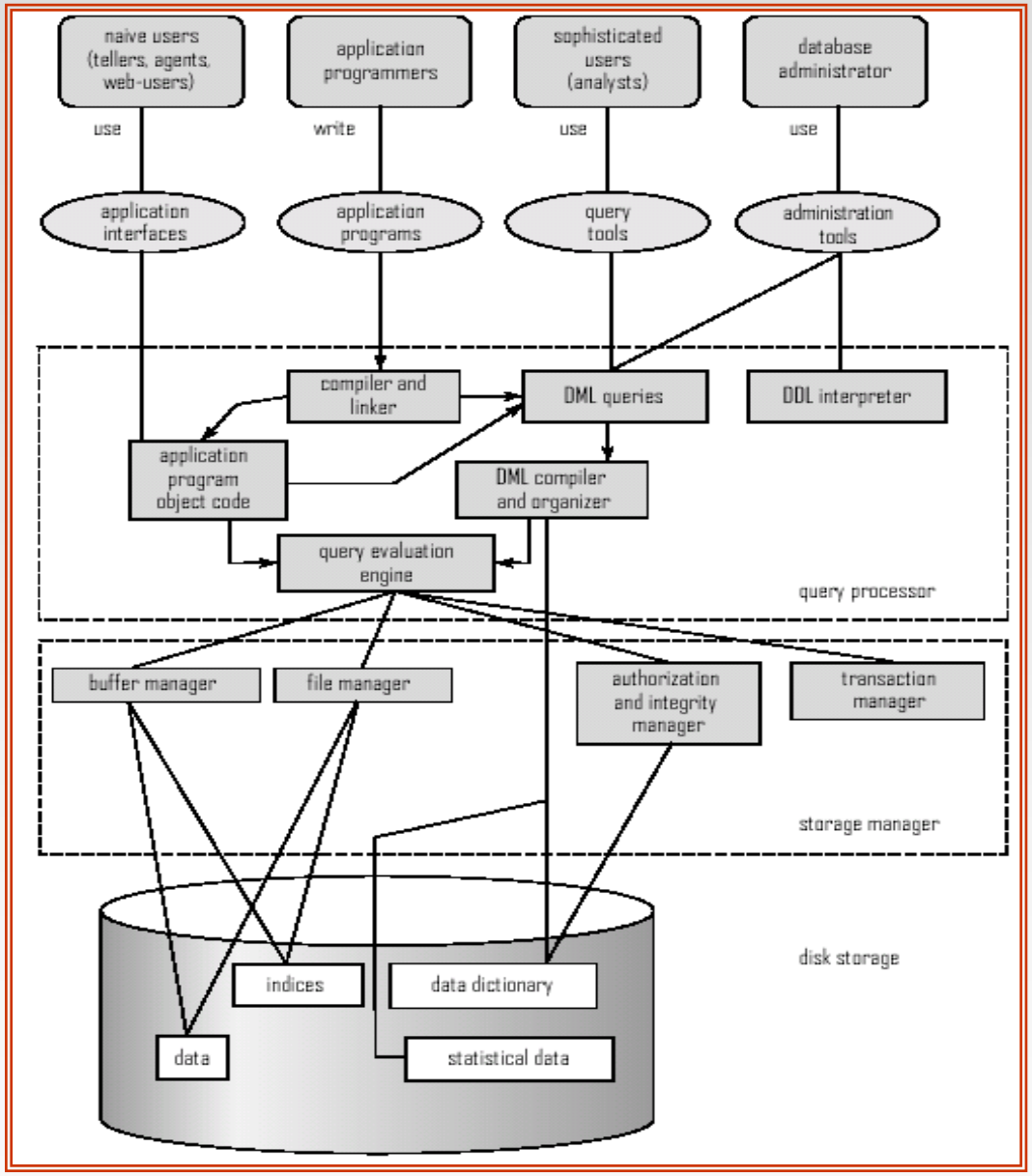
Query processing and optimization

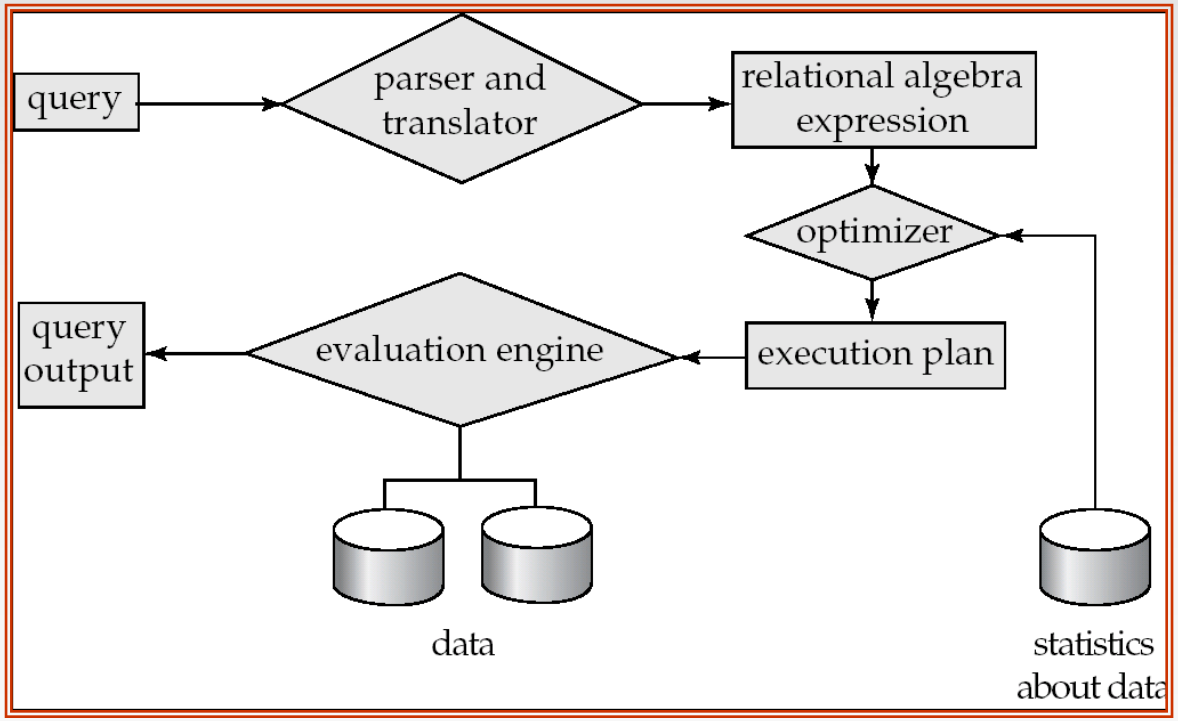
These slides are a modified version of the slides of the book "Database System Concepts" (Chapter 13 and 14), 5th Ed., McGraw-Hill, by Silberschatz, Korth and Sudarshan.
Original slides are available at www.db-book.com

DBMS: Overall Structure



Basic Steps in Query Processing

1. Parsing and translation: translate the query into its internal form. This is then translated into relational algebra. Parser checks syntax, verifies relations.
2. Optimization: A relational algebra expression may have many equivalent expressions. Generation of an **evaluation-plan.**
3. Evaluation: The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query



Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions

  E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$

  is equivalent to

  $\Pi_{balance}(\sigma_{balance<2500}(account))$

- Each relational algebra operation can be evaluated using one of several different **algorithms**

  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 $\geq$ 2500

- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.

Basic Steps

Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost.

- Cost is estimated using
  - Statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, number of distinct values for an attribute.
  - Statistics estimation for intermediate results to compute cost of complex expressions
  - Cost of individual operations
    - Selection Operation
    - Sorting
    - Join Operation
    - Other Operations

How to optimize queries, that is, how to find an evaluation plan with "good" estimated cost ?

Evaluation plan

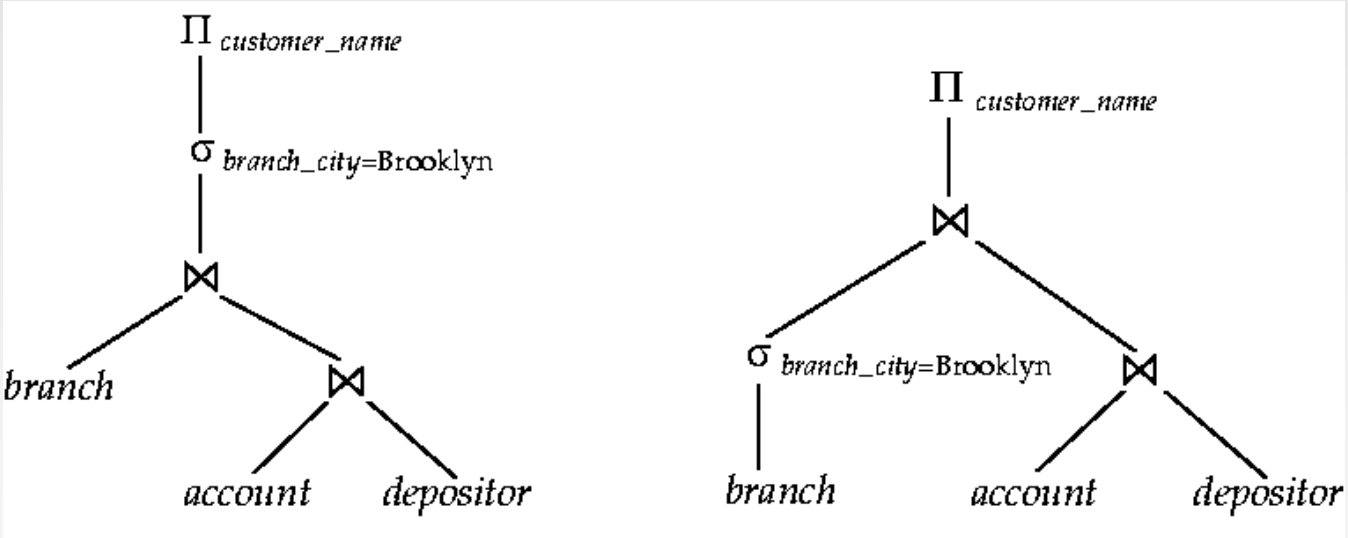
branch = (*branch_name*, *branch_city*, *assets*)

account = (*account_number*, *branch_name*, *balance*)

depositor = (*customer_id*, *customer_name*, *account_number)*

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

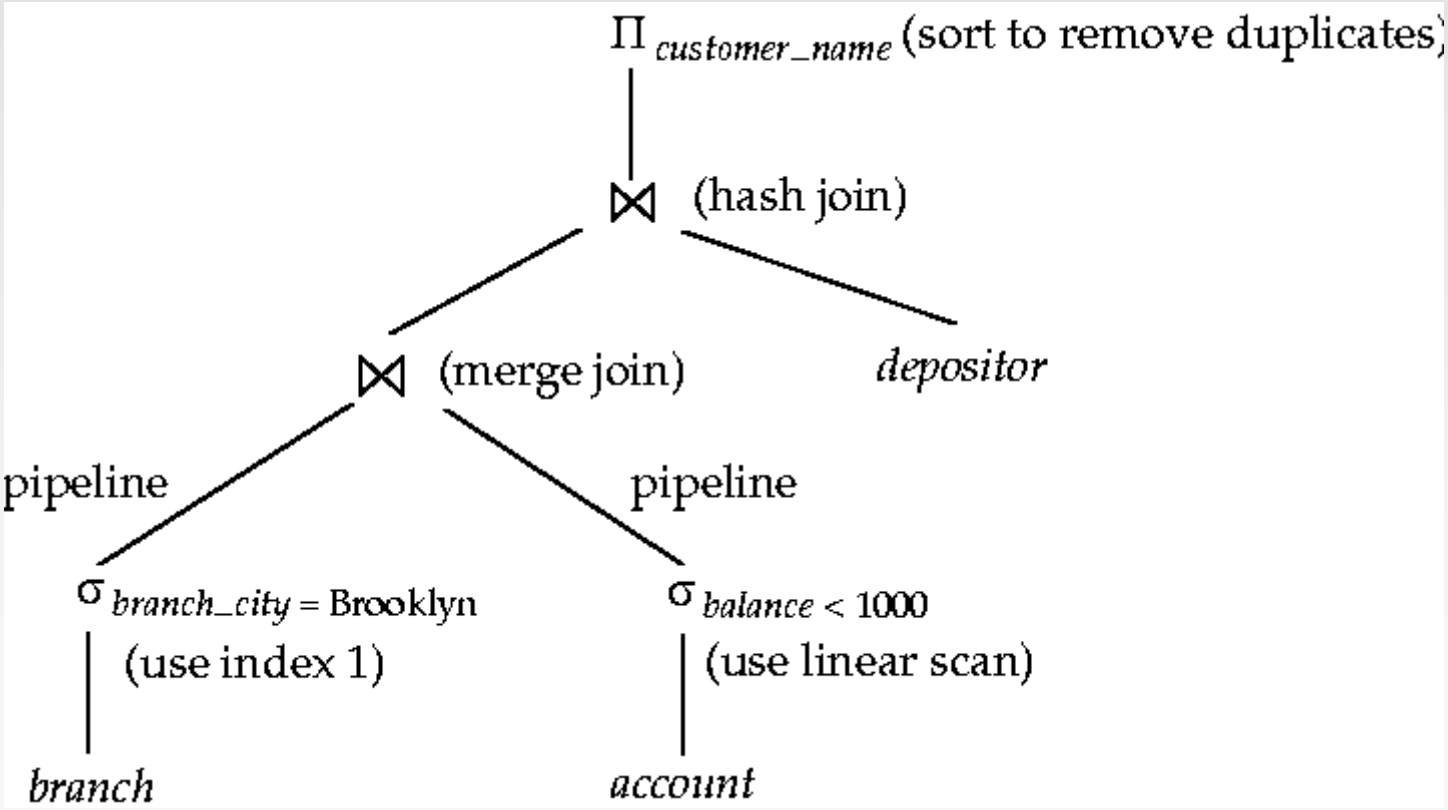
B - $\Pi_{customer\_name}((\sigma_{branch\_city=Brooklyn}(branch)) \bowtie (account \bowtie depositor))$



A B

Evaluation plan (cont.)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated (e.g., pipeline, materialization).



Cost-based query optimization

- 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**

Generating Equivalent Expressions

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

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.

$$\Pi_{L1}(\Pi_{L2}(\ldots \Pi_{Ln}(E))\ldots)) = \Pi_{L1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$
   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:

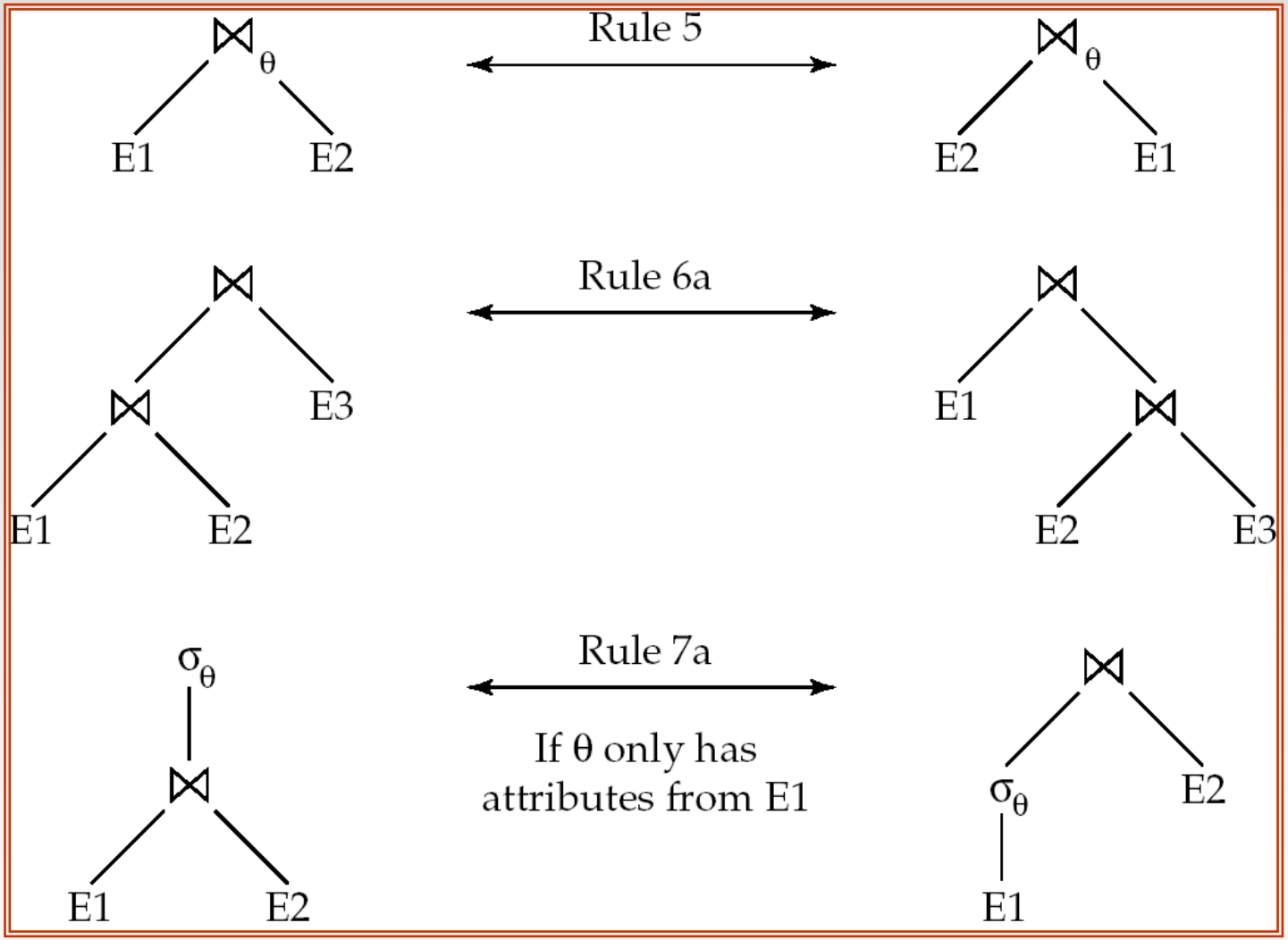
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(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 θ_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 θ_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 θ_1 involves only the attributes of E_1 and θ_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

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

Transformation: Pushing Selections down

- 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.

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):

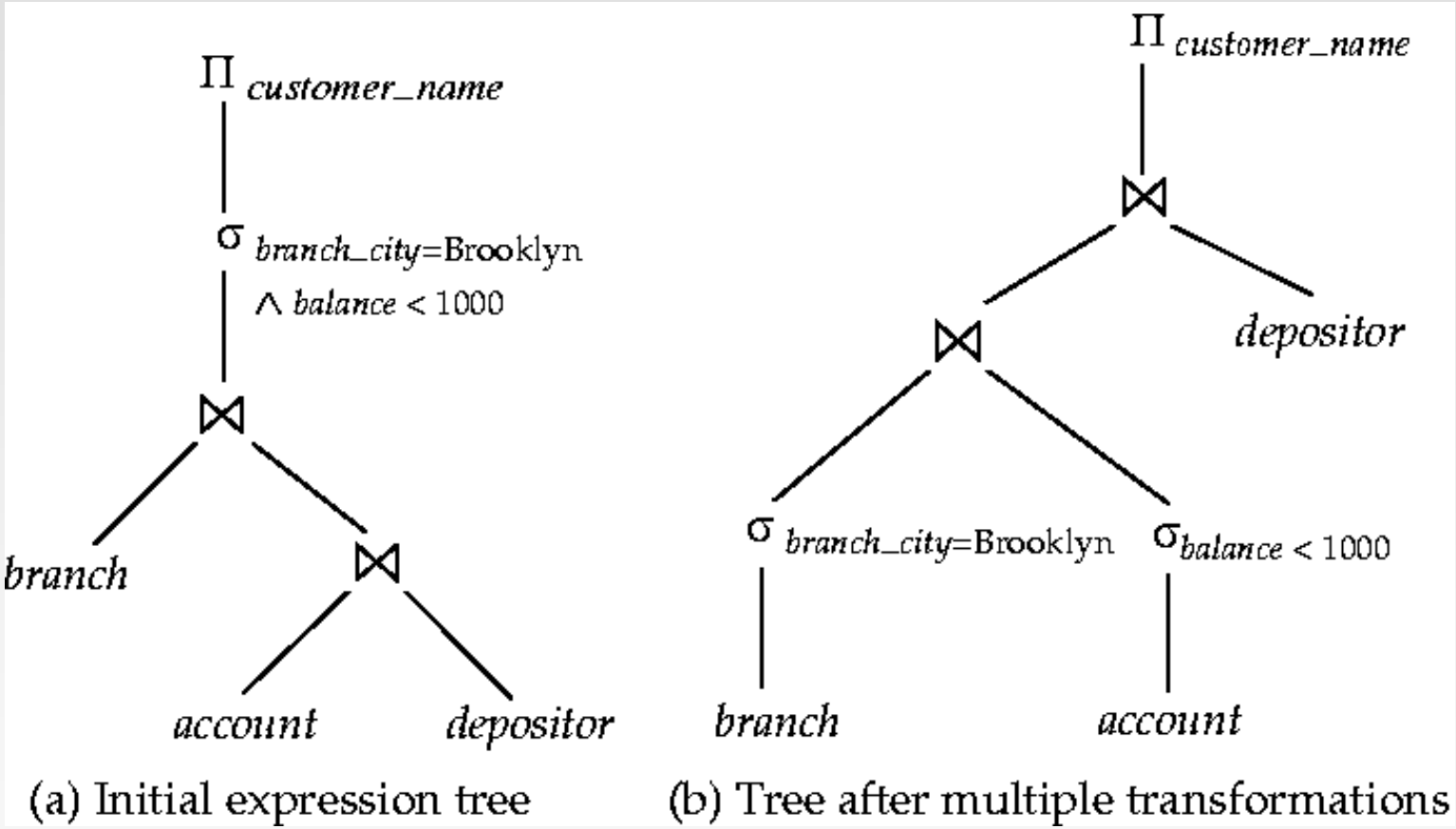
$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$$

- 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.)



Transformation: Pushing Projections down

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

- When we compute

  $$(\sigma_{branch\_city = \text{“Brooklyn”}} (branch) \bowtie account)$$

  we obtain a relation whose schema is:
  (*branch_name, branch_city, assets, account_number, balance)*
- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

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

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

Join Ordering Example

- For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

  (Join Associativity)

- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

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

  so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

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

- Could compute $account \bowtie depositor$ first, and join result with $\sigma_{branch\_city = \text{“Brooklyn”}}(branch)$ but $account \bowtie depositor$ is likely to be a large relation.
- Only a small fraction of the bank's customers are likely to have accounts in branches located in Brooklyn
  - it is better to compute

$$\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie account$$

first.

Measures of Query Cost

Measures of Query Cost

- Cost is the 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
  - Number of blocks read
  - Number of blocks written
    - 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.)

- **In this course**, for simplicity we just use the *number of block transfers from disk* as the cost measures (we ignore other costs for simplicity)
  - t_T – time to transfer one block
  - we do not include cost to writing output of the query 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
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation

Cost of individual operations

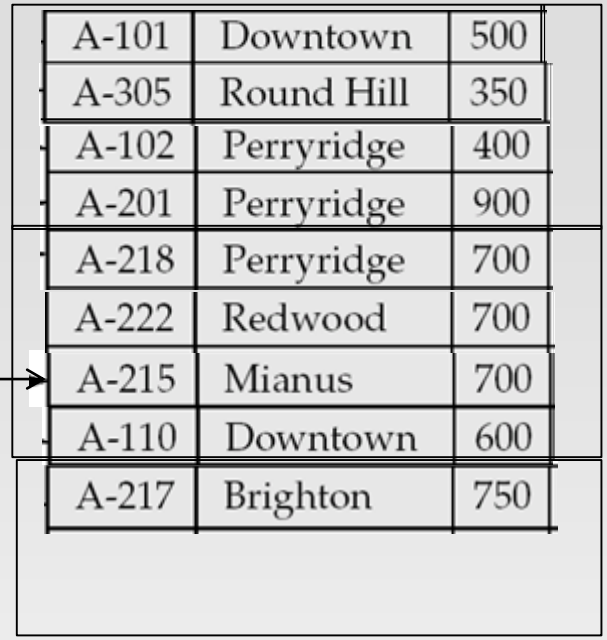
we assume that the buffer can hold only a few blocks of data, approximately one block for each relation

we use the number of block transfers from disk as the cost measure

we do not include cost to writing output of the query to disk in our cost formulae

Selection Operation

E.g., $\sigma_{balance<2500}(account)$



A-101	Downtown	500
A-305	Round Hill	350
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-215	Mianus	700
A-110	Downtown	600
A-217	Brighton	750

account file

File scan – search algorithms that locate and retrieve records in the file that fulfill a selection condition.

- *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
    - b_r number of blocks containing records from relation *r*

If selection is on a key attribute and an equality condition, can stop on finding record

- cost = (b_r/2) block transfers

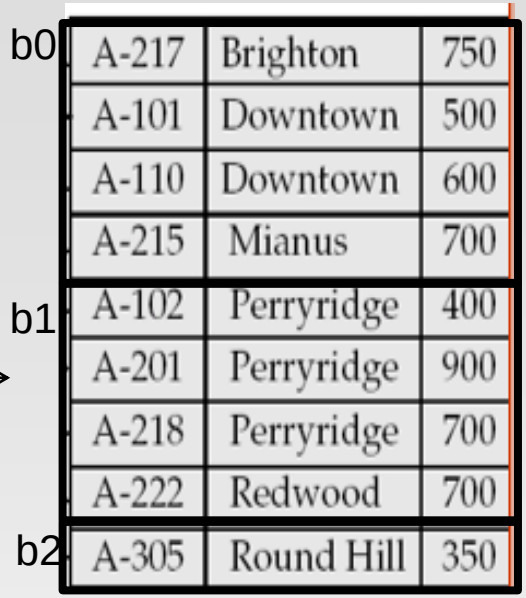
Linear search can always be applied regardless of the ordering of the file and the selection condition.

Selection Operation (Cont.)

- *A2. Binary search.* Applicable if selection is an equality comparison on the attribute on which file is ordered (sequentially ordered file on the attribute of the selection).

  E.g., $\sigma_{branch\text{-}name=\text{“Mianus”}}(account)$

  - 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
      - $\lceil \log_2(b_r) \rceil$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition



account file ordered on branch-name

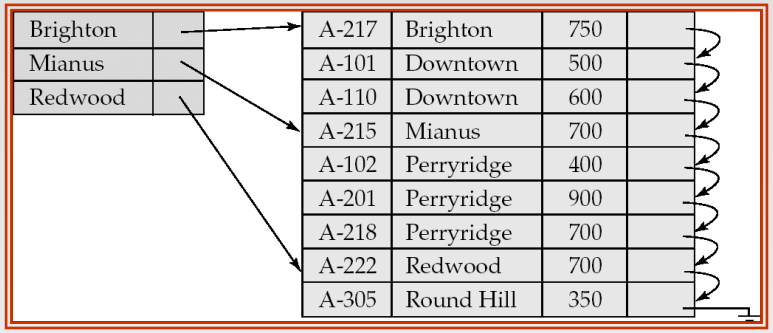
Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.

E.g., $\sigma_{branch\text{-}name=\text{“Mianus”}}(account)$

- *A3. Primary index on candidate key, equality*. Retrieve a single record that satisfies the corresponding equality condition
  - *Cost* = $(h_i + 1)$

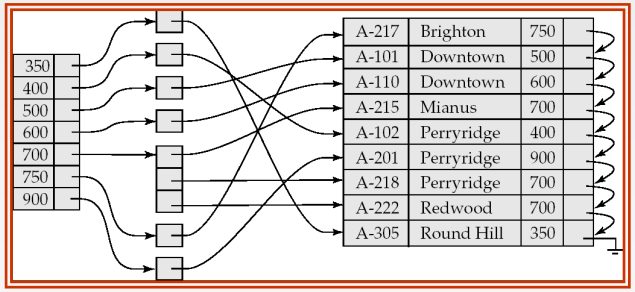
Primary *index*



- A4. P*rimary index on non key, equality.* Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records

  Cost = $(h_i + b)$

Secondary *index*



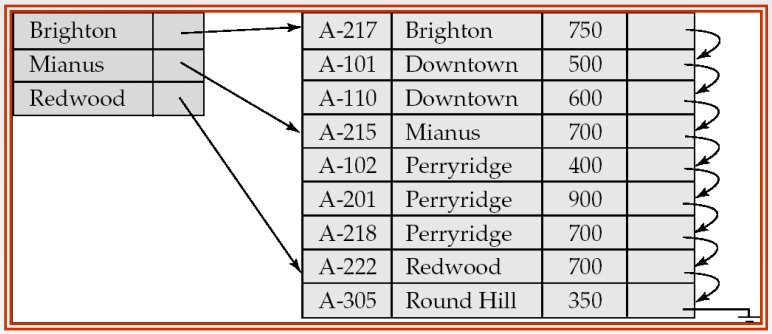
- A5. E*quality on search-key of secondary index.*
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* = $(h_i + 1)$
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - Cost = $(h_i + n)$
      - Can be very expensive!

Selections Involving Comparisons

$\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$

- 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 $\sigma_{A \ge V}(r)$ use index to find first tuple $\ge v$ and **scan relation** sequentially from there

    E.g. $\sigma_{\textit{branch-name} >= \text{“Mianus”}}(\textit{account})$



  - For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple > *v;* do not use index

    E.g. $\sigma_{\textit{branch-name} <= \text{“Mianus”}}(\textit{account})$

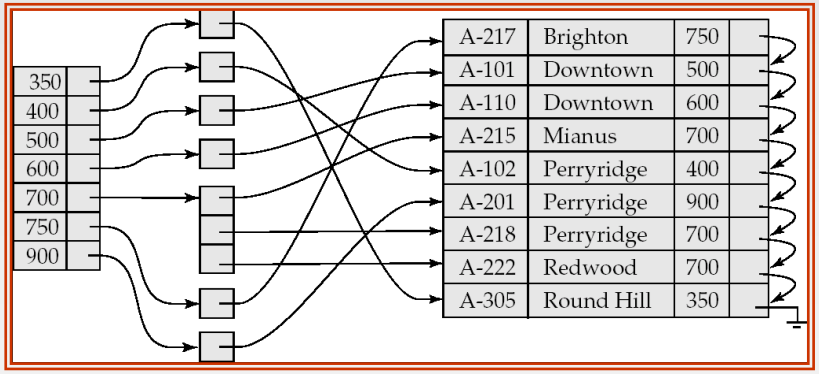
Selections Involving Comparisons

$\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$

A.7 Secondary index, comparison.

- For $\sigma_{A \ge V}(r)$ use index to find first index entry $\ge v$ and **scan index sequentially** from there, to find pointers to records.

  E.g. $\sigma_{balance >= 500}(account)$



- For $\sigma_{A \le V}(r)$ just **scan leaf pages of index** finding pointers to records, till first entry > v

  E.g. $\sigma_{balance <= 500}(account)$

- In either case, retrieve records that are pointed to
  - requires an I/O for each record
  - Linear file scan may be cheaper

Implementation of Complex Selections

Conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$

- *Conjunctive selection using one index.*
  - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
- *Conjunctive selection using multiple-key index*.
  - Use appropriate composite (multiple-key) index if available.
- *Conjunctive selection by intersection of identifiers, using more indices.*
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

Disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.

- *Disjunctive selection by union of identifiers*.
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file

Negation: $\sigma_{\neg\theta}(r)$

- Use linear scan on file
- If very few records satisfy $\neg\theta$, and an index is applicable to θ
  - Find satisfying records using index and fetch from file

Sorting

Sorting plays an important role in DBMS:

1) query can specify that the output be sorted

2) some operations can be implemented efficiently if the input relations are ordered (e.g., join)

- We may build an index on the relation, and then use the index to read the relation in sorted order. Records are ordered logically rather than physically. May lead to one disk block access for each tuple. Sometimes is desirable to order the records physically

- 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.

Sorting records in a file

External Sort-Merge

Let *M* denote the number disk blocks whose contents can be buffered in main memory

1. **Create sorted runs.**

   Let *i* be 0 initially.

   Repeat

   (a) Read *M* blocks of relation into memory
   (b) Sort the in-memory blocks
   (c) Write sorted data to run file R_i;
   (d) increment *i.*

   until the end of the relation

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

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.)

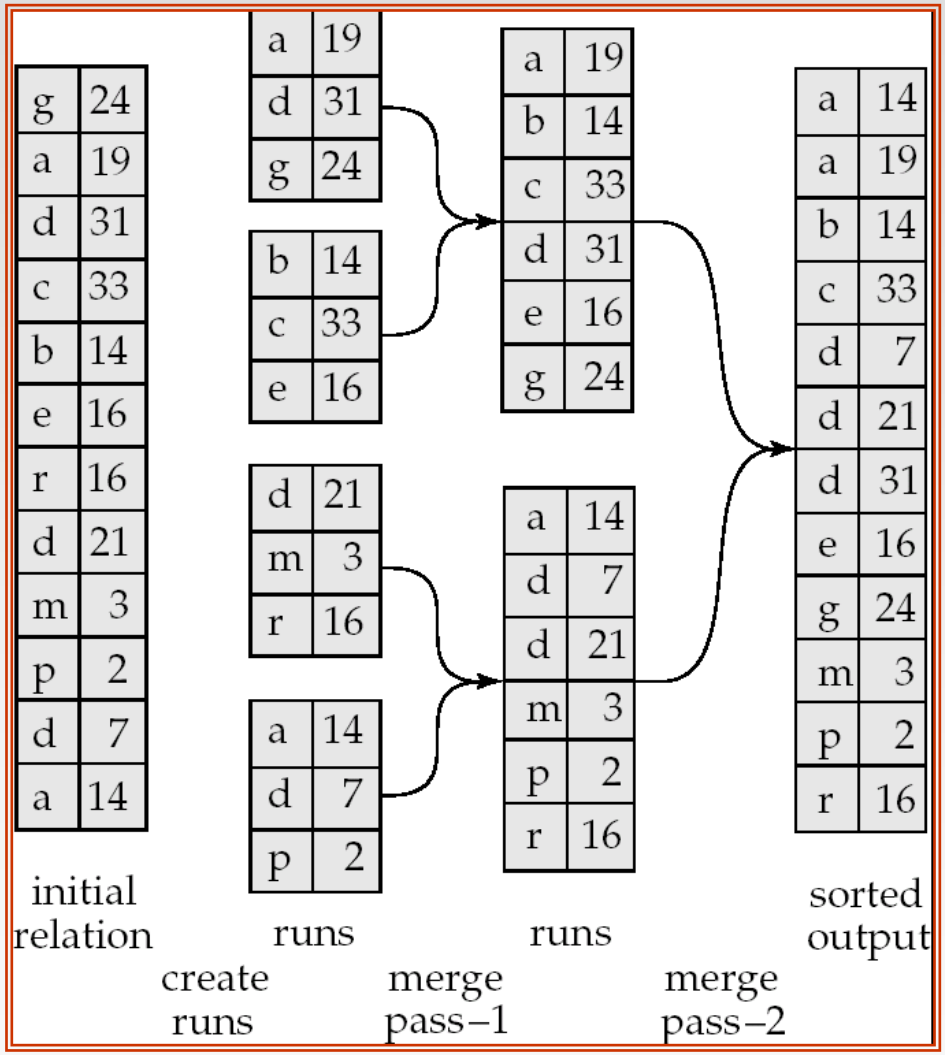
If *N* ≥ *M*, several merge *passes* are required ((M-1)-way merge).

- 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

$f_r = 1$ M = 3

- tranfer 3 blocks
- sort records
- store in a run



External Merge Sort (Cont.)

- Cost analysis:
  - Initial number of runs: $\lceil b_r/M \rceil$
  - Number of runs decrease of a factor M-1 in each merge pass. 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:

$$2b_r + 2b_r \lceil \log_{M-1}(b_r / M) \rceil - b_r$$

$$b_r \ (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$$

(- b_r because we do not include cost to writing output of the query to disk)

In the example:

$12 \ (2 \lceil \log_2 (12 / 3) \rceil + 1) = 12 \ (2*2 + 1) = 60$ block transfers

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate

customer = (*customer_name*, customer_street, customer_city, …)
depositor = (*customer_name*, *account_number,….)*

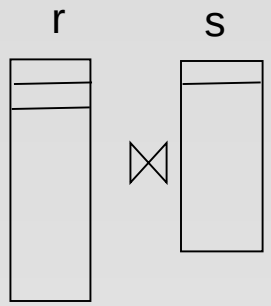
customer $\bowtie$ *depositor* *natural join*
customer $\bowtie_{customer.customer\_iname = depositor.customer\_iname}$ *depositor* *theta join*

- Examples use the following information

  Customer: Number of records 10.000 Number of blocks 400

  Depositor: Number of records 5.000 Number of blocks 100

Nested-Loop Join



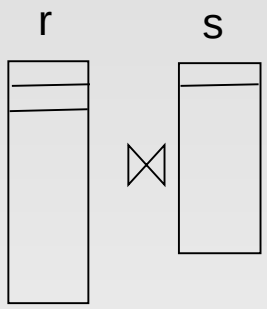
- To compute the theta join $r \bowtie_\theta s$

  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 θ
  if they do, add $t_r \bullet t_s$ 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 (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  $n_r * b_s + b_r$ block transfers

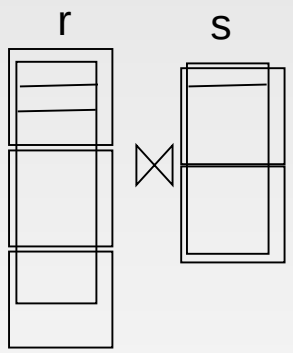


- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers
- Assuming worst case memory availability cost estimate is
  - with *depositor* as outer relation:
    - 5000 * 400 + 100 = 2,000,100 block transfers,
  - with *customer* as the outer relation
    - 10000 * 100 + 400 = 1,000,400 block transfers
- 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

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block B_r of r do begin
    for each block B_s of s do begin
        for each tuple t_r in B_r do begin
            for each tuple t_s in B_s do begin
                Check if (t_r,t_s) satisfy the join condition
                if they do, add t_r • t_s to the result.
            end
        end
    end
end
```

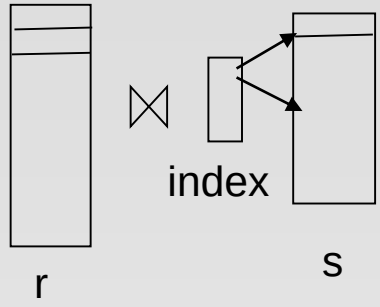


Block Nested-Loop Join (Cont.)

- Worst case estimate (if there is enough memory only to hold one block of each relation): $b_r * b_s + b_r$ block transfers
  - Each block in the inner relation *s* is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation
  - Assuming worst case memory availability cost estimate is
    - with *depositor* as outer relation:
      - 100 + 100*400 = 40,100 block transfers (outer the smallest more convenient)
    - with *customer* as the outer relation
      - 400 + 400*100 = 40,400 block transfers
- Best case (the smaller relation fits entirely in memory): $b_r + b_s$ block transfers.

- Improvements to nested loop and block nested loop algorithms:
  - Use index on inner relation if available (next slide)

Indexed Nested-Loop Join

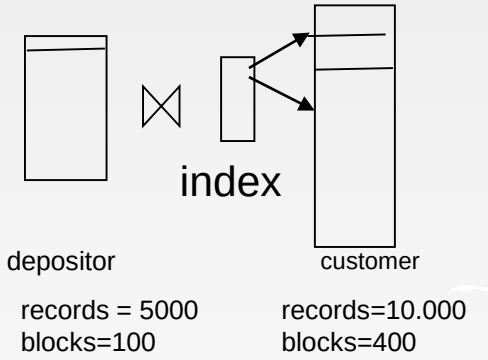


- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation *r,* use the index to look up tuples in *s* that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one block of *r*, and, for each tuple in *r*, we perform an index lookup on *s.*
- Cost of the join: $b_r + n_r * c$
  - Where *c* is the cost of traversing index and fetching all matching *s* tuples for one tuple or *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 Nested-Loop Join Costs

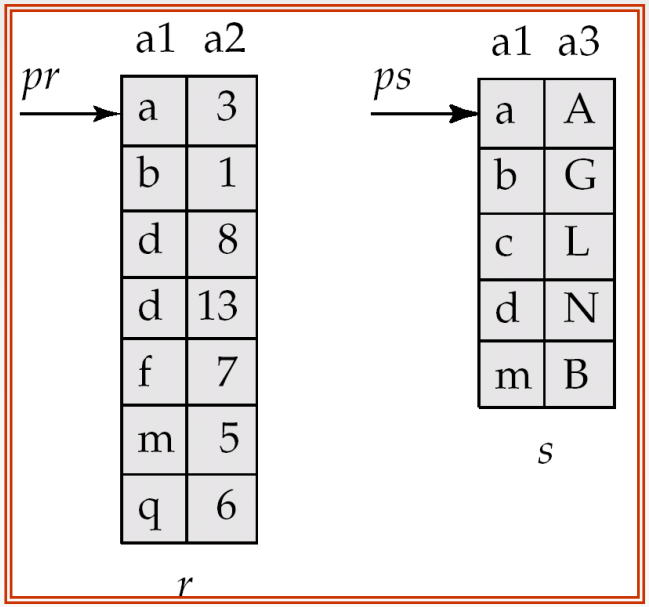
	Records	Blocks
customer = (*customer_name*, customer_street, customer_city, …)	10.000	400
depositor = (*customer_name*, *account_number*)	5000	100

- Compute *depositor* ⋈ *customer,* with *depositor* as the outer relation.
- Let *customer* have a primary B$^+$-tree index on the join attribute *customer-name,* which contains 20 entries in each index node.
- Since *customer* has 10,000 tuples, the height of the tree is $\lceil \log_{20} 10{,}000 \rceil = 4$, and one more access is needed to find the actual data
- *depositor* has 5000 tuples
- Cost of indexed nested loops join
  - 100 + 5000 * 5 = 25,100 block transfers



Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
    1. Join step is similar to the merge stage of the sort-merge algorithm.
    2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
    3. Detailed algorithm in book



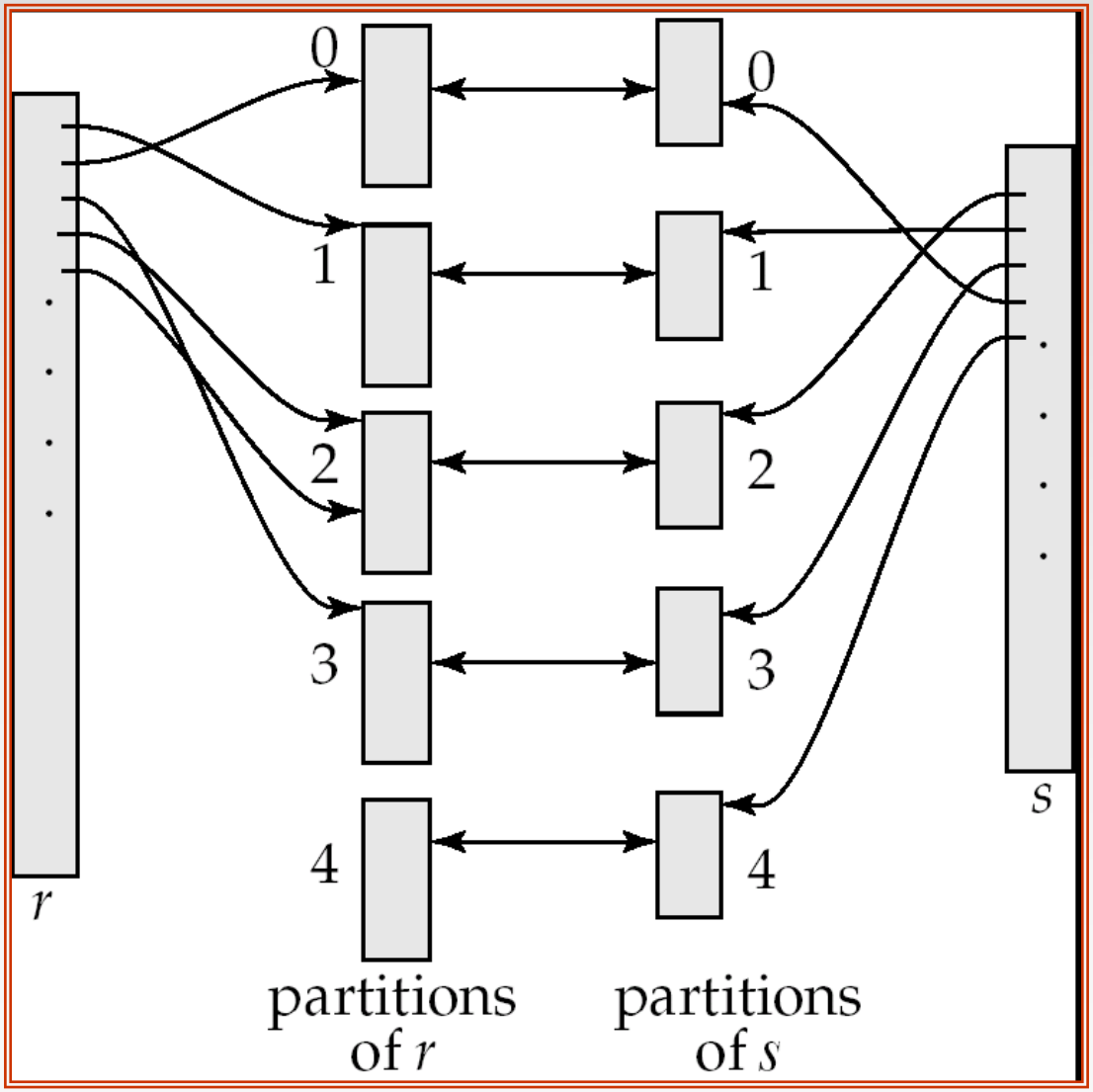
Merge-Join (Cont.)

- 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
  - + the cost of sorting if relations are unsorted.

Hash-Join

- Applicable only 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
    - 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
    - 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 the relation *s*, *h* is defined in such a way that partitions fit in memory
2. Partition *r* with the same hash function.
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 *h'* 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 relation** and *r* is called the **probe relation.**

The hash index is built in memory, there is no need to access the disk to retrieve the tuples.

Hash-Join algorithm (Cont.)

- The value *n* and the hash function *h* is chosen such that each s_i should fit in memory(build relation).
  - The probe relation partitions r_i (probe relation)need not fit in memory
  - Use the smaller relation as the build relation.

Cost of Hash-Join

- Partitioning of the relations requires complete reading and writing of r and s: $2(br + bs)$ block transfers
- Read and probe phases read each of the partitions once: $(br + bs)$ block transfers
- Number of blocks of the partition could be more than $(br + bs)$ as result of partially filled blocks that must be written and read back :
  2n block transfers for each relation
- Cost of hash join is
  $3(br + bs) + 4 * n$ block transfers
  n is usually quite small compared to $(br + bs)$ and can be ignored

Example of Cost of Hash-Join

customer ⋈ *depositor*

- Assume that memory size that can be used for a partition of the probe relation is 20 blocks
- $b_{customer}$ = 400 and $b_{depositor}$= 100.
- *depositor* is to be used as build relation. 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

Complex Joins

- Join with a conjunctive condition:

$$r \bowtie_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta i} s$
    - final result comprises those tuples in the intermediate result that satisfy the remaining conditions

$$\theta_1 \wedge \ldots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \ldots \wedge \theta_n$$

- Join with a disjunctive condition

$$r \bowtie_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n} s$$

  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta i} s$:

$$(r \bowtie_{\theta 1} s) \cup (r \bowtie_{\theta 2} s) \cup \ldots \cup (r \bowtie_{\theta n} s)$$

Other Operations

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
  - *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.

- **Projection:**
  - perform projection on each tuple
  - followed by duplicate elimination.

Other Operations : Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.

  E.g., select branch_name, sum (balance)

  from account

  group by branch_name

  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - *Optimization:* combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the aggregates
    - For avg, keep sum and count, and divide sum by count at the end

Other Operations : Set Operations

- **Set operations** ($\cup$, $\cap$ and —): we can implement these operations by first sorting both relations and then scanning once through each of the sorted relations.
- We can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function
  2. Process each partition *i* as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows
        - $r \cup s$:
          1. Add tuples in s_i to the hash index if they are not already in it.
          2. At end of s_i add the tuples in the hash index to the result.
        - $r \cap s$:
          1. output tuples in s_i to the result if they are already there in the hash index
        - $r - s$:
          1. for each tuple in s_i, if it is there in the hash index, delete it from the index.
          2. At end of s_i add remaining tuples in the hash index to the result.

Evaluation of Expressions

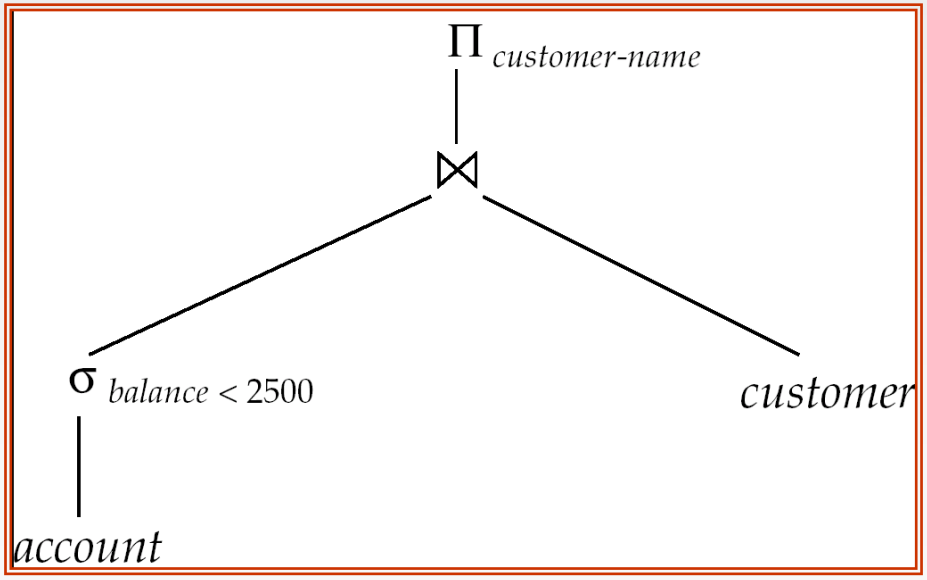
- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization**: generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining**: pass on tuples to parent operations even as an operation is being executed

Materialization

- **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below, compute and store

$$\sigma_{balance<2500}(account)$$

then compute the store its join with *customer,* and finally compute the projections on *customer-name.*



Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk

Pipelining

- **Pipelined evaluation** : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{balance<2500}(account)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.

- Pipelining may not always be possible – e.g., sort, hash-join.
  The *inputs to the operations are not available all at once for processing.*
  Each operation at the bottom of a pipeline continually generates output tuples and put them in its output buffer, until the buffer is full. When an operation uses tuples from its input buffer, removes tuples from the buffer.

Statistics for Cost Estimation

Statistical Information for Cost Estimation

The DBMS catalog stores the following statistical information.
For each relation r,

- n_r: number of tuples in a relation *r.*

  b_r: number of blocks containing tuples of *r.*

  l_r: size of a tuple of *r.*

  f_r: the number of tuples of *r* that fit into one block (blocking factor of *r*) with l_b size of a block

$$f_r = \left\lfloor \frac{l_b}{l_r} \right\rfloor$$

  if tuples of *r* are stored together physically in a file, then

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

- *V(A, r):* number of distinct values that appear in *r* for attribute *A.*
  This value is the same as the size of $\prod_A(r)$.
  If A is a key for relation r, $V(A, r) = n_r$

- min (A,r) and max(A,r)

Selection Size Estimation

The size estimate of the result of a selection operation depends on the selection predicate.

Let c denote the estimated number of tuples satisfying the condition.

- $\sigma_{A=V}(r)$ equality predicate
  - $c = n_r / V(A,r)$
  - $c = 1$ if equality condition on a key attribute

- $\sigma_{A \leq V}(r)$ (case of $\sigma_{A \geq V}(r)$ is symmetric)
  - If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r)
    - $c = n_r$ if v >= max(A,r)
    - $c = n_r . \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - In absence of statistical information c is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

Let s_i the size of $\sigma_{\theta i}(r)$, the probability that a tuple in the relation r satisfies θ_i *is* given by s_i / n_r

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *Assuming indepedence,* estimate of tuples in the result is:

$$n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$$

- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples:

$n_r - size(\sigma_\theta(r))$

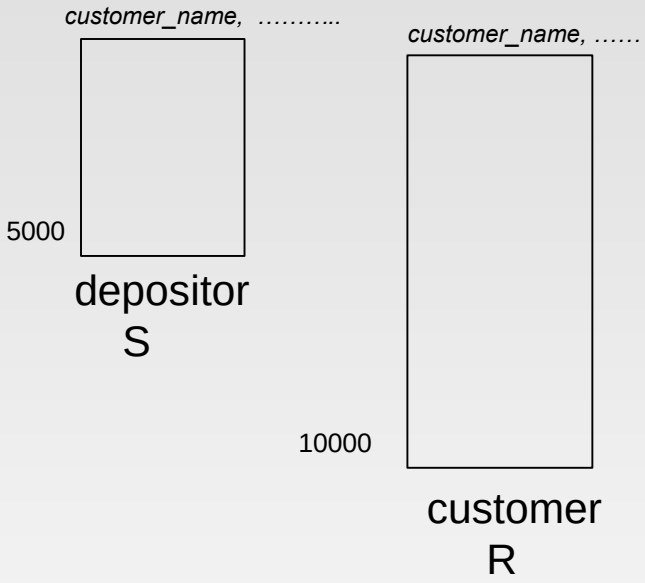
Join Operation: Running Example

depositor = (*customer_name*, *account_number, ….., A)*
customer= (*customer_name*, *customer_street, cusytomer_city, ……., A)*

Running example: *depositor* ⋈ *customer*

Catalog information for join examples:

- $n_{customer}$ = 10,000
  $f_{customer}$ = *25* implies that
  $b_{customer} = \lceil 10000/25 \rceil = 400$.
- $n_{depositor}$ = 5000.
  $f_{depositor}$ = 50 implies that
  $b_{depositor} = \lceil 5000/50 \rceil = 100$.



- *V(customer_name, customer) = 10000 (primary key!)*
- *V(customer_name, depositor)* = 2500, which implies that , on average, each customer in depositor has two accounts.
- Also assume that *customer_name* in *depositor* is a foreign key on *customer.*

Estimation of the Size of Joins

- The Cartesian product r x s contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes.

R set of attributes of r ; S set of attributes of s.

- If $R \cap S = \varnothing$, then $r \bowtie s$ is the same as r x s.

- If $R \cap S$ is a **key for *R***, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s:

    tuples in $r \bowtie s <= n_s$

- If $R \cap S$ is **a foreign key in *S* referencing *R***, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.

  tuples in $r \bowtie s = n_s$

In the example query *depositor* $\bowtie$ *customer, customer_name* in *depositor* is a foreign key of *customer*

- hence, the result has exactly $n_{depositor}$ tuples, which is 5000

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is **not a key for *R* or *S***.

  If we assume that every tuple *t* in *R* produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be:

$$n_r \cdot \frac{n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

$$n_s \cdot \frac{n_r}{V(A, r)}$$

The lower of these two estimates is probably the more accurate one

$$\min \left(n_r \cdot \frac{n_s}{V(A,s)} , n_s \cdot \frac{n_r}{V(A, r)} \right)$$

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for *depositor* ⋈ *customer* without using information about foreign keys:

  V(customer_name, depositor) = 2500, and
  V(customer_name, customer) = 10000

  - The two estimates are

    10000* 5000/2500 = 20,000

    and

    5000 * 10000/10000 = 5000

  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of

$$\Pi_A(r) = V(A,r)$$
$$\Pi_{AB}(r) = \min\{ V(A,r)*V(B,r), \; n_r \}$$

- Aggregation : estimated size of ${}_A\boldsymbol{g}_F(r) = V(A,r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(r)$
    - $\sigma_{\theta 1}(r) \cap \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \wedge \theta 2}(r)$
  - For operations on different relations:
    - estimated size of $r \cup s$ = size of r + size of s.
    - estimated size of $r \cap s$ = minimum size of r and size of s.
    - estimated size of $r - s$ = r.
    - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.

Estimation of Number of Distinct Values of an attribute: V(A, r')

Selections: $\sigma_\theta(r)$

- If θ forces A to take a specified value: $V(A,\sigma_\theta(r)) = 1$.
  - e.g., $A = 3$
- If θ forces A to take on one of a specified set of values:
  $V(A,\sigma_\theta(r))$ = number of specified values.
  - (e.g., $(A = 1 \ V \ A = 3 \ V \ A = 4)$),

- If the selection condition θ is of the form $A \ op \ r$
  $V(A,\sigma_\theta(r)) = V(A, r) * s$

  where s is the selectivity factor of the selection

  if $\sigma_{A \le V}(r)$ then $s = \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$

- In all the other cases: use approximate estimate of
  $\min(V(A,r), n_{\sigma\theta(r)})$

More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

Joins: $r \bowtie s$

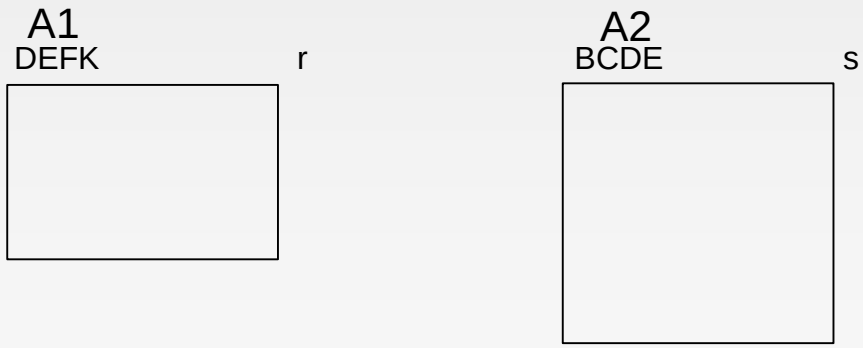
- If all attributes in A are from r
  estimated $V(A, r \bowtie s) = \min(V(A,r), n_{r \bowtie s})$
- If A contains attributes $A1$ from r and $A2$ from s, then estimated $V(A,r \bowtie s) =$

  $$\min(V(A1,r)*V(A2 - A1,s), V(A1 - A2,r)*V(A2,s), n_{r \bowtie s})$$

  - More accurate estimate can be got using probability theory, but this one works fine generally

$A = A1 \cup A2$

$A1 \cap A2 \neq \varnothing$,



Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in $\Pi_{A\,(r)}$ as in *r*.

- The same holds for grouping attributes of aggregation.

Choice of evaluation plans

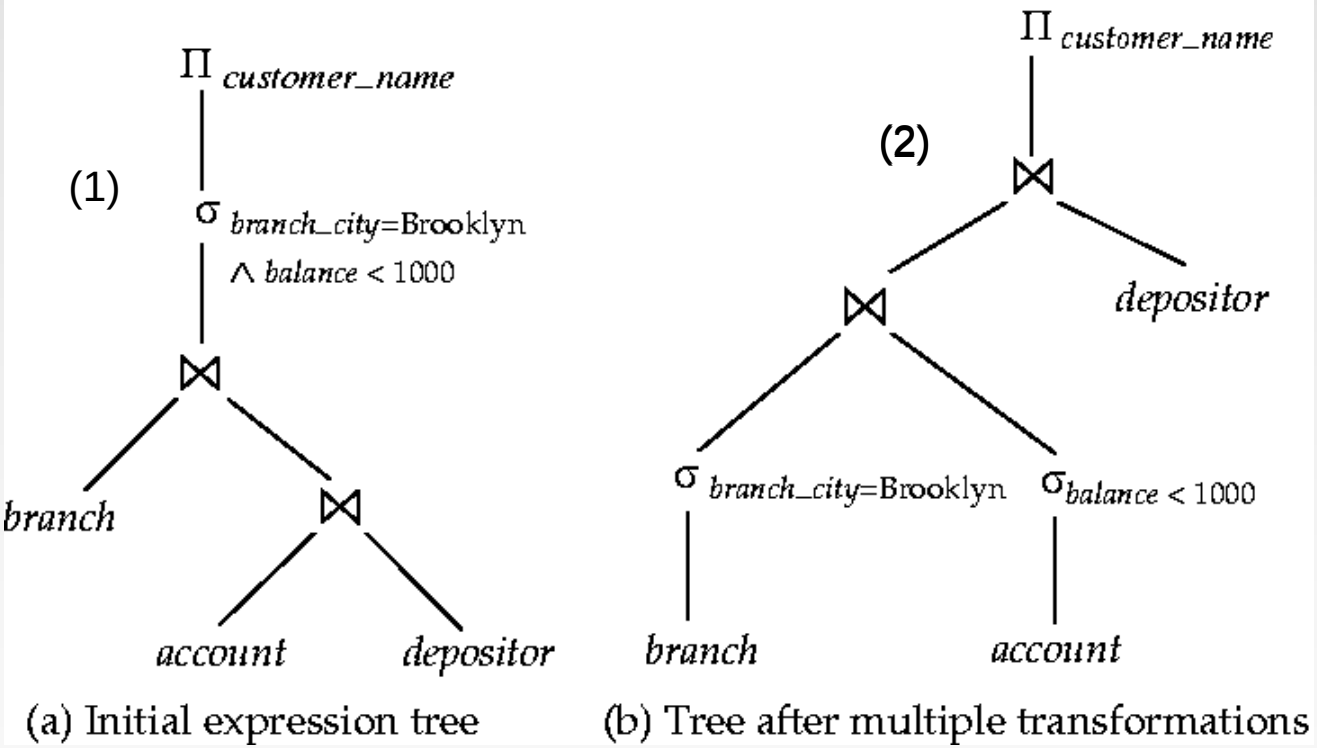
Evaluation plan

branch = (*branch_name*, *branch_city*, *assets*)

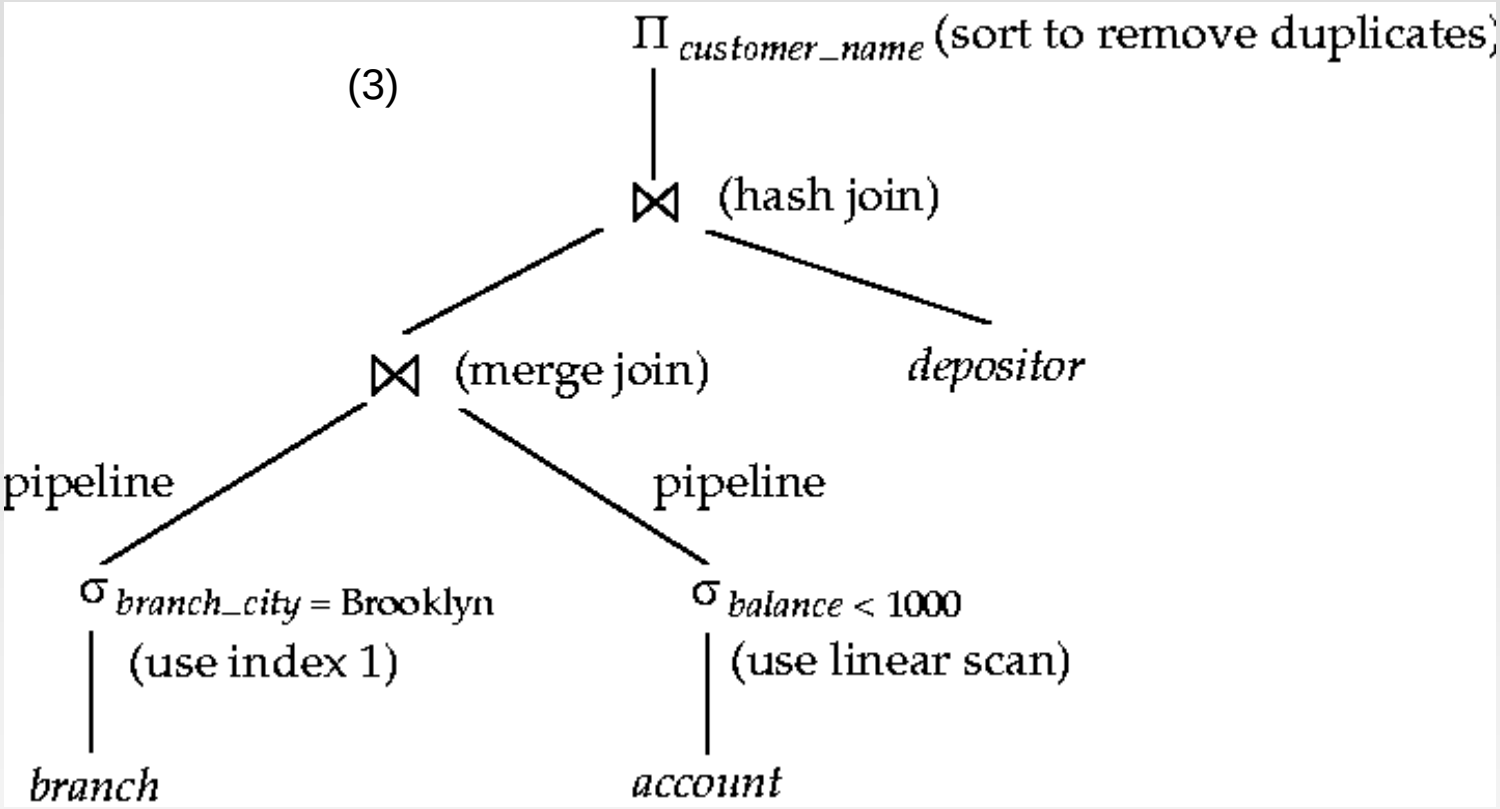
account = (*account_number*, *branch_name*, *balance*)

depositor = (*customer_id*, *customer_name*, *account_number)*

$\Pi_{customer\_name}(\sigma_{branch\_city=Brooklyn\ and\ balace\ <\ 1000}\ (branch \bowtie (account \bowtie depositor))))$



Evaluation plan



Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given query expression
- Can generate all equivalent expressions as follows:
  - Repeat
    - apply all applicable equivalence rules on every equivalent expression found so far

      "If one sub-expression satisfies one side of an equiv. rule, this sub-expression is substituted by the other side of the rule"

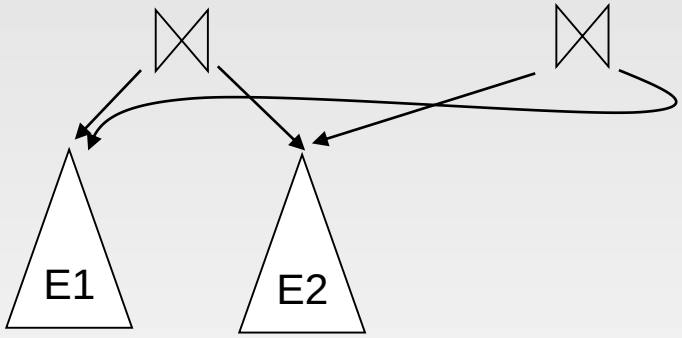
    - add newly generated expressions to the set of equivalent expressions

    Until no more new expressions ca be generated

- The above approach is very expensive in space and time

Implementing Transformation Based Optimization

- **Space requirements** reduced by sharing common sub-expressions
  - E.g. when applying join commutativity, subtrees below are the same and can be shared using pointers



- **Time requirements** reduced by not generating all expressions that can be generated with equivalence rules

OPTIMIZER: takes cost estimates of evaluation plans into account and avoids examining some of the expressions

Evaluation Plans

Point 1. choosing the cheapest algorithm for each operation independently may not yield best overall algorithm.
E.g.

- merge-join may be costlier than hash-join, but may provide a sorted output which reduce the cost for a later operation like elimination of duplicates.
- nested-loop join may provide opportunity for pipelining

TO CHOOSE THE BEST ALGORITHM, WE MUST CONSIDER EVEN NON OPTIMAL ALGORITHM FOR INDIVIDUAL OPERATIONS

Point 2. any ordering of operations that ensures that operation lower in the tree are executed before operation higher in the tree can be chosen

Different expressions for join

Different expressions for join

$r_1 \bowtie r_2 \bowtie r_3$

(r1,r2) r3	(r1,r3) r2	(r2,r3) r1	
(r2,r1) r3	(r3,r1) r2	(r3,r2) r1	12 different expressions
r3 (r1,r2)	r2 (r1,r3)	r1(r2,r3)	
r3 (r2,r1)	r2(r3,r1)	r1(r3,r2)	

Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.

There are $(2(n-1))! / (n-1)!$ different join orders for above expression.

n = 3, the number is (4!/2!) = 12
n = 7, the number is 665280
n = 10, the number is greater than 176 billion!

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

The result of the join between r1, r2 and r3, is joined to r4 and r5: **12 * 12 different** expressions BUT if we have found the best ordering between r1, r2 and r3, we may ignore all costlier orders of r1, r2 and r3: **12 + 12 different expressions**

Dynamic Programming in Optimization

Find best join tree for a set of *n* relations

Using dynamic programming, the least-cost join order for any subset of $S=\{R_1, R_2, \ldots R_n\}$ is computed only once and stored for future use.

- To find best plan for a set *S* of *n* relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of *S*.
- Recursively compute costs for joining subsets of *S* to find the cost of each plan.
- Choose the cheapest of the alternatives.
- Base case for recursion: single relation R_i access plan
  - Apply all **selections** on R_i using best choice of **indices** on R_i
- When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
  - Dynamic programming

Join Order Optimization Algorithm

```
procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
            return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
            set bestplan[S].plan and bestplan[S].cost based on the best way
            of accessing S  /* Using selections on S and indices on S */
    else for each non-empty subset S1 of S such that S1 ≠ S
            P1= findbestplan(S1)
            P2= findbestplan(S - S1)
            A = best algorithm for joining results of P1 and P2
            cost = P1.cost + P2.cost + cost of A
            if cost < bestplan[S].cost
                    bestplan[S].cost = cost
                    bestplan[S].plan = "execute P1.plan; execute P2.plan;
                                        join results of P1 and P2 using A"
    return bestplan[S]
```

Dynamic Programming in Optimization

$r_1 \bowtie r_2 \bowtie r_3$

BP(r1,r2,r3) = ∞

1) S1 = r1 S-S1 = r2, r3

BP(r1)

BP(r2,r3)

- BP(r2) BP(r3) Alg. Join

Alg Join

2) S1= r2 S-S1 = r1, r3

BP(r2)

BP(r1,r3)

- BP(r1) BP(r3) Alg. Join

Alg Join

3) S1= r3 S-S1 = r1, r2

BP(r3)

BP(r1,r2)

- BP(r1) BP(r2) Alg. Join

Alg Join

With dynamic programming time complexity of optimization of join is $O(3^n)$.

With n = 10, this number is 59000 instead of 176 billion!

4) S1 = r1,r2 S-S1 = r3

BP(r1,r2)

BP(r3)

- BP(r2) BP(r3) Alg. Join
-

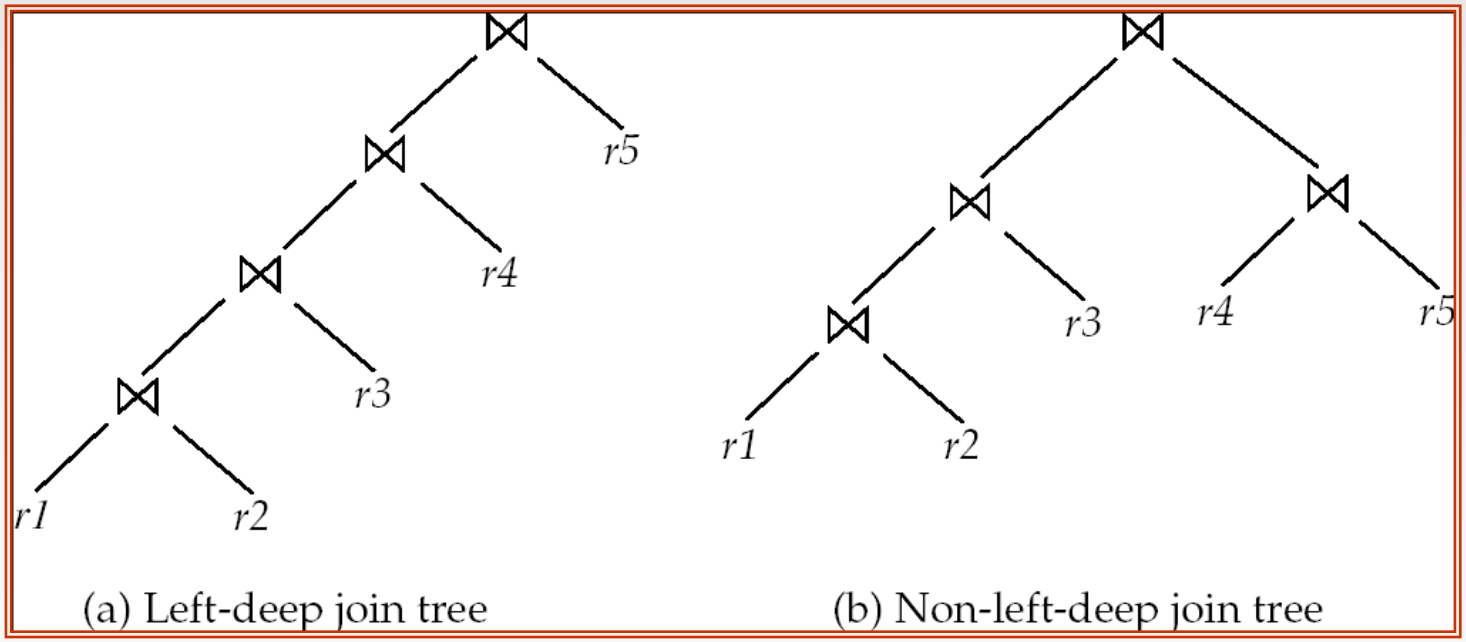
Alg Join

……..

Left Deep Join Trees

- Some optimizers consider only those join orders where the right operand of the join is an initial relation (**left-deep join trees)**, not the result of an intermediate join. For example IBM SystemR

- Only one input to each join is pipelined



Cost of Optimization

- To find best left-deep join tree for a set of *n* relations:
  - Consider *n* alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Modify optimization algorithm:
    - Replace "**for each** non-empty subset *S*1 of *S* such that *S*1 ≠ *S*"
    - By: **for each** relation r in S
      let S1 = S – r .
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Interesting Sort Orders

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$ (with A as common attribute)
- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation
  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join but generates result sorted on A
  - Which in turn may make merge-join with r_3 cheaper, which may reduce cost of join with r_3 and minimizing overall cost
  - Sort order may also be useful for *order by* and for *grouping*
- Not sufficient to find the best join order for each subset of the set of *n* given relations
  - must find the best join order for each subset, for each interesting sort order
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

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)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.

Futher reduce cost

- When examining a plan for an expression, if a part of the expression is costlier than the cheapest evaluation plan for full expression examined earlier, we can terminate

- No full expression containing such subexpression must be examined

STRATEGY:

- Heuristic guess of a good plan; evaluate the plan
- Only a few competitive plans will require full analysis

Structure of Query Optimizers

- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Left deep join starting from a different relation
  - Repeatedly pick “best” relation to join next
    - Pick best among the relations
- Intricacies of SQL complicate query optimization
  - E.g. nested subqueries

Query Optimizers (Cont.)

- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries
- OPTIMIZE THE QUERY ONCE AND STORE THE QUERY PLAN

Summary

- The process of finding a good strategy for processing a query is called **query optimization**
- There are a number of equivalence rules that can be used to transform an expression into an equivalent one (execution plan of a query).
- For evaluating cost of a query the system stores statistics for each relation (these statistics allow to estimate size and cost of intermediate results):
  - Number of tuples in the relation
  - Size of records in the relation
  - Number of distinct values that appear in the relation for a particular attribute
- Presence of indices has a signigficant influence on the choice of a query processing strategy
- Heuristics are used to reduce the number of plans considered. Heuristics include Push selection down and Push projection down rules

- **Query optimizer finds a “good” solution for processing the query**
- MySQL
  - update statistics
  - EXPLAIN **select** *attributes* **from** *tables* **where** *condition*;