Distributed Database Systems
Fall 2018

Query Processing
SL03

- ▶ Query Processing Overview
- ▶ Distributed Query Processing Steps
- ▶ Query Decomposition
- ▶ Data Localization

Query Processing Overview/1

- ▶ **Query processing**: A 3-step process that transforms a high-level query (of relational calculus/SQL) into an **equivalent** and **more efficient** lower-level query (of relational algebra).
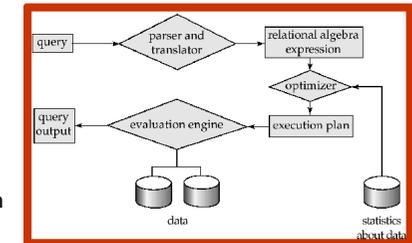
  1. **Parsing and translation**
     - ▶ Check syntax and verify relations.
     - ▶ Translate the query into an equivalent relational algebra expression.

  2. **Optimization**
     - ▶ Generate an optimal evaluation plan (with lowest cost) for the query plan.
  3. **Evaluation**
     - ▶ The query-execution engine takes an (optimal) evaluation plan, executes that plan, and returns the answers to the query.

Query Processing Overview/2

- ▶ The success of RDBMSs is due, in part, to the availability of
  - ▶ declarative query languages that allow to easily express complex queries without knowing the details of the physical data organization and
  - ▶ advanced query processing technology that transforms the high-level user/application queries into efficient lower-level query execution strategies.
- ▶ The query transformation should achieve both **correctness** and **efficiency**
  - ▶ The main difficulty is to achieve efficiency
  - ▶ This is also one of the most important tasks of any DBMS

Query Processing Overview/3

- ▶ **Distributed query processing**: Transform a high-level query (of relational calculus/SQL) on a distributed database (i.e., a set of global relations) into an **equivalent** and **efficient** lower-level query (of relational algebra) on relation fragments.
- ▶ Distributed query processing is more complex
  - ▶ Not only ordering of operations must be considered
  - ▶ Fragmentation/replication of relations
  - ▶ There are operations for exchanging data between sites
  - ▶ Additional communication costs
  - ▶ Transformations of data must be considered
  - ▶ Parallel execution

Query Processing Example/1

- ▶ **Example:** Transformation of an SQL-query into an RA-query.
- ▶ Relations:
    - ▶ EMP(ENO, ENAME, TITLE)
    - ▶ ASG(ENO,PNO,RESP,DUR)
- ▶ Query: *Find the names of employees who are managing a project?*
    - ▶ High level query

        ```sql
        SELECT ENAME
        FROM EMP,ASG
        WHERE EMP.ENO = ASG.ENO AND RESP = 'MGR'
        ```

    - ▶ Two possible transformations of the query are:
        - ▶ Expression 1: $\pi_{ENAME}(\sigma_{RESP='MGR' \wedge EMP.ENO=ASG.ENO}(EMP \times ASG))$
        - ▶ Expression 2: $\pi_{ENAME}(EMP \bowtie_{ENO} (\sigma_{RESP='MGR'}(ASG)))$
    - ▶ Expression 2 avoids the expensive and large intermediate Cartesian product, and therefore typically is better.
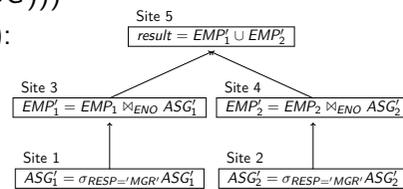
Query Processing Example/2

- ▶ We make the following assumptions about the data fragmentation
    - ▶ Data is (horizontally) fragmented:
        - ▶ Site1: $ASG1 = \sigma_{ENO \leq 'E3'}(ASG)$
        - ▶ Site2: $ASG2 = \sigma_{ENO > 'E3'}(ASG)$
        - ▶ Site3: $EMP1 = \sigma_{ENO \leq 'E3'}(EMP)$
        - ▶ Site4: $EMP2 = \sigma_{ENO > 'E3'}(EMP)$
        - ▶ Site5: Result
- ▶ Relations ASG and EMP are fragmented in the same way
- ▶ Relations ASG and EMP are locally clustered on attributes RESP and ENO, respectively. Thus, there is direct access to tuples of ASG and EMP based on the values of attributes RESP and ENO.

Query Processing Example/3

- ▶ Now consider the expression
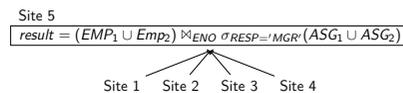 $\pi_{ENAME}(EMP \bowtie_{ENO} (\sigma_{RESP='MGR'}(ASG)))$
- ▶ Strategy 1 (partially parallel execution):
    - ▶ Produce ASG'_1 and move to Site 3
    - ▶ Produce ASG'_2 and move to Site 4
    - ▶ Join ASG'_1 with EMP_1 at Site 3 and move the result to Site 5
    - ▶ Join ASG'_2 with EMP_2 at Site 4 and move the result to Site 5
    - ▶ Union the result in Site 5

- ▶ Strategy 2:
    - ▶ Move ASG_1 and ASG_2 to Site 5
    - ▶ Move EMP_1 and EMP_2 to Site 5
    - ▶ Select and join at Site 5

- ▶ For simplicity, the final projection is omitted.

Query Processing Example/4

- ▶ Calculate the cost of the two strategies under the following assumptions:
    - ▶ Tuples are uniformly distributed to the fragments
    - ▶ 20 tuples satisfy RESP='MGR'
    - ▶ size(EMP) = 400
    - ▶ size(ASG) = 1000
    - ▶ tuple access cost = 1 unit;
    - ▶ tuple transfer cost = 10 units
    - ▶ ASG and EMP have a local index on RESP and ENO

Query Processing Example/5

- ▶ Strategy 1

  - ▶ Produce ASG_1', ASG_2': (10+10) * tuple access cost   20
  - ▶ Transfer ASG_1', ASG_2' to sites of EMPs:
    (10+10) * tuple transfer cost   200
  - ▶ Produce EMP_1', EMP_2': (10+10) * tuple access cost * 2   40
  - ▶ Transfer EMP_1', EMP_2' to result site:
    (10+10) * tuple transfer cost   200
  - ▶ Total cost   460

Query Processing Example/6

- ▶ Strategy 2

  - ▶ Transfer EMP_1, EMP_2 to site 5: 400 * tuple transfer cost   4,000
  - ▶ Transfer ASG_1, ASG_2 to site 5: 1000 * tuple transfer cost   10,000
  - ▶ Select tuples from $ASG_1 \cup ASG_2$: 1000 * tuple access cost   1,000
  - ▶ Join EMP and ASG': 400 * 20 * tuple access cost   8,000
  - ▶ Total cost   23,000

Distributed Query Processing Steps
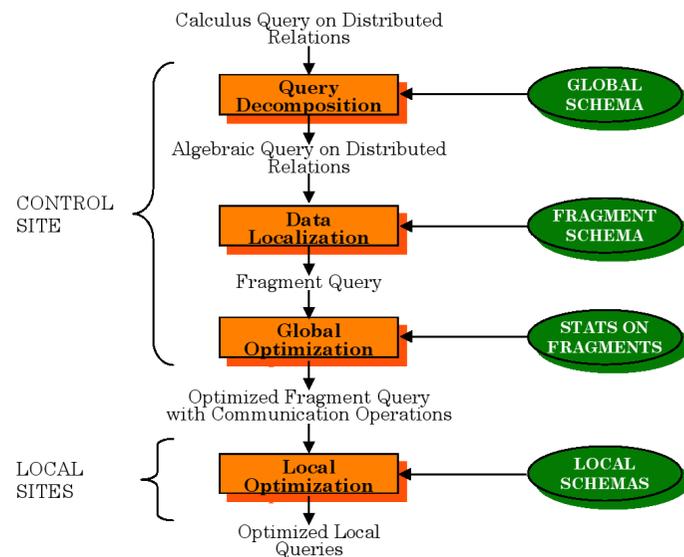
Query Decomposition

- ▶ **Query decomposition:** Mapping of calculus query (SQL) to algebra operations (select, project, join, rename)
- ▶ Input and output queries refer to global relations, without knowledge of the data distribution.
- ▶ The output query is semantically correct and good in the sense that redundant work is avoided.
- ▶ Query decomposistion consists of 4 steps:
  1. **Normalization**: Transform query to a normalized form
  2. **Analysis**: Detect and reject "incorrect" queries; possible only for a subset of relational calculus
  3. **Elimination of redundancy**: Eliminate redundant predicates
  4. **Rewriting**: Transform query to RA and optimize query
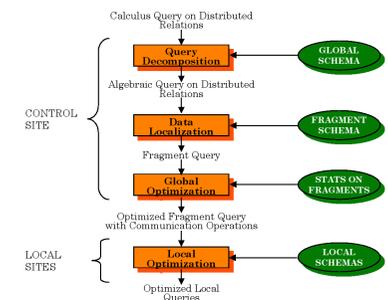
Query Normalization/1

- ▶ **Normalization:** Transform the query to a normalized form to facilitate further processing. Consists mainly of two steps.
  1. **Lexical** and **syntactic** analysis
     - ▶ Check validity (similar to compilers)
     - ▶ Check for attributes and relations
     - ▶ Type checking on the qualification
  2. Put into **normal form**
     - ▶ With SQL, the query qualification (WHERE clause) is the most difficult part as it might be an arbitrary complex predicate preceded by quantifiers ($\exists, \forall$)
     - ▶ Conjunctive normal form

       $$(p_{11} \lor p_{12} \lor \cdots \lor p_{1n}) \land \cdots \land (p_{m1} \lor p_{m2} \lor \cdots \lor p_{mn})$$

     - ▶ Disjunctive normal form

       $$(p_{11} \land p_{12} \land \cdots \land p_{1n}) \lor \cdots \lor (p_{m1} \land p_{m2} \land \cdots \land p_{mn})$$

     - ▶ In the disjunctive normal form, the query can be processed as independent conjunctive subqueries linked by unions (corresponding to the disjunction). Might yield replicated predicates for joins and selections.

Query Normalization/2

- ▶ **Example:** Consider the following query: *Find the names of employees who have been working on project P1 for 12 or 24 months?*

- ▶ The query in SQL:

  ```sql
  SELECT ENAME
  FROM EMP, ASG
  WHERE EMP.ENO = ASG.ENO
  AND ASG.PNO = 'P1' AND ( DUR = 12 OR DUR = 24 )
  ```

- ▶ The qualification in conjunctive normal form:

  $$EMP.ENO = ASG.ENO \land ASG.PNO =' P1' \land (DUR = 12 \lor DUR = 24)$$

- ▶ The qualification in disjunctive normal form:

  $$(EMP.ENO = ASG.ENO \land ASG.PNO =' P1' \land DUR = 12) \lor$$
  $$(EMP.ENO = ASG.ENO \land ASG.PNO =' P1' \land DUR = 24)$$

Query Analysis/1

- ▶ **Analysis:** Identify and reject type incorrect or semantically incorrect queries
- ▶ Type incorrect
  - ▶ Checks whether the attributes and relation names of a query are defined in the global schema
  - ▶ Checks whether the operations on attributes do not conflict with the types of the attributes, e.g., $ENAME > 200$ is not type correct
- ▶ Semantically incorrect
  - ▶ Checks whether the components contribute in any way to the generation of the result
  - ▶ Only a subset of relational calculus queries can be tested for correctness, i.e., those that do not contain disjunction and negation
  - ▶ Typical data structures used to detect the semantically incorrect queries are:
    - ▶ Connection graph (query graph)
    - ▶ Join graph

Query Analysis/2

- ▶ **Example:** Consider a query:

  ```sql
  SELECT ENAME, RESP
  FROM EMP, ASG, PROJ
  WHERE EMP.ENO = ASG.ENO
  AND ASG.PNO = PROJ.PNO
  AND PNAME = 'CAD/CAM'
  AND DUR >= 36
  AND TITLE = 'Programmer'
  ```
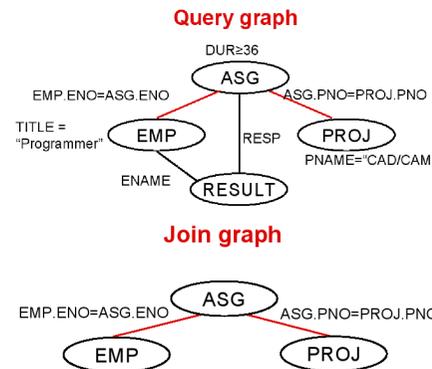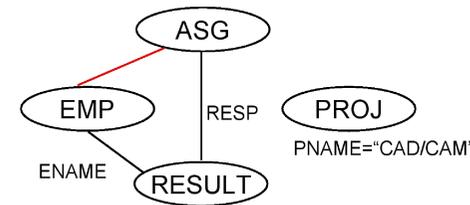
Query Analysis/3

- ► Query/connection graph
  - ► Nodes represent operand or result relation
  - ► Edge represents a join if both connected nodes represent an operand relation, otherwise it is a projection
- ► Join graph
  - ► a subgraph of the query graph that considers only the joins
- ► Since the query graph **is connected**, the query is semantically correct

Query graph

DUR≥36

EMP.ENO=ASG.ENO — ASG — ASG.PNO=PROJ.PNO

TITLE = "Programmer" — EMP — RESP — PROJ

PNAME="CAD/CAM"

ENAME — RESULT

Join graph

ASG

EMP.ENO=ASG.ENO — ASG.PNO=PROJ.PNO

EMP   PROJ

Query Analysis/4

- ► **Example:** Consider the following query and its query graph:

```
SELECT ENAME,RESP
FROM EMP, ASG, PROJ
WHERE EMP.ENO = ASG.ENO
AND PNAME = 'CAD/CAM'
AND DUR >= 36
AND TITLE = 'Programmer'
```

ASG

EMP — RESP — PROJ

PNAME="CAD/CAM"

ENAME — RESULT

Query Analysis/4

- ► Since the graph **is not connected**, the query is semantically incorrect.
- ► 3 possible solutions:
  - ► Reject the query
  - ► Assume an implicit Cartesian Product between ASG and PROJ
  - ► Infer from the schema the missing join predicate ASG.PNO = PROJ.PNO

Elimination of Redundancy/1

- ► **Elimination of redundancy:** Simplify the query by eliminating redundancies, e.g., redundant predicates
  - ► Redundancies are often due to semantic integrity constraints expressed in the query language
  - ► e.g., queries on views are expanded into queries on relations that satisfy certain integrity and security constraints
- ► Transformation rules are used, e.g.,
  - ► $p \wedge p \iff p$
  - ► $p \vee p \iff p$
  - ► $p \wedge true \iff p$
  - ► $p \vee false \iff p$
  - ► $p \wedge false \iff false$
  - ► $p \vee true \iff true$
  - ► $p \wedge \neg p \iff false$
  - ► $p \vee \neg p \iff true$
  - ► $p_1 \wedge (p_1 \vee p_2) \iff p_1$
  - ► $p_1 \vee (p_1 \wedge p_2) \iff p_1$

Elimination of Redundancy/2

- ▶ **Example:** Consider the following query:

  ```sql
  SELECT TITLE
  FROM EMP
  WHERE EMP.ENAME = 'J. Doe'
  OR ( NOT(EMP.TITLE = 'Programmer')
  AND ( EMP.TITLE = 'Elect. Eng.'
  OR EMP.TITLE = 'Programmer' )
  AND NOT(EMP.TITLE = 'Elect. Eng.') )
  ```

- ▶ Let p_1 be ENAME = 'J. Doe', p_2 be TITLE = 'Programmer' and p_3 be TITLE = 'Elect. Eng.'
- ▶ Then the qualification can be written as $p_1 \vee (\neg p_2 \wedge (p_2 \vee p_3) \wedge \neg p_3)$ and then be transformed into p_1
- ▶ Simplified query:

  ```sql
  SELECT TITLE
  FROM EMP
  WHERE EMP.ENAME = 'J. Doe'
  ```

Query Rewriting/1

- ▶ **Rewriting:** Convert relational calculus query to relational algebra query and find an **efficient** expression.
- ▶ **Example:** Find the names of employees other than John Doe who worked on the CAD/CAM project for either 1 or 2 years.
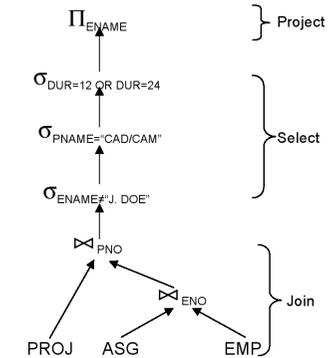
  - ▶
    ```sql
    SELECT ENAME
    FROM EMP, ASG, PROJ
    WHERE EMP.ENO = ASG.ENO
    AND ASG.PNO = PROJ.PNO
    AND ENAME <> 'J. Doe'
    AND PNAME = 'CAD/CAM'
    AND (DUR = 12 OR DUR = 24)
    ```

- ▶ A **query tree** represents the RA-expression
  - ▶ Relations are leaves (FROM clause)
  - ▶ Result attributes are root (SELECT clause)
  - ▶ Intermediate leaves should give a result from the leaves to the root

---

Query Rewriting/2

- ▶ By applying **transformation rules**, many different trees/expressions may be found that are **equivalent** to the original tree/expression, but might be more efficient.
- ▶ In the following we assume relations $R(A_1, \ldots, A_n)$ and $S(B_1, \ldots, B_n)$, which is union-compatible to R.
- ▶ **Commutativity** of binary operations
  - ▶ $R \times S = S \times R$
  - ▶ $R \bowtie S = S \bowtie R$
  - ▶ $R \cup S = S \cup R$
- ▶ **Associativity** of binary operations
  - ▶ $(R \times S) \times T = R \times (S \times T)$
  - ▶ $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- ▶ **Idempotence** of unary operations
  - ▶ $\pi_A(\pi_A(R)) = \pi_A(R)$
  - ▶ $\sigma_{p1(A1)}(\sigma_{p2(A2)}(R)) = \sigma_{p1(A1) \wedge p2(A2)}(R)$
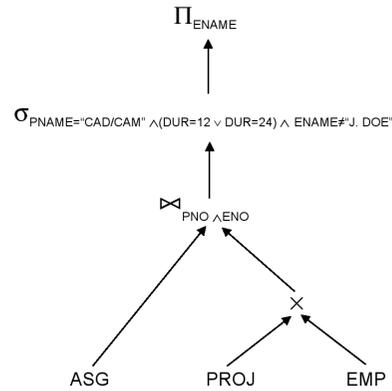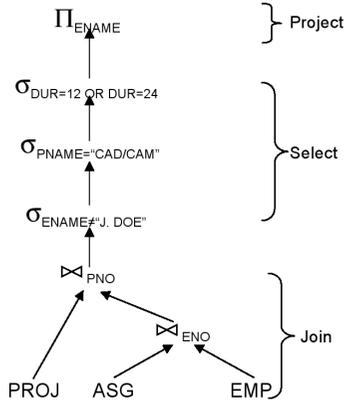

Query Rewriting/3

- ▶ **Commuting selection** with binary operations
  - ▶ $\sigma_{p(A)}(R \times S) \iff \sigma_{p(A)}(R) \times S$
  - ▶ $\sigma_{p(A_1)}(R \bowtie_{p(A_2,B_2)} S) \iff \sigma_{p(A_1)}(R) \bowtie_{p(A_2,B_2)} S$
  - ▶ $\sigma_{p(A)}(R \cup T) \iff \sigma_{p(A)}(R) \cup \sigma_{p(A)}(T)$
    - ▶ (A belongs to R and T)

- ▶ **Commuting projection** with binary operations (assume $C = A' \cup B'$, $A' \subseteq A$, $B' \subseteq B$)
  - ▶ $\pi_C(R \times S) \iff \pi_{A'}(R) \times \pi_{B'}(S)$
  - ▶ $\pi_C(R \bowtie_{p(A',B')} S) \iff \pi_{A'}(R) \bowtie_{p(A',B')} \pi_{B'}(S)$
  - ▶ $\pi_C(R \cup S) \iff \pi_C(R) \cup \pi_C(S)$

Query Rewriting/4

- ▶ **Example:** Two equivalent query trees for the previous example
  - ▶ Recall the schemas: EMP(ENO, ENAME, TITLE)
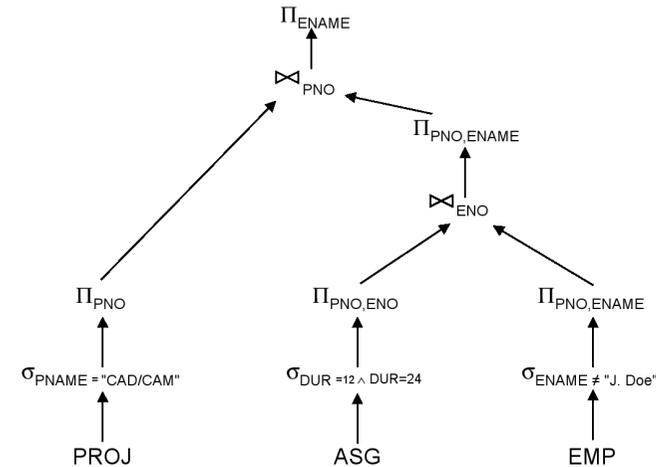   PROJ(PNO, PNAME, BUDGET)
    ASG(ENO, PNO, RESP, DUR)

Π_{ENAME} — Project

$\sigma_{DUR=12\ OR\ DUR=24}$

$\sigma_{PNAME="CAD/CAM"}$ — Select

$\sigma_{ENAME\neq"J.\ DOE"}$

$\bowtie_{PNO}$

$\bowtie_{ENO}$ — Join

PROJ ASG EMP

Π_{ENAME}

$\sigma_{PNAME="CAD/CAM"\ \wedge(DUR=12\ \vee\ DUR=24)\ \wedge\ ENAME\neq"J.\ DOE"}$

$\bowtie_{PNO\ \wedge ENO}$

$\times$

ASG PROJ EMP

Query Rewriting/5

- ▶ **Example (contd.):** Another equivalent query tree, which allows a more efficient query evaluation, since the most selective operations are applied first.

Π_{ENAME}
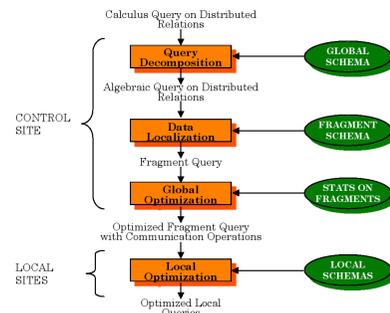
$\bowtie_{PNO}$

$\Pi_{PNO,ENAME}$

$\bowtie_{ENO}$

Π_{PNO} $\Pi_{PNO,ENO}$ $\Pi_{PNO,ENAME}$

$\sigma_{PNAME\ =\ "CAD/CAM"}$ $\sigma_{DUR\ =12\ \wedge\ DUR=24}$ $\sigma_{ENAME\ \neq\ "J.\ Doe"}$

PROJ   ASG   EMP

Data Localization/1

- ▶ **Data localization**
  - ▶ **Input:** Algebraic query on global conceptual schema
  - ▶ **Purpose:**
    - ▶ Apply data distribution information to the algebra operations and determine which fragments are involved
    - ▶ Substitute global query with queries on fragments
    - ▶ Optimize the global query

Calculus Query on Distributed Relations

Query Decomposition ← GLOBAL SCHEMA

Algebraic Query on Distributed Relations

CONTROL SITE

Data Localization ← FRAGMENT SCHEMA

Fragment Query

Global Optimization ← STATS ON FRAGMENTS

Optimized Fragment Query with Communication Operations

LOCAL SITES

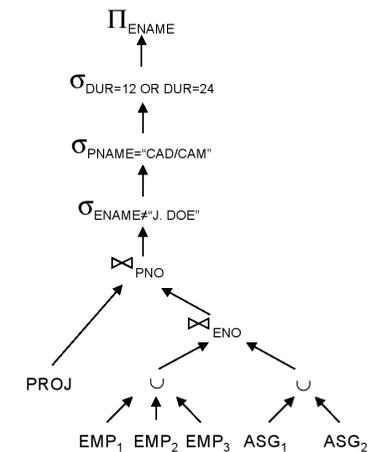Local Optimization ← LOCAL SCHEMAS

Optimized Local Queries

Data Localization/2

- ▶ **Example:**
  - ▶ Assume EMP is horizontally fragmented into EMP1, EMP2, EMP3 as follows:
    - ▶ $EMP1 = \sigma_{ENO\leq'E3'}(EMP)$
    - ▶ $EMP2 = \sigma_{'E3'<ENO\leq'E6'}(EMP)$
    - ▶ $EMP3 = \sigma_{ENO>'E6'}(EMP)$
  - ▶ ASG fragmented into ASG1 and ASG2 as follows:
    - ▶ $ASG1 = \sigma_{ENO\leq'E3'}(ASG)$
    - ▶ $ASG2 = \sigma_{ENO>'E3'}(ASG)$
- ▶ Simple approach: Replace in all queries
  - ▶ EMP by (EMP1∪EMP2∪ EMP3)
  - ▶ ASG by (ASG1∪ASG2)
  - ▶ Result is also called **generic query**
- ▶ In general, the **generic query is inefficient** since important restructurings and simplifications can be done.
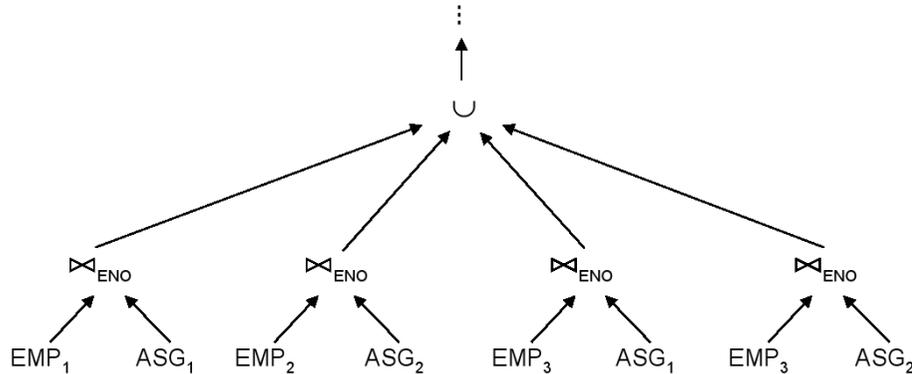
Π_{ENAME}

$\sigma_{DUR=12\ OR\ DUR=24}$

$\sigma_{PNAME="CAD/CAM"}$

$\sigma_{ENAME\neq"J.\ DOE"}$

$\bowtie_{PNO}$

$\bowtie_{ENO}$

PROJ   ∪   ∪

EMP$_1$ EMP$_2$ EMP$_3$ ASG$_1$ ASG$_2$

Data Localization/3

- ▶ **Example (contd.)**: Parallelsim in the evaluation is often possible
  - ▶ Depending on the horizontal fragmentation, the fragments can be joined in parallel followed by the union of the intermediate results.
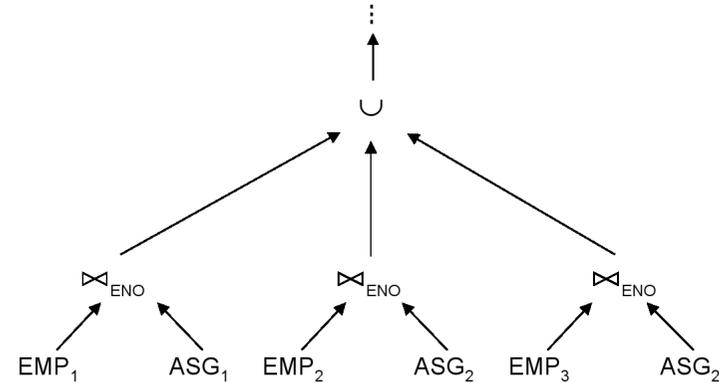
Data Localization/4

- ▶ **Example (contd.)**: Unnecessary work can be eliminated
  - ▶ e.g., $EMP_3 \bowtie ASG_1$ gives an empty result
    - ▶ $EMP3 = \sigma_{ENO > 'E6'}(EMP)$
    - ▶ $ASG1 = \sigma_{ENO \leq 'E3'}(ASG)$

Data Localizations Issues

- ▶ Various more advanced **reduction techniques** are possible to generate simpler and optimized queries.
- ▶ Reduction of horizontal fragmentation (HF)
  - ▶ Reduction with selection
  - ▶ Reduction with join
- ▶ Reduction of vertical fragmentation (VF)
  - ▶ Find empty relations

Reduction for HF/1

- ▶ **Reduction with selection for HF**
  - ▶ Consider relation R with horizontal fragmentation $F = \{R_1, R_2, \ldots, R_k\}$, where $R_i = \sigma_{p_i}(R)$
  - ▶ **Rule1:** Selections on fragments, $\sigma_\theta(R_i)$, that have a qualification contradicting the qualification of the fragmentation generate empty relations, i.e., $\sigma_\theta(R_i) = \emptyset \iff \forall x \in R(p_i(x) \wedge \theta(x) = false)$
  - ▶ Can be applied if fragmentation predicate is inconsistent with the query selection predicate.

- ▶ **Example:** Consider the query:
  ```
  SELECT * FROM EMP WHERE ENO='E5'
  ```
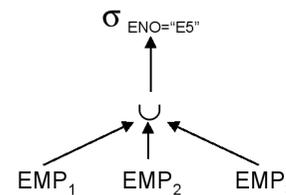

After commuting the selection with the union operation, it is easy to detect that the selection predicate contradicts the predicates of EMP_1 and EMP_3.
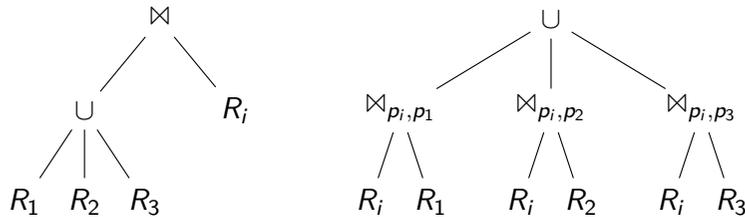
Reduction for HF/2

- ▶ **Reduction with join for HF**
  - ▶ Joins on horizontally fragmented relations can be simplified when the joined relations are fragmented according to the join attributes.
  - ▶ Distribute join over union

  $$(R_1 \cup R_2) \bowtie S \iff (R_1 \bowtie S) \cup (R_2 \bowtie S)$$

  - ▶ **Rule 2**: Useless joins of fragments, $R_i = \sigma_{p_i}(R)$ and $R_j = \sigma_{p_j}(R)$, can be determined when the qualifications of the joined fragments are contradicting, i.e.,

  $$R_i \bowtie R_j = \emptyset \iff \forall x \in R_i, \forall y \in R_j(p_i(x) \wedge p_j(y) = false)$$

Reduction for HF/3

- ▶ **Example:** Consider the following query and fragmentation:
  - ▶ Query:
    SELECT $\star$ **FROM** EMP, ASG **WHERE** EMP.ENO=ASG.ENO
  - ▶ Horizontal fragmentation:
    - ▶ $EMP1 = \sigma_{ENO \leq 'E3'}(EMP)$
    - ▶ $EMP2 = \sigma_{'E3' < ENO \leq 'E6'}(EMP)$
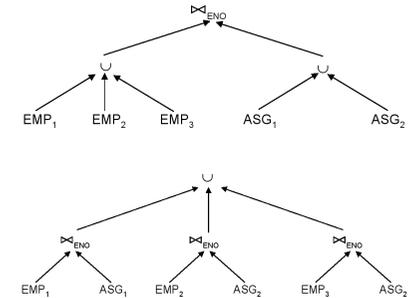   - ▶ $EMP3 = \sigma_{ENO > 'E6'}(EMP)$
    - ▶ $ASG1 = \sigma_{ENO \leq 'E3'}(ASG)$
    - ▶ $ASG2 = \sigma_{ENO > 'E3'}(ASG)$
  - ▶ Generic query
  - ▶ The query reduced by distributing joins over unions and applying rule 2 can be implemented as a union of three partial joins that can be done in parallel.

03.6, 03.7

Reduction for HF/4

- ▶ **Reduction with join for derived HF**
  - ▶ The horizontal fragmentation of one relation is **derived** from the horizontal fragmentation of another relation by using semijoins.
- ▶ If the fragmentation is not on the same predicate as the join, derived horizontal fragmentation can be applied in order to make efficient join processing possible.
- ▶ **Example:** Assume the following query and fragmentation of the EMP relation:
  - ▶ Query:
    SELECT $\star$ **FROM** EMP, ASG **WHERE** EMP.ENO=ASG.ENO
  - ▶ Fragmentation (**not** on the join attribute):
    - ▶ $EMP1 = \sigma_{\text{TITLE}=\text{"Programmer"}}(EMP)$
    - ▶ $EMP2 = \sigma_{\text{TITLE} \neq \text{"Programmer"}}(EMP)$
  - ▶ To achieve efficient joins ASG can be fragmented as follows:
    - ▶ $ASG1 = ASG \ltimes_{ENO} EMP1$
    - ▶ $ASG2 = ASG \ltimes_{ENO} EMP2$
  - ▶ The fragmentation of ASG is derived from the fragmentation of EMP
  - ▶ Queries on derived fragments can be reduced, e.g.,
    $ASG_1 \bowtie EMP_2 = \emptyset$

Reduction for VF/1

- ▶ **Reduction for Vertical Fragmentation**
  - ▶ Recall, VF distributes a relation based on projection, and the reconstruction operator is the join.
  - ▶ Similar to HF, it is possible to identify useless intermediate relations, i.e., fragments that do not contribute to the result.
  - ▶ Assume a relation $R(A)$ with $A = \{A_1, \ldots, A_n\}$, which is vertically fragmented as $R_i = \pi_{A_i'}(R)$, where $A_i' \subseteq A$.
  - ▶ **Rule 3**: $\pi_{D,K}(R_i)$ is useless if the set of projection attributes D is not in A_i' and K is the key attribute.
  - ▶ Note that the result is not empty, but it is useless, as it contains only the key attribute.

Reduction for VF/2

- ▶ **Example:** Consider the following query and vertical fragmentation:
  - ▶ Query: **SELECT** ENAME **FROM** EMP
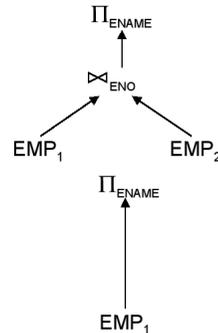 - ▶ Fragmentation:
    - ▶ $EMP1 = \pi_{ENO,ENAME}(EMP)$
    - ▶ $EMP2 = \pi_{ENO,TITLE}(EMP)$

- ▶ Generic query

- ▶ Reduced query
  - ▶ By commuting the projection with the join (i.e., projecting on ENO, ENAME), we can see that the projection on EMP_2 is useless because ENAME is not in EMP_2.

$$\Pi_{ENAME}$$
$$\bowtie_{ENO}$$
$$EMP_1 \qquad EMP_2$$
$$\Pi_{ENAME}$$
$$EMP_1$$

Conclusion/1

- ▶ Query processing transforms a high level query (relational calculus) into an equivalent lower level query (relational algebra). The main difficulty is to achieve the efficiency in the transformation

- ▶ Query processing is done in the following sequence: query decomposition→data localization→global optimization→ local optimization

- ▶ **Query decomposition** and **data localization** maps calculus query into algebra operations and applies data distribution information to the algebra operations.

- ▶ **Query decomposition** consists of normalization, analysis, elimination of redundancy, and rewriting.

- ▶ **Data localization** reduces horizontal fragmentation with join and selection, and vertical fragmentation with joins, and aims to find empty relations.