Query Processing

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

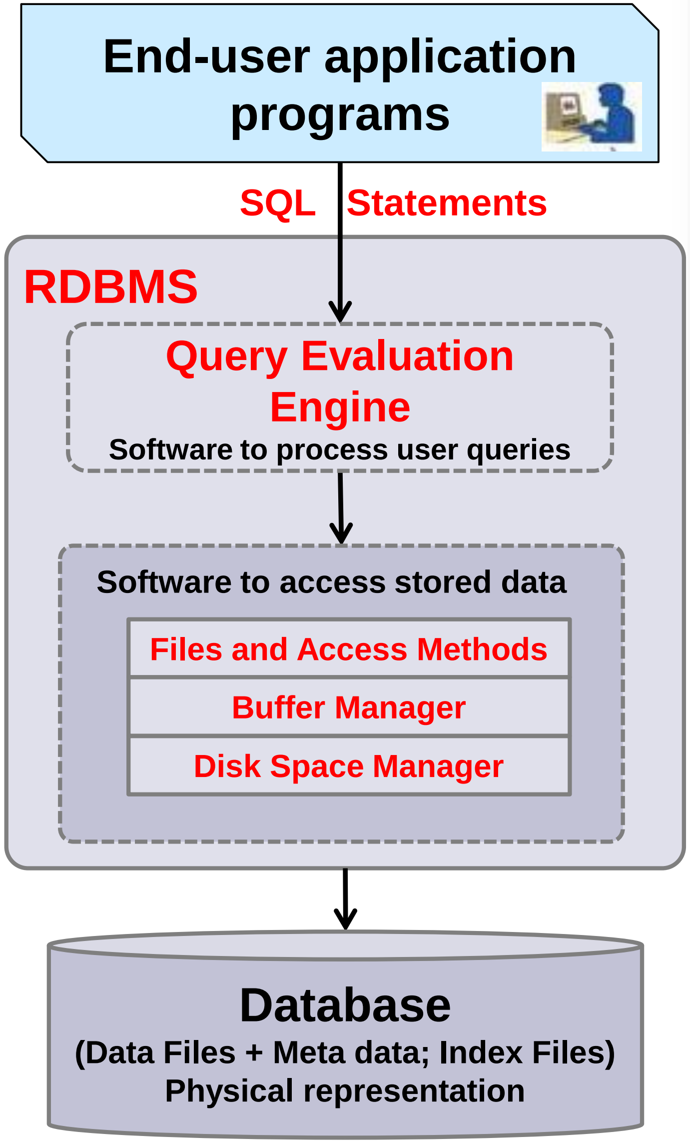
Introduction

- Users are expected to write "efficient" queries, but they don't always do that :
  - Users typically don't have enough information about the database to write efficient queries.

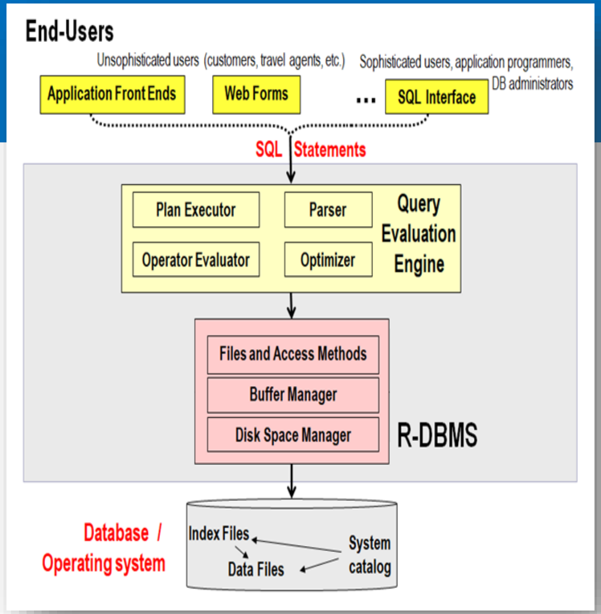
E.g.: no information on table size

- DBMS's job is to optimize the user's query by:
  - Converting the query to an internal representation (tree or graph)
  - Evaluate the costs of several possible ways of executing the query and find the best one.

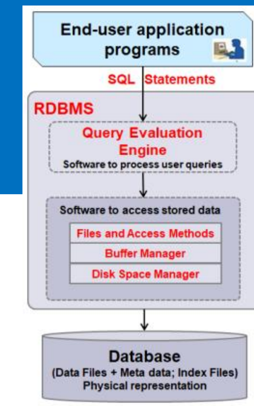
Simplified DBMS Structure



We are here!!!!



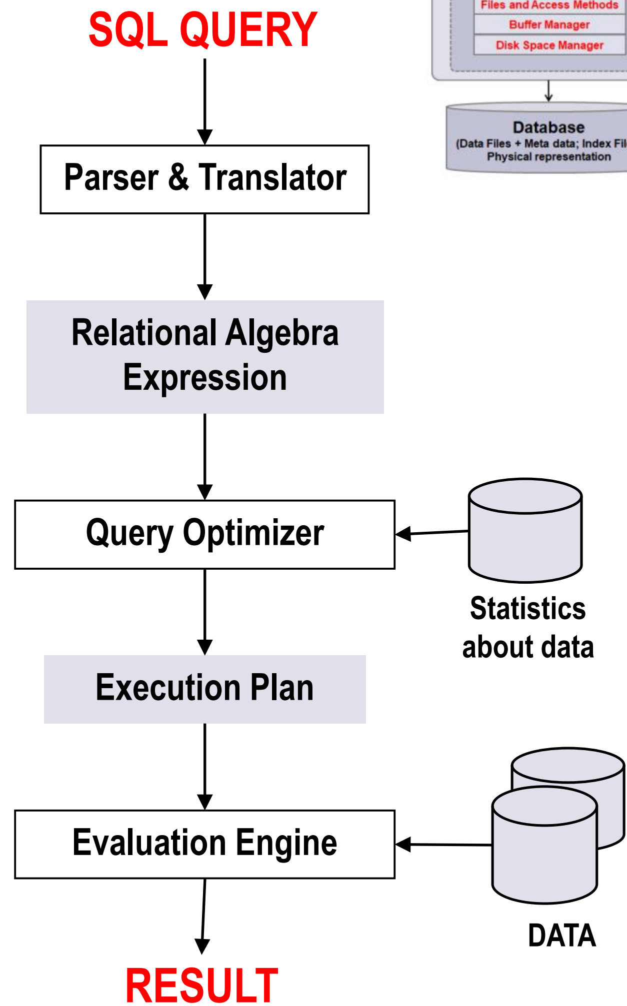
Basic Steps in Query Processing



1. Parsing and translation
   - translate the query into its internal form.
     This is then translated into relational algebra.
     - ☞ Query Tree, Graph
   - Parser checks syntax, verifies relations

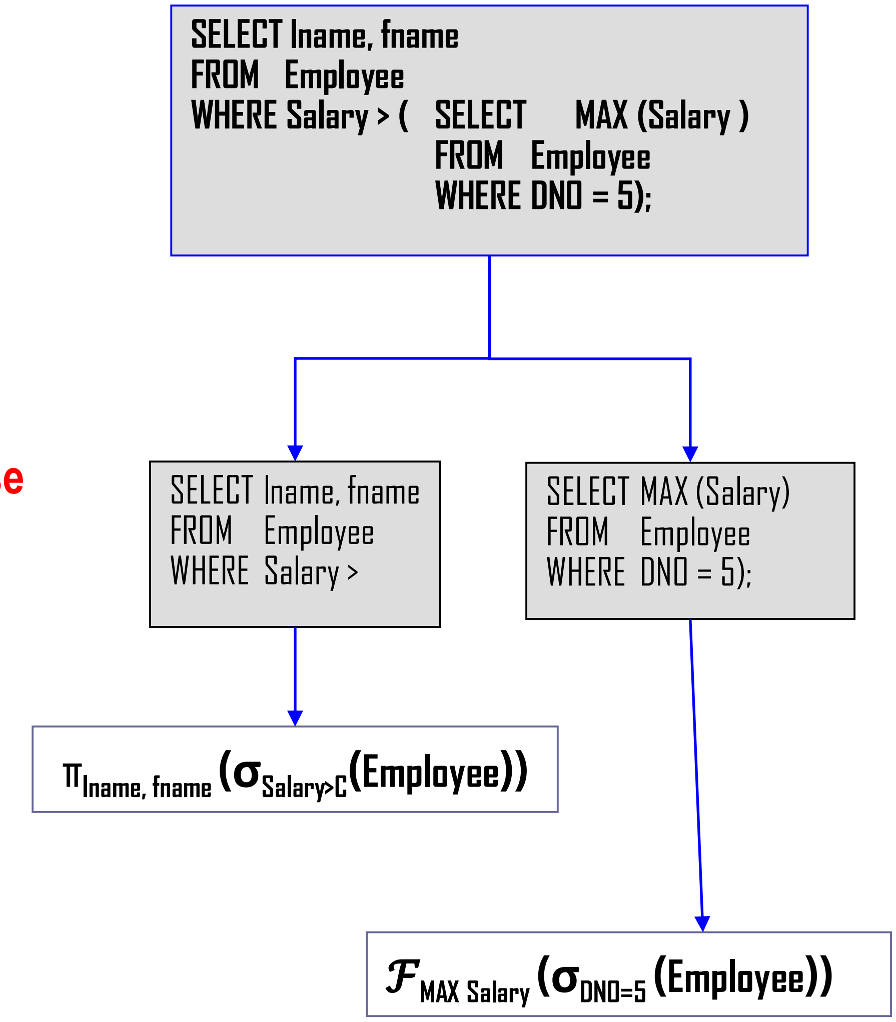
2. Query Optimization: The process of choosing a suitable execution strategy for processing a query.

3. Evaluation
   - The query-execution engine takes a query-evaluation plan, executes that plan,
   - and returns the answers to the query.



Translating SQL Queries into Relational Algebra

- **Query block:** The basic unit that can be translated into the algebraic operators and optimized.
- A query block contains:
  - a single SELECT-FROM-WHERE expression,
  - as well as GROUP BY and HAVING clause if these are part of the block.
- **Nested queries** within a query are identified as separate query blocks.
- **Aggregate operators** in SQL must be included in the extended algebra.



Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions

E.g. $\sigma_{balance<2500}(\Pi_{balance}(account))$ and $\Pi_{balance}(\sigma_{balance<2500}(account))$ are equivalent

- 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 $\geq$ 2500

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, etc.

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

disk access:

Typically 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 *number of block transfers from disk as the cost measure*
  - We ignore the difference in cost between sequential and random I/O for simplicity
  - We also ignore CPU costs for simplicity

- Costs depends on the **size** of the buffer in main memory
  - Having more memory reduces need for disk access
  - Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual execution
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

- Real systems take CPU cost into account, differentiate between sequential and random I/O, and take buffer size into account

- ☺ We do not include cost to writing output to disk in our cost formulae

Selection Operation

File scan: search algorithms that locate and retrieve records that fulfill a selection condition.

$\sigma_{balance<2500}$

- **linear search:**
  - Scan each file block and test all records to see whether they satisfy the selection condition.
  - Linear search can be applied regardless of selection condition or ordering of records in the file, or availability of indices.

- **binary search:**

  Applicable if selection is an equality comparison on the attribute on which file is ordered.

Selections Using Indices

Index scan: search algorithms that use an index

selection condition must be on search-key of index.

- Primary index on candidate key, equality - Retrieve a single record that satisfies the corresponding equality condition
- Primary index on nonkey, equality - Retrieve multiple records (Records will be on consecutive blocks)
- Equality on search-key of secondary index)
  - Retrieve a single record if the search-key is a candidate key
  - Retrieve multiple records if search-key is not a candidate key

Selections Involving Comparisons

Can implement selections of the form "$\sigma_{A \le V}(r)$ 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 $\sigma_{A \ge V}(r)$ <u>use index</u> to find first tuple $\ge v$ and scan relation sequentially from there
    - For $\sigma_{A \le V}(r)$ <u>just scan</u> relation sequentially till first tuple > *v;* do not use index
  - A7 (*secondary index, comparison*).
    - For $\sigma_{A \ge V}(r)$ <u>use index</u> to find first index entry $\ge v$ and scan index sequentially from there, to find pointers to records.
    - For $\sigma_{A \le V}(r)$ <u>just scan</u> 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 if many records are to be fetched!

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 a specific algorithms 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).**
  - Requires indices with record pointers.
  - 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

Join Operation

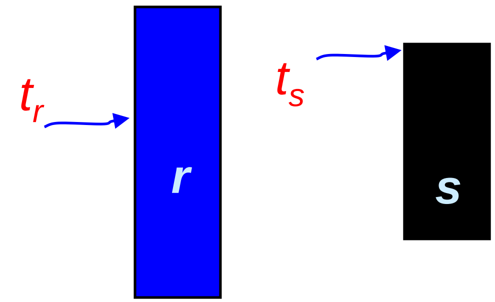
- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate

Examples use the following information

	Customer	*Depositor*
Number of records	10,000	5000
Number of blocks	400	100

Join Operation: Nested-Loop Join

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



outer relation

inner relation

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 $t_r \cdot t_s$ 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.

Join Operation: 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

$$n_r * b_s + b_r$$ disk accesses.

- If the smaller relation fits entirely in memory, **use that as the inner relation**.

Reduces cost to $b_r + b_s$ disk accesses.

Customer	**Depositor**
n=10,000	n=5000
b=400	b=100

- Assuming worst case memory availability cost estimate is
  - $5000 * 400 + 100 = \boxed{2{,}000{,}100}$ disk accesses with *depositor* as outer relation, and
  - $10000 * 100 + 400 = \boxed{1{,}000{,}400}$ disk accesses with *customer* as the outer relation.

If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 disk accesses.

- Block nested-loops algorithm (next slide) is preferable.

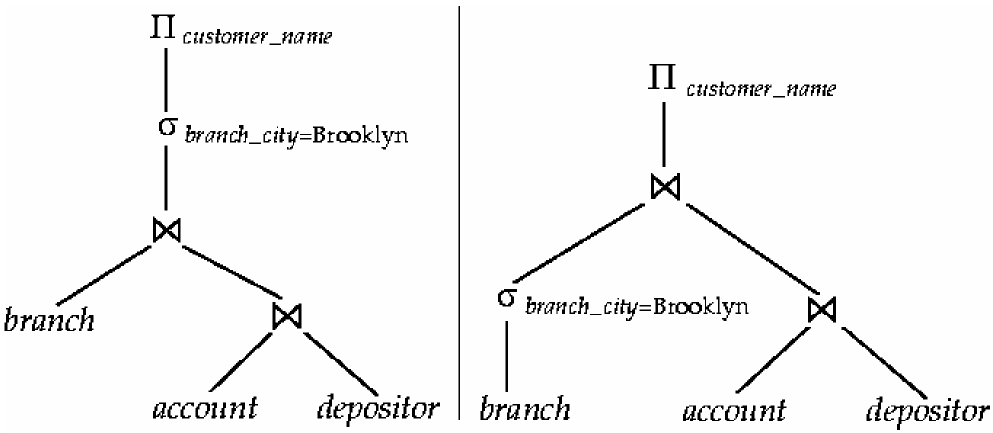
Query Optimization Using Heuristics

Query Optimization - using Heuristics

An overview

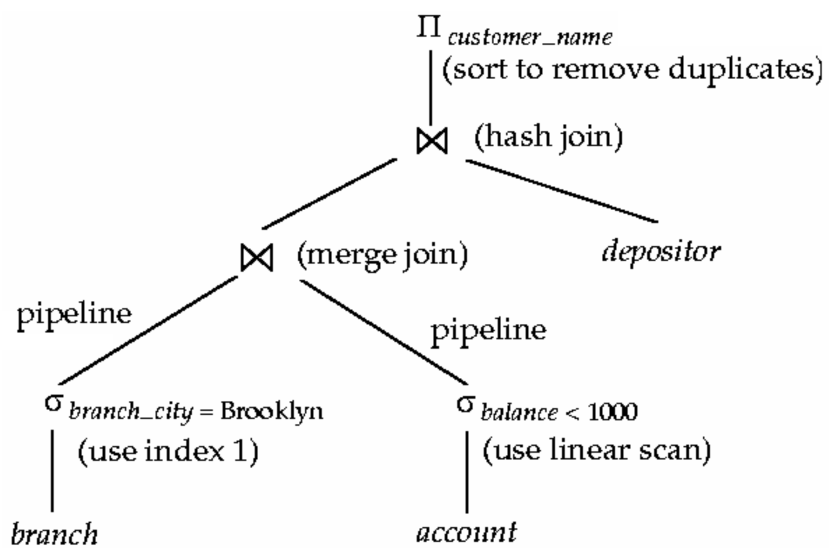
Alternative ways of evaluating a given query

- **Equivalent expressions**
- **Different algorithms for each operation**



An evaluation plan defines exactly:

- **what algorithm is used for each operation,**
- **and how the execution of the operations is coordinated.**



Steps in cost-based query optimization

- Generate logically equivalent expressions using *equivalence rules*
- Choose the cheapest plan based on estimated cost

Estimation of plan cost based on:

- Statistical information about relations - number of tuples, number of distinct values for an attribute
- Statistics estimation for intermediate results

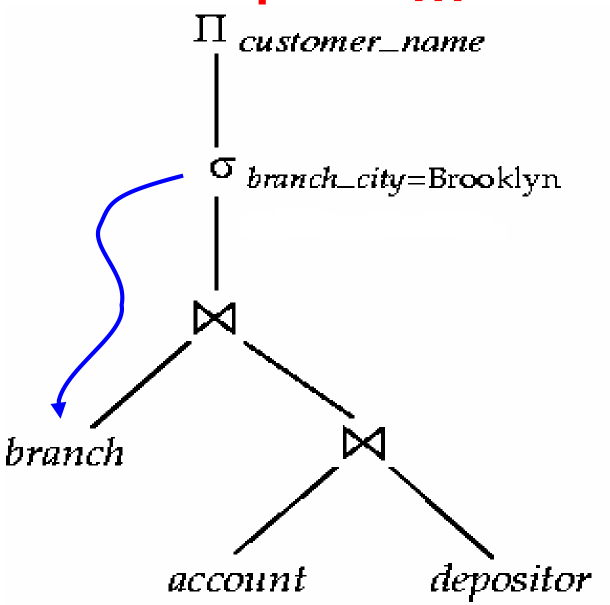
Example- Pushing Selections

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

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

branch (branch-name, branch-city, assets)
account (account-number, #branch-name, balance)
depositor (#customer-name, #account-number)

Transformation using rule 7a $(\sigma_{\theta 0}(E_1 *_\theta E_2) = (\sigma_{\theta 0}(E_1)) *_\theta E_2)$.



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

☞ **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.

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

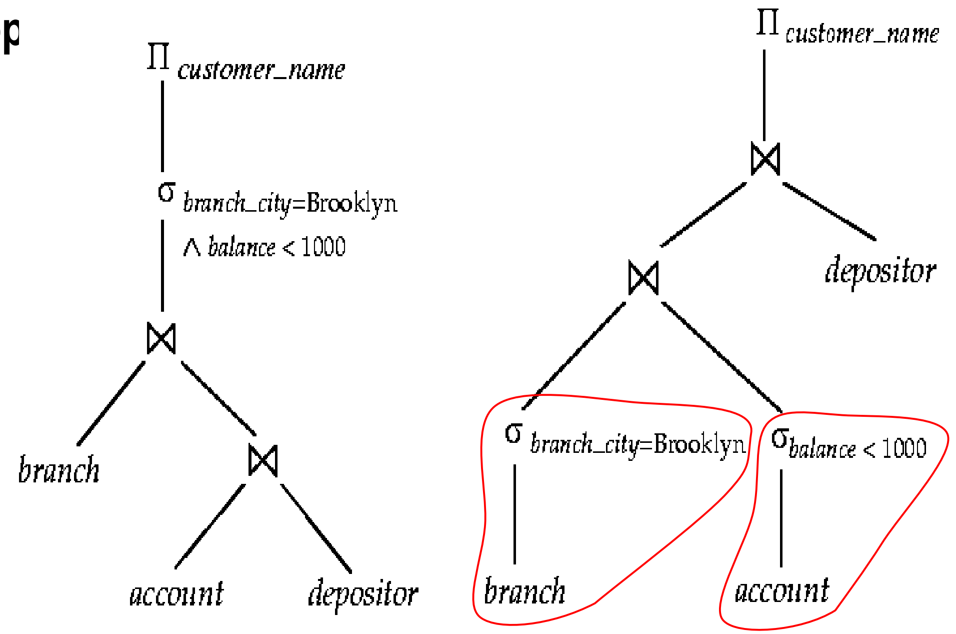
Transformation using join associatively (Rule 6a):

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

Second form provides an opportunity to app the “perform selections early” rule, resulting in the subexpression

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

$$* \ \sigma_{balance > 1000}(account)$$



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

Query Optimization
Join Ordering

For all relations r_1, r_2, and r_3,

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

If $r_2 \infty r_3$ is quite large and $r_1 \infty r_2$ is small, we choose $(r_1 \infty r_2) \infty r_3$

branch (branch-name, branch-city, assets)
account (account-number, #branch-name, balance)
depositor (#customer-name, #account-number)

Example:

Consider the expression

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

- Could compute (account ∞ depositor) first, and join result with ($\sigma_{branch\_city = \text{"Brooklyn"}}$(branch)
- but (account ∞ 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) ∞ account " first.

Query Optimization-

Query Execution Plans

An execution plan for a relational algebra query consists

of a combination of the relational algebra query tree and information about the access methods to be used for each relation as well as the methods to be used in computing the relational operators stored in the tree.

- **Materialized evaluation:** the result of an operation is stored as a temporary relation.

- **Pipelined evaluation:** as the result of an operator is produced, it is forwarded to the next operator in sequence.

22

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm.
  - Practical query optimizers incorporate elements of the following two broad approaches:
    1. Search all the plans and choose the best plan in a cost-based fashion.
    2. Uses heuristics to choose a plan.

- Cost-Based Optimization
  - Consider finding the best join-order for $r_1 \infty r_2 \infty \ldots \infty r_n$.
  - There are $(2(n - 1))!/(n - 1)!$ different join orders for above expression.

    With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!

  - ☞ No need to generate all the join orders.
  - ☞ Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

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
- Intricacies of SQL complicate query optimization. E.g. nested subqueries

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
- 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

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:

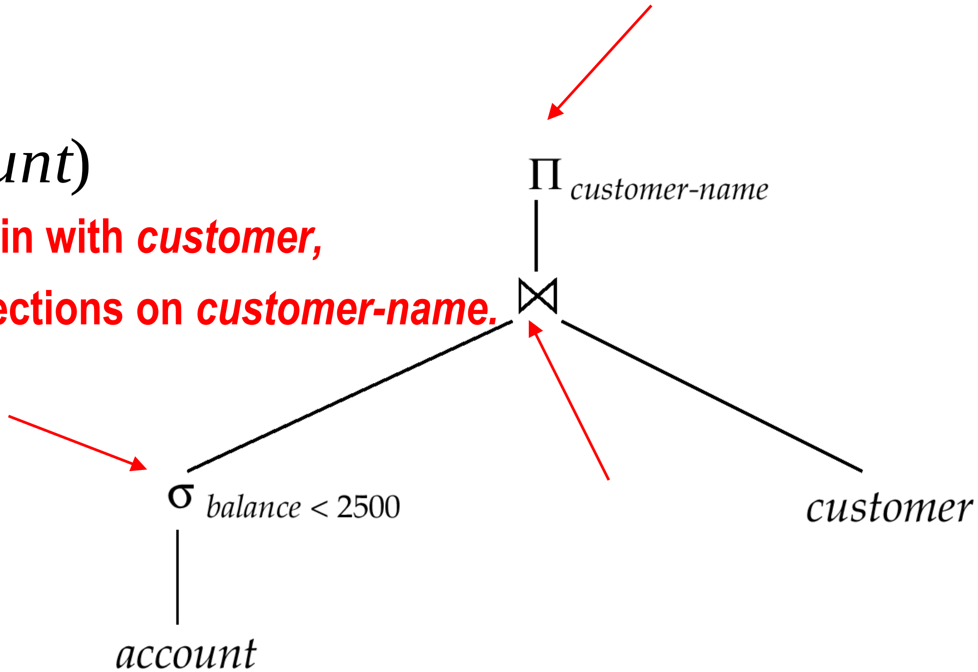
evaluate one operation at a time,

starting at the lowest-level.

Use intermediate results materialized into temporary relations to evaluate next-level operations.

E.g., in figure below,

1. compute and store
   $$\sigma_{balance<2500}(account)$$
2. then compute the store its join with *customer,*
3. 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
- Double buffering: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

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.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: demand driven and producer driven