HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

IT3090E - Databases

Chapter 9: Query processing and optimization

Muriel VISANI
murielv@soict.hust.edu.vn

Outline

- Overview
  - What is query processing
  - Phases of query processing
  - Parser
  - Optimizer
- Understanding query optimizer
  - Step 1: Equivalence transformation
  - Step 2: Annotation for the algorithm of the Relational Algebra expression
  - Step 3: Cost estimation for different query execution plans

Objectives

- Upon completion of this lesson, students will be able to:
  - Understand different phases of query processing
  - Understand the implementation of query optimizer

Keywords

Query processing	Activities involved in retrieving/storing data from/to the database
Query optimization	Selection of an efficient query execution plan
Relational algebra	An algebra whose operands are relations or variables that represent Relations

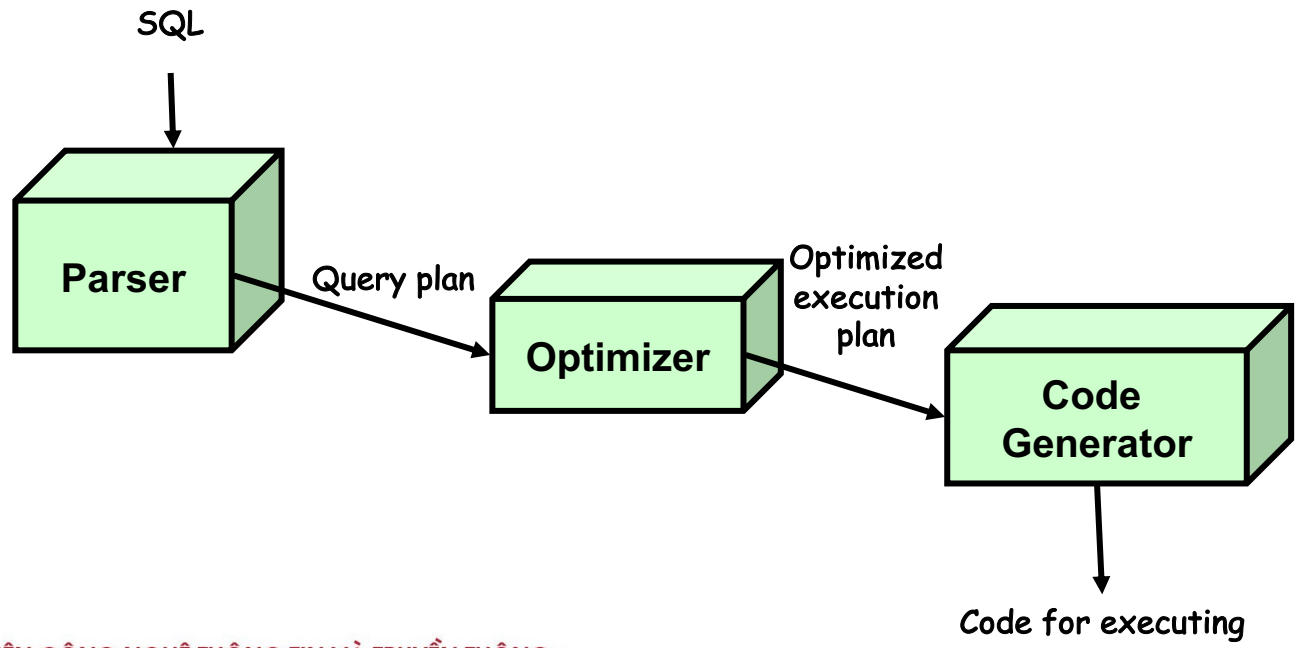
1. Overview

- What is query processing
- Phases of query processing
- Parser
- Optimizer

1.1. What is **query processing**?

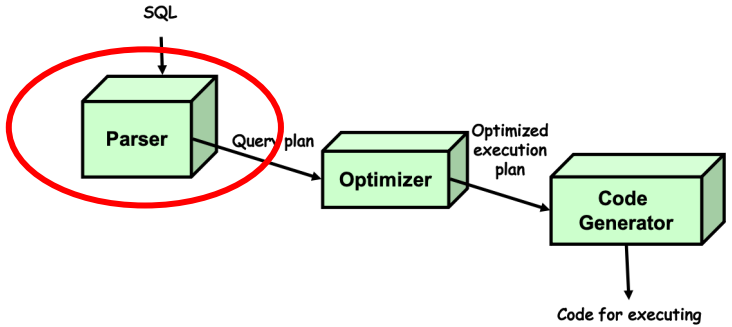
- The entire process or activities involved in retrieving data from the database
  - SQL **query translation** into low level instructions (usually relational algebra)
  - **Query optimization** to save resources, cost estimation or evaluation of query
  - **Query execution** for the extraction of data from the database.

1.2. Phases of query processing

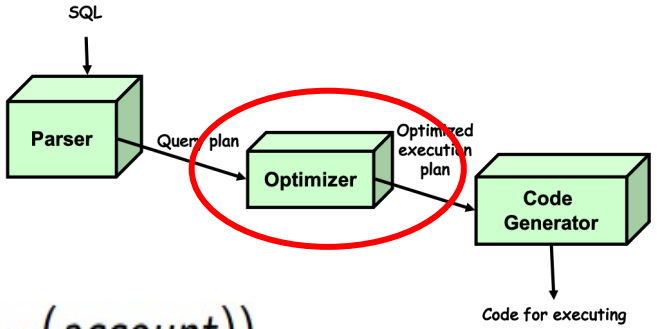


1.3. Parser

- Scans and parses the query into individual tokens and verifies the **correctness of the query**
  - Does it contain the right keywords?
  - Does it conform to the syntax?
  - Does it containt the valid tables, attributes?
- **Output: Query planS**
  - E.g.
    - Input: SELECT balance FROM account WHERE balance < 2500
    - Output: Relational Algebra (RA) expression $\sigma_{balance<2500}(\Pi_{balance}(account))$
    - But it's not unique (multiple possible RA expressions for 1 SQL statement)



1.4. Optimizer



- **Input:** RA expression(s)
- **Output:** Query **execution** plan $\Pi_{balance}(\sigma_{balance<2500}(account))$

- **Query execution plan** = query plan + the algorithms for the executions of RA operations
  - Aim: choose the cheapest execution plan out of all the possible execution plans
  - Step 1: Equivalence transformation
  - Step 2: Query execution plan: annotation on a query plan (RA expression) for detailing the algorithm to implement the query plan
    - Annotated RA expression specifying detailed evaluation strategy is called the execution plan (includes, e.g., whether index is used, join algorithms, . . .)
  - Step 3: Cost estimation for different query execution plans

$\Pi_{balance}$

|

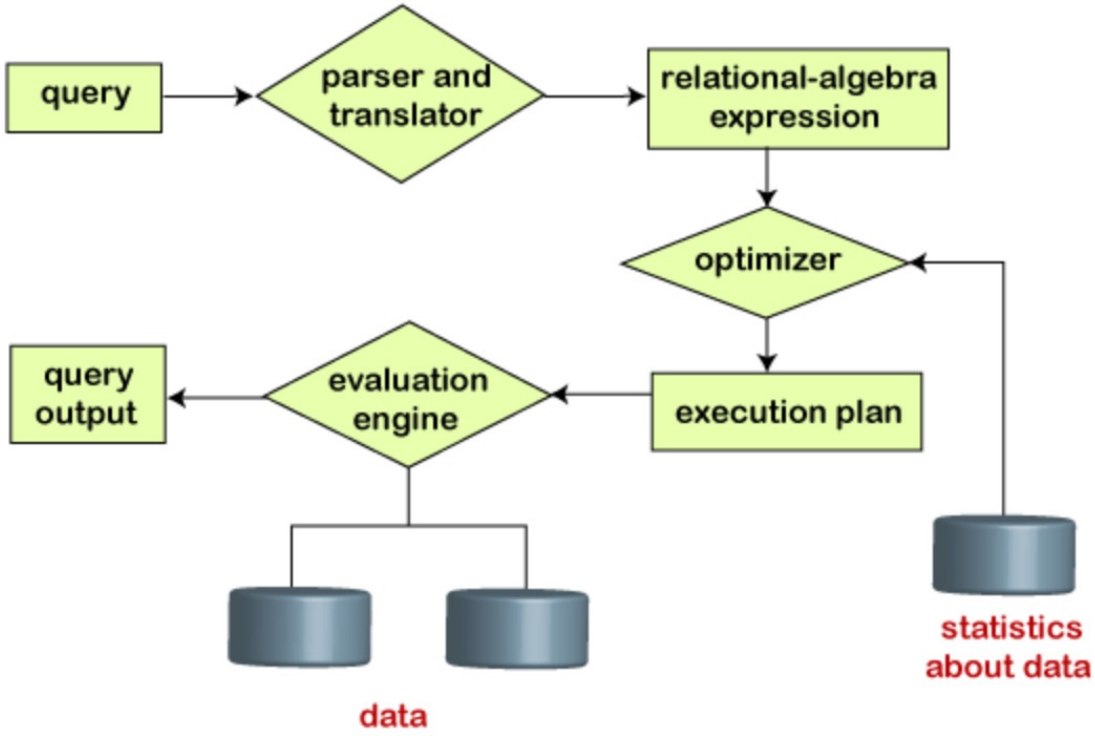
$\sigma_{balance<2500}$

use index 1

|

account

1.4. Parser *vs.* optimizer



Steps in query processing

2. Understanding optimizer

- Choose the cheapest execution plan out of the possible ones
  - Step 1: Equivalence transformation of the RA expression
  - Step 2: Annotations for the algorithmic execution of the RA expression(s)
  - Step 3: Cost estimation for different query execution plans

2.1. Step 1: Equivalence transformation

- RA expressions are **equivalent** if they generate the same set of tuples ***on every database instance***
- **Equivalence rules:**
  - Transform one relational algebra expression into equivalent one
  - Similar to numeric algebra: a + b = b + a, a(b + c) = ab + ac, etc
- **Why producing equivalent expressions?**
  - equivalent algebraic expressions give the same result
  - but usually the execution time varies significantly

2.1. Step 1: Equivalence transformation

- Recall about Relational Agebra (RA) notations

The operations have their own symbols.

Operation	Symbol
Union	$\cup$
Intersection	$\cap$
Set difference	-

Operation	Symbol
Projection	π
Selection	σ
Cartesian product	$\times$
Join	$\bowtie$
Left outer join	$\overset{\leftarrow}{\bowtie}$
Right outer join	$\overset{\rightarrow}{\bowtie}$
Full outer join	$\overset{\leftrightarrow}{\bowtie}$

2.1. Step 1: Equivalence transformation

- Recall about joins
- There are 3 kinds of inner joins:
  - **THETA JOIN** allows you to merge two tables based on the condition represented by θ. Theta joins work **for all comparison operators:** $A \bowtie_\theta B$
    - **Example:** $A \bowtie_{A.column\ 2 > B.column\ 2} (B)$
  - **EQUI JOIN** is a special kind of θ join where θ is an equivalence
    - **Example:** $A \bowtie_{A.column\ 2 = B.column\ 2} (B)$
  - **NATURAL JOIN** is an equi join performed on two (at least one) common attribute(s) between two relations (same name, same domain) $C \bowtie D$

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - To optimize a query, we must convert the query into its equivalent form, as long as an equivalence rule is satisfied
    - There are **12 equivalence rules** in total:

  - **(1)** Conjunctive selection operations can be deconstructed into a sequence of individual selections; cascade of σ
    - $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right)$
    - **Explanation**: Applying condition $\theta_1 \wedge \theta_2$ is expensive. Instead, filter out tuples satisfying condition θ_2 (inner selection) and then apply condition θ_1 (outer selection) to the resulting fewer tuples => fewer records to process for the second selection

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(2)** Selection operations are commutative
    - $\sigma_{\theta_1}\left(\sigma_{\theta_2}(E)\right) = \sigma_{\theta_2}\left(\sigma_{\theta_1}(E)\right)$
    - **Explanation:** selection is commutative in nature. This means, it gives the same result whether we apply σ_{θ_1} first or σ_{θ_2} first. In practice, it is in general better and more optimal to apply first the selection which yields a fewer number of tuples (most restricive condition first). This saves time on our outer selection.

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(3)** Only the final operations in a sequence of projection operations is needed; cascade of Π
    - $\Pi_{L_1}\left(\Pi_{L_2}\left(\ldots \Pi_{L_n}(E)\ \ldots\right)\right) = \Pi_{L_1}(E)$
    - **Explanation**: A cascade or a series of projections is meaningless, because in the end, we are only selecting those columns which are specified in the last, or the outermost projection (here Π_{L_1})

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(4)** Selections can be combined with Cartesian products and inner joins
    - $\sigma_{\theta} \quad (E_1 \times E_2) = E_1 \bowtie_{\theta} \quad E_2$
      - **Explanation**: The cross product operation very expensive, because it matches each tuple of E1 (total m tuples) with each tuple of E2 (total n tuples). This yields m*n entries. If we apply a selection operation after that, we would have to scan through m*n entries to find the suitable tuples which satisfy θ. Instead of doing this, it is more optimal to use the Theta Join, a join specifically designed to select only those entries in the cross product which satisfy the Theta condition, without evaluating the entire cross product first.
    - $\sigma_{\theta_1}\left(E_1 \bowtie_{\theta_2} E_2\right) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
      - **Explanation**: Theta Join radically decreases the number of resulting tuples, so if we apply $\theta_1 \wedge \theta_2$ into the Theta Join itself, we get fewer scans to do.

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(5)** Theta Join operations are commutative
    - $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
    - **Explanation:** Theta Joins are commutative, and the query processing time depends to some extent which table is used as the outer loop and which one is used as the inner loop during the join process (based on the indexing structures and blocks).

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(6)** Inner join operations are associative
    - **Natural join:** $E_1 \bowtie (E_2 \bowtie E_3) = (E_1 \bowtie E_2) \bowtie E_3$
      - **Explanation:** Joins are all commutative as well as associative, so one must join those two tables first which yield fewer resulting records, and then apply the other join.
    - **Theta joins** are associative in the following manner, where θ_2 involves attributes from E2 and E3 only
      - $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(7)** Selection distributes over joins in the following ways
    - If θ_1 involves attributes of E1 only:
      - $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = \sigma_{\theta_1}(E_1) \bowtie_{\theta_2} E_2$
      - This can be extended to two selection conditions as follows
    - If predicate θ_1 involves only attributes of E1 and θ_2 involves only attributes of E2
      - $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_3} E_2) = \sigma_{\theta_1}(E_1) \bowtie_{\theta_3} \sigma_{\theta_2}(E_2)$
      - This is a consequence of rules 1 and 7

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(8)** Projection distributes over join as follows
    - If θ involves attributes in $L_1 \cup L_2$ only and L_i contains attributes of E_i, then:
      - $\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1}(E_1) \bowtie_\theta \Pi_{L_2}(E_2)$
      - **Explanation:** in such a case, it is better to individually apply the projections on both the tables before joining. This leaves us with a fewer number of columns on either side, hence contributing to an easier join
    - Assume that E_1 and E_2 have sets of attributes L_1 and L_2. Assume that L_3 are attributes of the expression E_1, involved in the θ join condition but not in $L_1 \cup L_2$. Similarly, let L_4 be attributes of the expression E_2 involved only in the θ join condition and not in $L_1 \cup L_2$ attributes:
      - $\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}(\Pi_{L_1 \cup L_3}(E_1) \bowtie_\theta \Pi_{L_2 \cup L_4}(E_2))$

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(9)** The set operations union and intersection are commutative
    - $E_1 \cup E_2 = E_2 \cup E_1$
    - $E_1 \cap E_2 = E_2 \cap E_1$
  - **(10)** The union and intersection are associative
    - $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
    - $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$
    - **Tip:** Union and intersection are both associative, so we can enclose any tables in parentheses according to requirement and ease of access.

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(11)** The selection operation distributes over union, intersection, and set-difference
    - $\sigma_\theta(E_1 \cup E_2) = \sigma_\theta(E_1) \cup \sigma_\theta(E_2)$
    - $\sigma_\theta(E_1 \cap E_2) = \sigma_\theta(E_1) \cap \sigma_\theta(E_2)$
    - $\sigma_\theta(E_1 - E_2) = \sigma_\theta(E_1) - \sigma_\theta(E_2)$
    - **Explanation:** In set difference, we know that only those tuples are shown which belong to table E1 and do not belong to table E2. So, applying a selection condition on the entire set difference is equivalent to applying the selection condition on the individual tables and then applying set difference. This will reduce the number of comparisons in the set difference step (same idea for union and intersection)

2.1. Step 1: Equivalence transformation

- Equivalence tranformation rules
  - **(12)** The projection distributes over the union
    - $\Pi_L(E_1 \cup E_2) = \Pi_L(E_1) \cup \Pi_L(E_2)$
    - **Explanation**: Applying individual projections before computing the union of E1 and E2 is more optimal: lesss columns to process during the union.

2.1. Step 1: Equivalence transformation

- Exercise
  - For each of the following pairs of expressions, give instances of relations that show the expressions are **NOT** equivalent.

1. $\Pi_A(R - S)$ and $\Pi_A(R) - \Pi_A(S)$
2. (R ⟕ S) ⟕ T and R ⟕ (S ⟕ T) In other words, the natural left outer join is not associative.
3. $\sigma_\theta(E_1 \bowtie E_2)$ and $E_1 \bowtie \sigma_\theta(E_2)$, if θ involves only attributes from E2

2.1. Step 1: Equivalence transformation

- Exercise solution
  1.

2.1. Step 1: Equivalence transformation

- Exercise solution
  2.

2.1. Step 1: Equivalence transformation

- Exercise solution
  3.

2.2. Step 2: Execution algorithms of RA operations

- An algebra expression **is not a query execution plan** (it is only a query plan)
- **Additional decisions must be made by the optimizer:**
  - which indexes to use, for example, for joins and selects?
  - which algorithms to use for joins, for example, sort-merge vs. hash join?
  - materialize intermediate results or pipeline them?

- The aim of the **optimizer** is to translate SQL statements (SELECT, INSERT, UPDATE, or DELETE) into an efficient execution plan (access plan)
  - The execution plan is made up of various relational algebra operators (join, duplicate elimination, union, and so on).
  - The operators within the execution plan may not be structurally equivalent to the original SQL statement, but the access plan's various operators will compute a result that is semantically equivalent to that SQL request.

2.2. Step 2: Execution algorithms of RA operations

- Basic Operators
  - One-pass operators:
    - Scan
    - Select
    - Project
  - Multi-pass operators:
    - Join
      - Various implementations
      - Handling of larger-than-memory sources
    - Aggregation, union, etc.

2.2. Step 2: Execution algorithms of RA operations

- 1-Pass Operators: Scanning a Table
  - **FULL TABLE SCAN** (Sequential scan): read through blocks of table
    - Reads the entire table (all rows and columns) as stored on the disk. Although multi-block read operations improve the speed of a full table scan considerably, it is still one of the most expensive operations. Besides high IO rates, a full table scan must inspect all table rows so it can also consume a considerable amount of CPU time.
    - However, full table scanning isn't necessarily a bad thing: on small tables a full scan is pretty insignificant. But, it can be a good place to start looking, if a query is unacceptably long running, and this query involves a FTS on a very large table…
    - Full table accesses may still occur on a table with an index, even if the query uses an indexed column, simply because the query optimizer may deem if faster to blat the entire table data into memory than bother with the indirection of going to the index, looking for the relevant rows, then picking them off the disk…

2.2. Step 2: Execution algorithms of RA operations

- 1-Pass Operators: Scanning a Table
  - **TABLE ACCESS BY INDEX ROWID**
    - ROWID must be previously retrieved from an index lookup, using an INDEX SCAN
      - Index scan: retrieve tuples in index order (keywork ROWID)
      - Depending on the DBMSs: INDEX UNIQUE SCAN, INDEX RANGE SCAN, INDEX FULL SCAN, INDEX FAST FULL SCAN…
      - Example using ORACLE: https://use-the-index-luke.com/sql/explain-plan/oracle/operations

2.2. Step 2: Execution algorithms of RA operations

- Nested-loop INNER JOIN

```
For each tuple tr in r {
    for each tuple ts in s {
        if (tr and ts satisfy the join condition) {
            add tuple tr x ts to the result set
        }
    }
}
```

- No index needed
- Any join type (THETA, EQUI, NATURAL)
- Expensive: $O(n^2)$

2.2. Step 2: Execution algorithms of RA operations

- Example: nested-loop for **EQUI** JOIN: implementation
  - **buffer** = M blocks (M-1 for reading, 1 for writing)
    - More about **buffers:** https://en.wikipedia.org/wiki/Data_buffer
    - M-2 blocks for R1 records, 1 block for R2 records, 1 block for writing
  - For each M-2 blocks ∈ R1 *(outer loop)*
    - For each block ∈ R2 *(inner loop)*
      - output matching pairs
- Disk I/O:
  - Number of blocks to read? b(R1) + b(R1) * b(R2) / (M-2)
  - Number of blocks to write? # of blocks of the join results
  - Which file to use as outer loop file?

2.2. Step 2: Execution algorithms of RA operations

- Single-loop EQUI JOIN (Index-based)
  - Provided there exists an index on table S

```
For each r ∈ R1 do
    retrieve tuples from R2 using index search (R2.C = r.C)
```

    - Index needed
    - Cheaper: O(nlogm)
- Disk I/O:
  - Number of blocks to read? b(R1) + r(R1) * (Index search cost on R2)
  - Which file to use as outer loop file?

2.2. Step 2: Execution algorithms of RA operations

- Sort-merge JOIN
  - Requires data physically sorted by join attributes: Merge and join sorted files, reading sequentially a block at a time
  - **Conceptual algorithm:**
    - (1) if R1 and R2 are not sorted, then sort them
    - (2) i ← 1; j ← 1;
    - while (i ≤ r(R1)) && (j ≤ r(R2)) do
      - if R1{ i }.C == R2{ j }.C then output matched tuple
      - else if R1{ i }.C > R2{ j }.C then j ← j+1
      - else if R1{ i }.C < R2{ j }.C then i ← i+1
  - Idea:
    - Maintain two pointers: one on R1, one on R2
    - While tuple at R1 < tuple at R2, advance R1 (and vice versa)
    - While tuples match, output all possible pairings
  - Very efficient for presorted data. Otherwise, may require a sort (adds cost + delay)

2.2. Step 2: Execution algorithms of RA operations

- Sort-merge EQUI JOIN
  - Recall: buffer = M blocks (M-1 for reading, 1 for writing)
  - Disk I/O:
    - # of blocks to read if R1 and R2 are sorted: b(R1) + b(R2)
    - # blocks accesses to sort R1 and R2:

$$2b(R1)*\log_M b(R1) + 2b(R2)*\log_M b(R2)$$

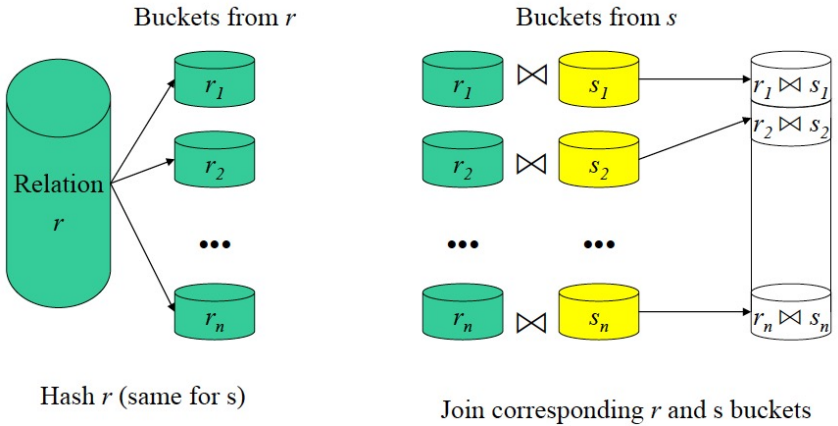
2.2. Step 2: Execution algorithms of RA operations

- **Merge join** is used when projections of the joined tables are sorted on the join columns. Merge joins are faster and use less memory than the alternative when there is no index to be used: hash joins.

- **Hash join** is used when projections of the joined tables are not already sorted on the join columns, and when no index can be used.
  - In this case, the optimizer builds an **in-memory hash table** on the **inner table's** join column. The optimizer then scans the **outer table** for matches to the hash table, and joins data from the two tables accordingly.
  - The cost of performing a hash join is low if the entire hash table can fit in memory (RAM). Cost rises significantly if the hash table must be written to disk.

2.2. Step 2: Execution algorithms of RA operations

- Partition-hash JOIN
  - Hash two relations on join attributes
  - Join buckets accordingly



2.2. Step 2: Execution algorithms of RA operations

- Partition-hash JOIN: implementation (conceptual)
  - (1) Partitioning phase:
    - Builds an **in-memory hash table** to hash R1 tuples into k buckets (partitions) based on the join column(s)
    - Hash R2 tuples into k buckets (partitions) based on the hash table
  - (2) Joining phase (nested block join for each pair of partitions):
    - For i = 0 to k do
      - join tuples in the ith partitions of R1 and R2

2.2. Step 2: Execution algorithms of RA operations

Algorithms for Join - Summary

Join algorithm	Disk I/O Cost	Notes
Nested block join	b(R1) + b(R1)*b(R2)/(M-2)	ok for "small" relations (relative to memory size); I/O = b(R1) + b(R2) if R1 can fit into buffer
Sort-merge join w/o sort	b(R1) + b(R2)	best if relations are sorted; good for non-equi-join (e.g., R1.C > R2.C)
Sort-merge join w/ sort	$(2\log_M b(R1) +1) * b(R1) + (2\log_M b(R2) +1) * b(R2)$	
Hash join	3b(R1) + 3b(R2)	best for equi-join if relations are not sorted and no indexes exist
Index join	b(R1) + r(R1) * (Index search cost on R2)	could be useful if index exists but depends on expected result size

2.2. Step 2: Execution algorithms of RA operations

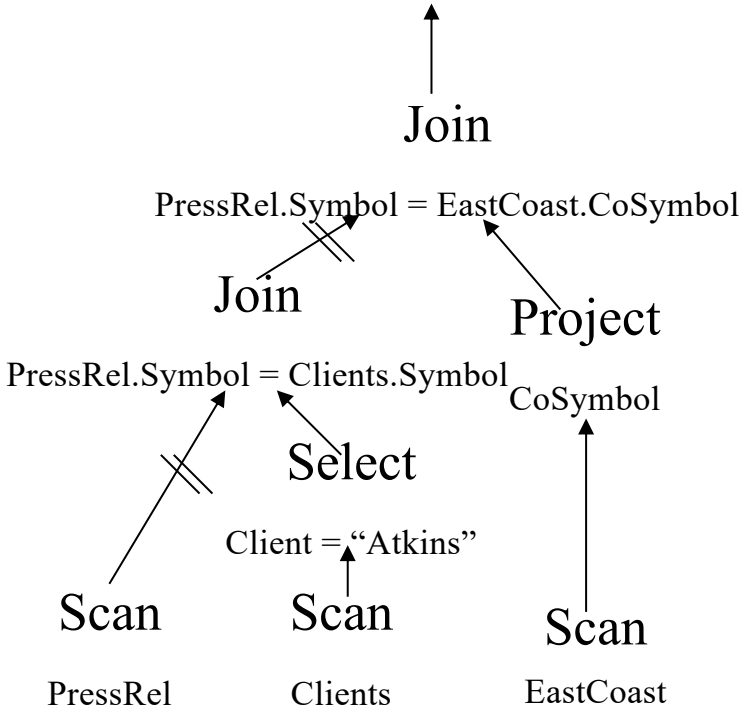
- Exercise
  - Consider the relations R1(A, B,C), R2(C, D, E), and R3(E, F), with primary keys A, C, and E, respectively.
  - Assume that R1 has 1000 tuples, R2 has 1500 tuples, and R3 has 750 tuples.
    1. Give the **maximum** size of R1 ⋈ R2 ⋈ R3,
    2. Give an efficient strategy for computing the join.

2.2. Step 2: Execution algorithms of RA operations

- Exercise solution

2.2. Step 2: Execution algorithms of RA operations

- **Execution plan tree:** to be read from the leaves to the root
- The DBMS can display the execution plan
  - The syntax depends on the DBMS



2.2. Step 2: Execution algorithms of RA operations

- Execution plan tree:

```
select prod_category, avg(amount_sold)
from sales s, products p
where p.prod_id = s.prod_id
group by prod_category;
```

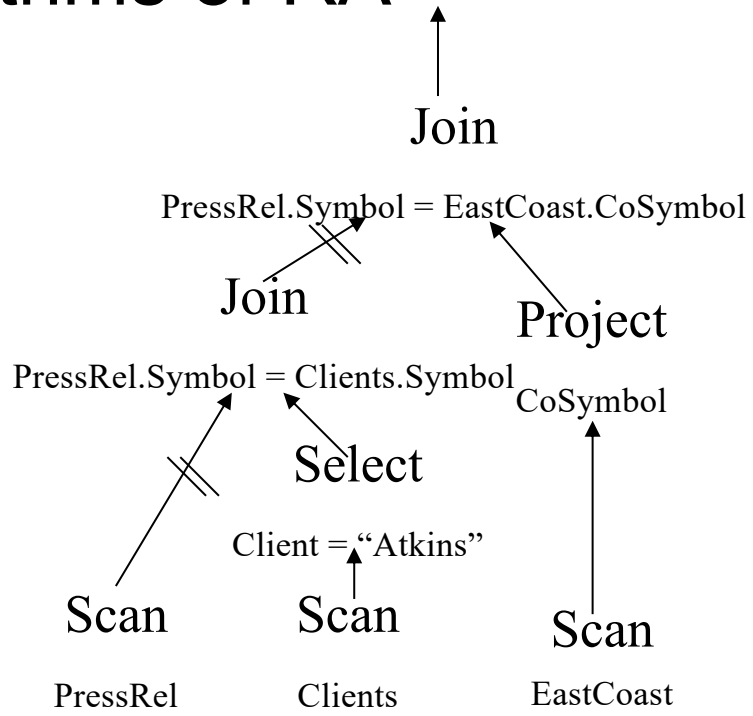
The tabular representation of this query's plan is:

```
-----------------------------------------
 Id   Operation              Name
-----------------------------------------
   0  SELECT STATEMENT
   1   HASH GROUP BY
   2    HASH JOIN
   3     TABLE ACCESS FULL   PRODUCTS
   4     PARTITION RANGE ALL
   5      TABLE ACCESS FULL  SALES
-----------------------------------------
```

[https://blogs.oracle.com/optimizer/how-do-i-display-and-read-the-execution-plans-for-a-sql-statement]

2.2. Step 2: Execution algorithms of RA operations

- Execution Strategy: Materialization vs. Pipelining
  - Execution strategy defines how to walk the query execution plan
    - Materialization
    - Pipelining



Execution plan tree: to read from the leaves to the root…

2.2. Step 2: Execution algorithms of RA operations

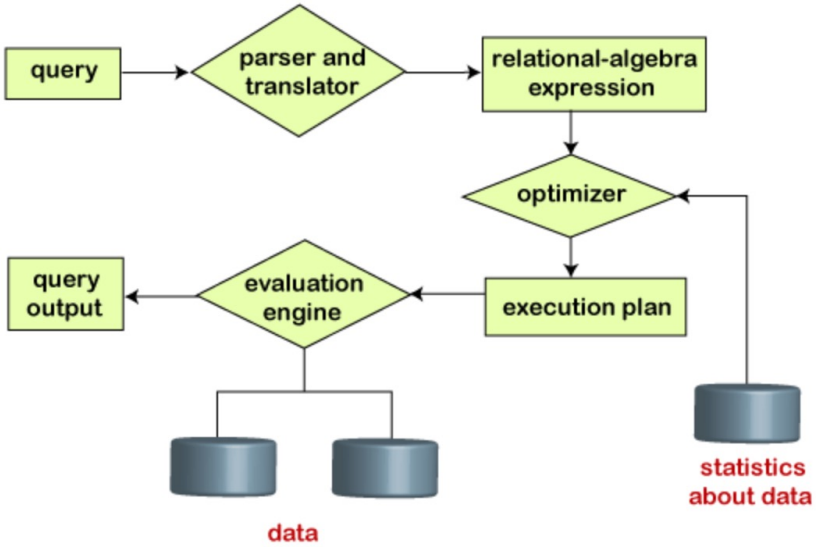
- **Materialization**
  - Performs the innermost (or leaf-level) operations first of the query execution plan
  - The intermediate result of each operation is materialized into **temporary relation** and becomes input for the following operations (upper in the tree)
  - The cost of materialization is the sum of the individual operations plus the cost of writing the intermediate results to disk
    - lots of temporary files, lots of I/O.

2.2. Step 2: Execution algorithms of RA operations

- **Pipelining**
  - Operations form a queue, and results are passed from one operation to another as they are calculated
  - Pipelining restructures the individual operation algorithms so that they take streams of tuples as both input and output.
  - **Limitation**
    - algorithms that require sorting can only use pipelining if the input is already sorted beforehand
      - since sorting, by nature, cannot be performed until all tuples to be sorted are known.

2.3. Step 3: Cost estimation

- Last step for the optimizer
- Each relational algebra expression can result in many query execution plans
- Some query execution plans may be better than others
- Finding the fastest one
  - Just an estimation under certain assumptions
    - And also, some statistics from the database's metadata
      - These statistics might be outdated (see next slide)…
  - Huge number of query plans may exist



Steps in query processing

2.3. Step 3: Cost estimation

- Cost estimation factors
  - Catalog information: database maintains statistics about relations
  - Ex.
    - number of tuples per relation
    - number of blocks on disk per relation
    - number of distinct values per attribute
    - histogram of values per attribute
  - Problems
    - cost can only be estimated
    - updating statistics is expensive, thus they are often out of date

2.3. Step 3: Cost estimation

- Choosing the cheapest query plan
  - **Problem:**
    - Estimating cost for all possible plans too expensive.
  - **Solutions:**
    - Pruning: stop early to evaluate a plan
    - Heuristics: do not evaluate all plans
    - Examples of heuristics:
      - perform selections as early as possible
      - perform projections early, avoid Cartesian products
  - **Real databases use a combination of**
    - Applying heuristics to choose promising query plans.
    - Choose the cheapest plan among the promising plans, using pruning.

2.3. Step 3: Cost estimation

- Heuristic rules – more details
  - Break apart conjunctive selections into a sequence of simple selections
  - Move σ down the query tree as soon as possible
  - Replace σ-x pairs by $\bowtie$
  - Break apart and move Π down the tree as soon as possible
  - Perform the joins with the smallest expected result first

2.3. Step 3: Cost estimation

- After it performed its 3 steps, the optimizer might still have selected a suboptimal query execution plan
  - Some DBMS's optimizers are more effective than others…
- In general, it is not a problem
  - But it might be a problem if you are dealing with VERY big tables
  - In most cases, you cannot really modify the optimizer's query execution plan…
    - For instance, in most cases, the way you formulate your SQL statement does not influence the optimizer (except maybe for extremely complex queries and low-effectiveness optimizers)
  - … But you still can influence the optimizer so that it gives better plan (by provinding hints for instance, like forcing the optimizer to use a given index)
    - Examples with ORACLE: https://docs.oracle.com/database/121/TGSQL/tgsql_influence.htm#TGSQL246

Summary

- Query processing is the entire process or activities involved in retrieving data from the database
  - Parser
  - Optimizer
  - Code generator
- Query optimizer
  - Step 1: Equivalence transformation
  - Step 2: Annotation for the algorithm of the RA expression
  - Step 3: Cost estimation for different query execution plans

Quiz 1.

Quiz Number	1	Quiz Type	OX	Example Select
Question	What is the output of the parser in query processing?			
Example	A. Query execution plan B. Query plan C. Relational algebra expression D. Code for executing			
Answer				
Feedback				

Quiz 2.

Quiz Number	2	Quiz Type	OX	Example Select
Question	What can the query optimizer do?			
Example	A. Equivalence transformation B. Annotation for the algorithmic execution of the RA expression C. Cost estimation D. Executing the query plan and return results			
Answer				
Feedback				

Summary

- Overview of query processing
  - Parser
  - Optimizer
- Understanding query optimizer
  - Step 1: Equivalence transformation
  - Step 2: Annotation for the algorithmic execution of the RA expression
  - Step 3: Cost estimation for different query execution plans

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!

soict.hust.edu.vn/ fb.com/groups/soict