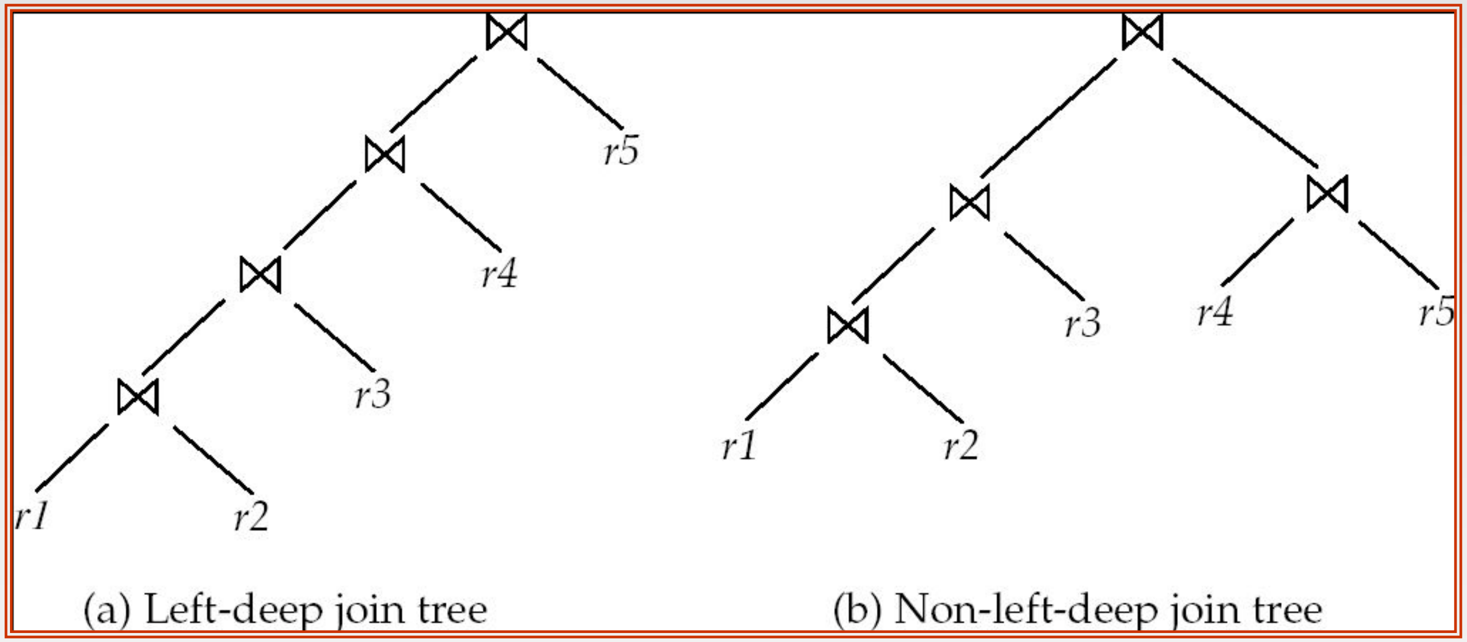


$\Pi_{customer\_name}$
$\sigma_{branch\_city = \text{Brooklyn} \wedge balance < 1000}$
$\bowtie$
branch
account
depositor
$\Pi_{customer\_name}$
$\bowtie$
depositor
$\sigma_{branch\_city = \text{Brooklyn}}$
$\sigma_{balance < 1000}$
branch
account

(a) Initial expression tree (b) Tree after multiple transformations

# Left Deep Join Trees

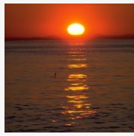
- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.

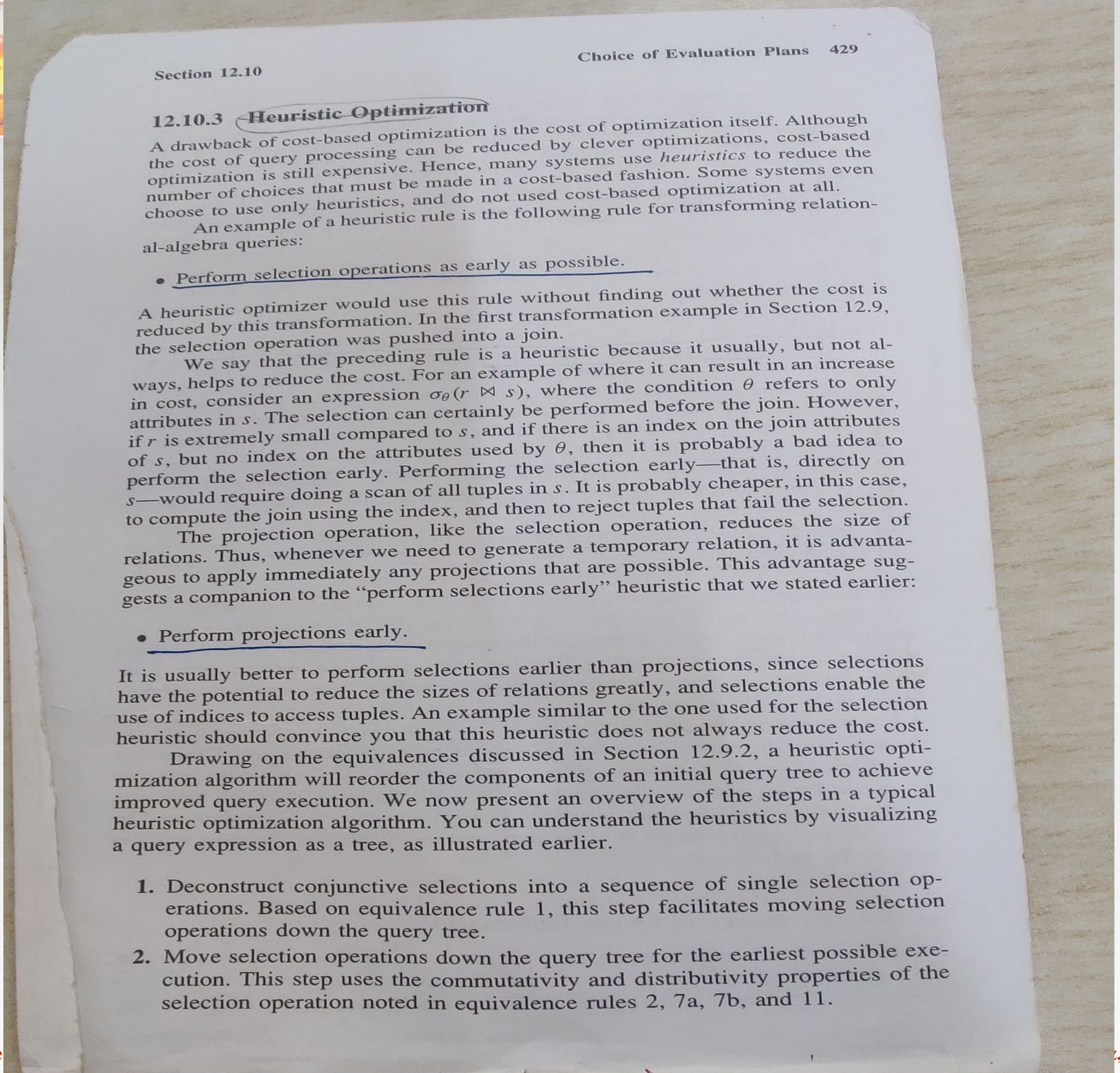


# Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Deconstruct conjunctive selections into a sequence of single selection operations.(Rule1),this step facilitates moving selection operations down the query tree.
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.

  - Some systems use only heuristics, others combine heuristics with partial cost-based optimization.





### 12.10.3 Heuristic Optimization

A drawback of cost-based optimization is the cost of optimization itself. Although the cost of query processing can be reduced by clever optimizations, cost-based optimization is still expensive. Hence, many systems use *heuristics* to reduce the number of choices that must be made in a cost-based fashion. Some systems even choose to use only heuristics, and do not used cost-based optimization at all.

An example of a heuristic rule is the following rule for transforming relational-algebra queries:

- Perform selection operations as early as possible.

A heuristic optimizer would use this rule without finding out whether the cost is reduced by this transformation. In the first transformation example in Section 12.9, the selection operation was pushed into a join.

We say that the preceding rule is a heuristic because it usually, but not always, helps to reduce the cost. For an example of where it can result in an increase in cost, consider an expression $\sigma_\theta(r \bowtie s)$, where the condition $\theta$ refers to only attributes in $s$. The selection can certainly be performed before the join. However, if $r$ is extremely small compared to $s$, and if there is an index on the join attributes of $s$, but no index on the attributes used by $\theta$, then it is probably a bad idea to perform the selection early. Performing the selection early—that is, directly on $s$—would require doing a scan of all tuples in $s$. It is probably cheaper, in this case, to compute the join using the index, and then to reject tuples that fail the selection.

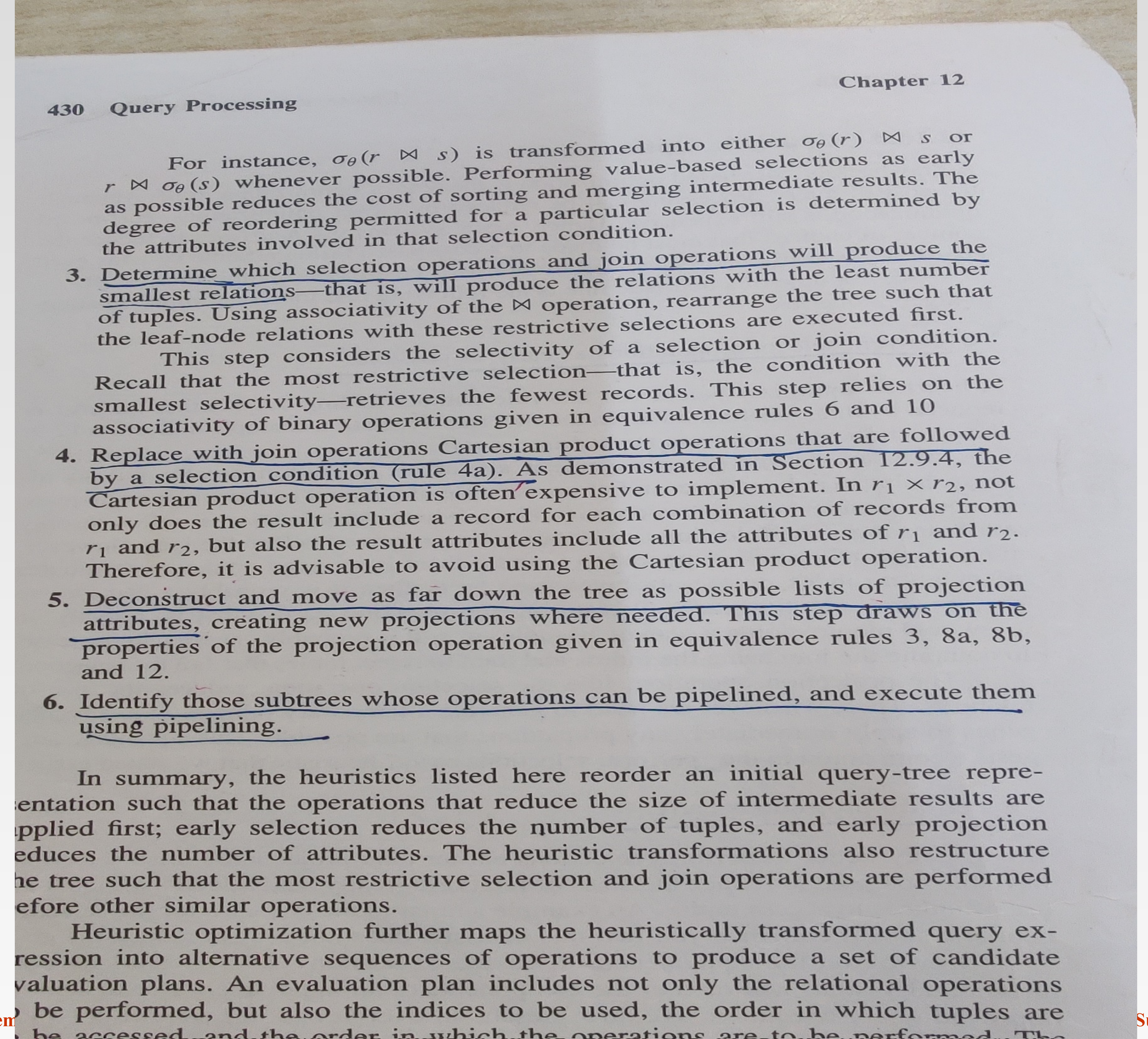
The projection operation, like the selection operation, reduces the size of relations. Thus, whenever we need to generate a temporary relation, it is advantageous to apply immediately any projections that are possible. This advantage suggests a companion to the "perform selections early" heuristic that we stated earlier:

- Perform projections early.

It is usually better to perform selections earlier than projections, since selections have the potential to reduce the sizes of relations greatly, and selections enable the use of indices to access tuples. An example similar to the one used for the selection heuristic should convince you that this heuristic does not always reduce the cost.

Drawing on the equivalences discussed in Section 12.9.2, a heuristic optimization algorithm will reorder the components of an initial query tree to achieve improved query execution. We now present an overview of the steps in a typical heuristic optimization algorithm. You can understand the heuristics by visualizing a query expression as a tree, as illustrated earlier.

1. Deconstruct conjunctive selections into a sequence of single selection operations. Based on equivalence rule 1, this step facilitates moving selection operations down the query tree.
2. Move selection operations down the query tree for the earliest possible execution. This step uses the commutativity and distributivity properties of the selection operation noted in equivalence rules 2, 7a, 7b, and 11.



For instance, $\sigma_\theta(r \bowtie s)$ is transformed into either $\sigma_\theta(r) \bowtie s$ or $r \bowtie \sigma_\theta(s)$ whenever possible. Performing value-based selections as early as possible reduces the cost of sorting and merging intermediate results. The degree of reordering permitted for a particular selection is determined by the attributes involved in that selection condition.

3. Determine which selection operations and join operations will produce the smallest relations—that is, will produce the relations with the least number of tuples. Using associativity of the $\bowtie$ operation, rearrange the tree such that the leaf-node relations with these restrictive selections are executed first.

   This step considers the selectivity of a selection or join condition. Recall that the most restrictive selection—that is, the condition with the smallest selectivity—retrieves the fewest records. This step relies on the associativity of binary operations given in equivalence rules 6 and 10

4. Replace with join operations Cartesian product operations that are followed by a selection condition (rule 4a). As demonstrated in Section 12.9.4, the Cartesian product operation is often expensive to implement. In $r_1 \times r_2$, not only does the result include a record for each combination of records from $r_1$ and $r_2$, but also the result attributes include all the attributes of $r_1$ and $r_2$. Therefore, it is advisable to avoid using the Cartesian product operation.

5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed. This step draws on the properties of the projection operation given in equivalence rules 3, 8a, 8b, and 12.

6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining.

In summary, the heuristics listed here reorder an initial query-tree repre-
entation such that the operations that reduce the size of intermediate results are
pplied first; early selection reduces the number of tuples, and early projection
educes the number of attributes. The heuristic transformations also restructure
he tree such that the most restrictive selection and join operations are performed
efore other similar operations.

Heuristic optimization further maps the heuristically transformed query ex-
ression into alternative sequences of operations to produce a set of candidate
valuation plans. An evaluation plan includes not only the relational operations
be performed, but also the indices to be used, the order in which tuples are
be accessed and the order in which the operations are to be performed. The

## Heuristic Optimization

- $\sigma_\theta( R \times S)$

$$\Downarrow$$

- $( R \bowtie_\theta S)$