Chapter 15: Query Processing

(and also, chapter 22: Parallel Query Processing)

Sistemas de Bases de Dados 2020/21

Capítulo refere-se a: Database System Concepts, 7th Ed

Chapter 15: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
- Parallel query processing
- Query processing in Oracle

Intraquery Parallelism

- **Intraquery parallelism**: execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism:
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.
    - Limited degree of parallelism
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query
    - Supports high degree of parallelism

Parallel Processing of Relational Operations

- Our discussion of parallel algorithms assumes:
  - *read-only* queries
  - shared-nothing architecture
  - *n* nodes, N_1, ..., N_n
    - *Each assumed to have disks and processors*.
  - Initial focus on parallelization to a shared-nothing node
    - Parallel processing within a shared memory/shared disk node discussed later
  - Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.
    - Algorithms for shared-nothing systems can thus be run on shared-memory and shared-disk systems.
    - However, some optimizations may be possible.

Interoperator Parallelism

- Pipelined parallelism
  - Consider a join of four relations
    - r1 $\bowtie$ r2 $\bowtie$ r3 $\bowtie$ r4
  - Set up a pipeline that computes the three joins in parallel
    - Let P1 be assigned the computation of temp1 = r1 $\bowtie$ r2
    - P2 be assigned the computation of temp2 = temp1 $\bowtie$ r3
    - and P3 be assigned the computation of temp2 $\bowtie$ r4
  - Each of these operations can execute in parallel, sending the result tuples it computes to the next operation even as it is computing further results
    - It requires a pipelineable join evaluation algorithm, e.g. (indexed) nested loops join

Factors Limiting Utility of Pipeline Parallelism

- Pipeline parallelism is useful since it avoids writing intermediate results to disk
- Useful with small number of processors, but does not scale up well with more processors.
  - Pipeline chains usually do not have sufficient length!
- Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g. aggregate and sort)
- Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others.

Intraoperation parallelism

- Parallelize the execution of each individual operation in the query
  - Supports high degree of parallelism

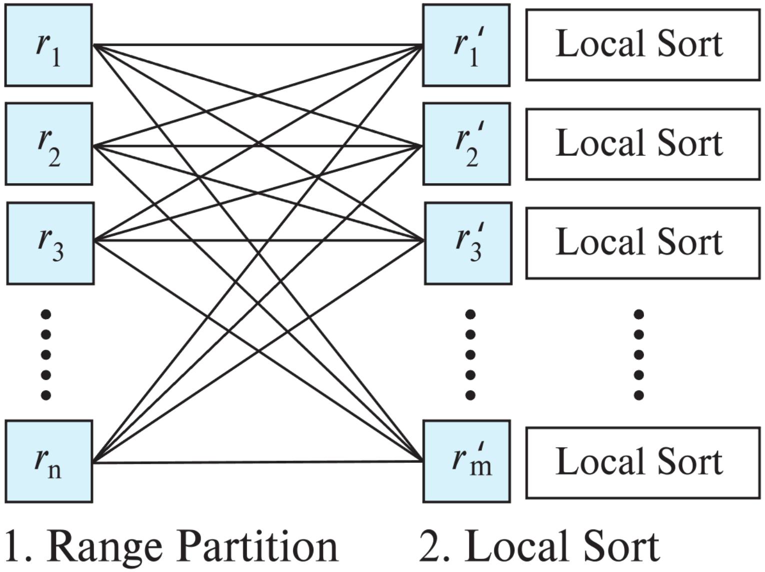
- Devise parallel algorithms for the various operators
  - Some operators are more amenable than others (e.g. in selection there is not much opportunity for parallelism).

- The basic idea of parallel algorithms for querying is:
  1. Start by partitioning the relation(s) in a convenient way
  2. Perform the operation in parallel in each of the partitions
  3. (If needed) do some nonparallel processing in the end

Range Partitioning Sort

Range-Partitioning Sort

- Choose nodes N_1, ..., N_m,
- Create range-partition vector with m-1 entries, on the sorting attributes
- Redistribute the relation using range partitioning
- Each node N_i sorts its partition of the relation locally.
  - Example of **data parallelism:** each node executes the same operation in parallel with other nodes, without any interaction with the others.
- Final merge operation is trivial: range-partitioning ensures that, if i < j, all key values in node N_i are all less than all key values in N_j.

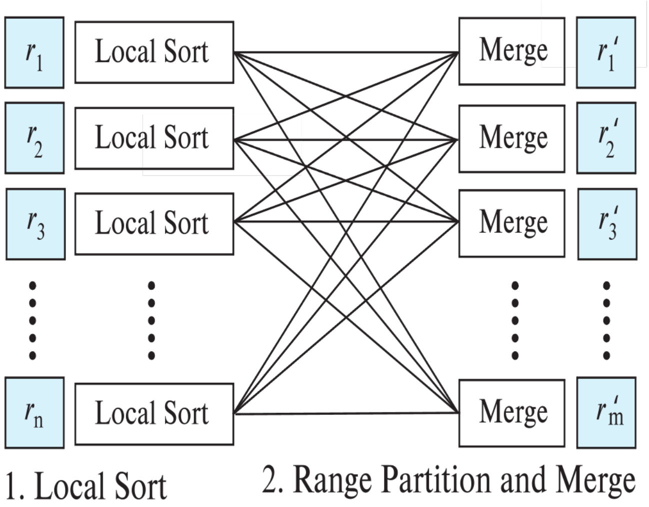


(a) Range Partitioning Sort

Parallel External Sort-Merge

Parallel External Sort-Merge

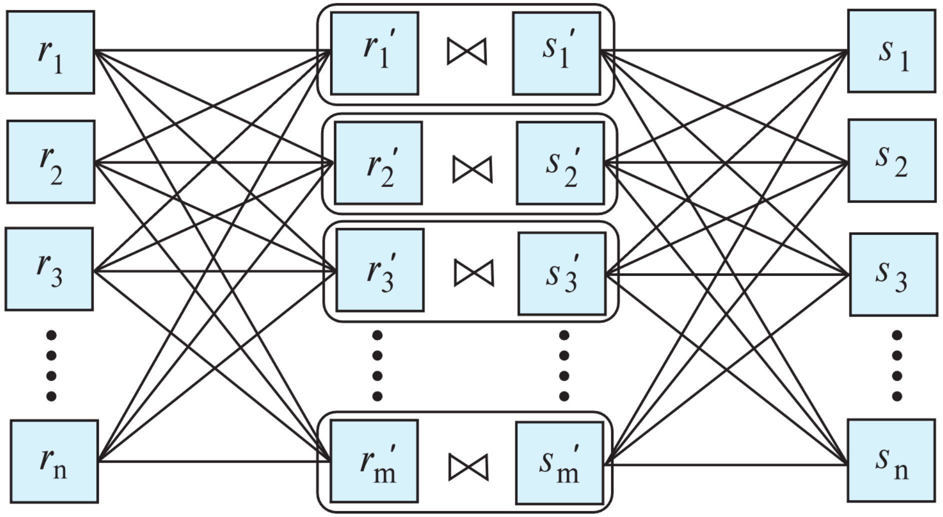
- Assume the relation has already been partitioned among nodes $N_1, ..., N_n$ (in whatever manner – not necessary by range).
- Each node N_i locally sorts the data (using local disk as required)
- The sorted runs on each node are then merged in parallel:
  - The sorted partitions at each node N_i are range-partitioned across the processors $N_1, ..., N_n$.
  - Each node Ni performs a merge on the streams as they are received, to get a single sorted run.
  - The sorted runs on nodes $N_1, ..., N_n$ are concatenated to get the result.
- Algorithm as described vulnerable to execution skew
  - all nodes send to node 1, then all nodes send data to node 2, …
  - Can be modified so each node sends data to all other nodes in parallel (block at a time)



(b) Parallel External Sort-Merge

Partitioned Parallel Join

Partition using range or hash partitioning, on join attributes



Step 1: Partition *r*   Step 2: Partition *s*

Step 3: Each node N_i computes $r_i' \bowtie s_i'$

Partitioned Parallel Join (Cont.)

- For equijoins and natural joins, it is possible to *partition* the two input relations across the processors and compute the join locally at each processor.
- Can use either *range partitioning* or *hash partitioning*.
- *r* and *s* must be partitioned on their join attributes, using the same range-partitioning vector or hash function.
- Join can be computed at each site using any of
  - Hash join, leading to **partitioned parallel hash join**
  - Merge join, leading to **partitioned parallel merge join**
  - Nested loops join, leading to **partitioned parallel nested-loops join** or **partitioned parallel index nested-loops join**

Partitioned Parallel Hash-Join

Parallelizing partitioned hash join:

- A hash function h_1 takes the join attribute value of each tuple in *s* and maps this tuple to one of the *n* nodes.
- As tuples of relation *s* are received at the destination nodes, they are partitioned further using another hash function, h_2, which is used to compute the hash-join locally.
- Repeat above for each tuple in *r*.
- Each node N_i executes the build and probe phases of the hash-join algorithm on the local partitions r_i and s_i of *r* and *s* to produce a partition of the final result of the hash-join.

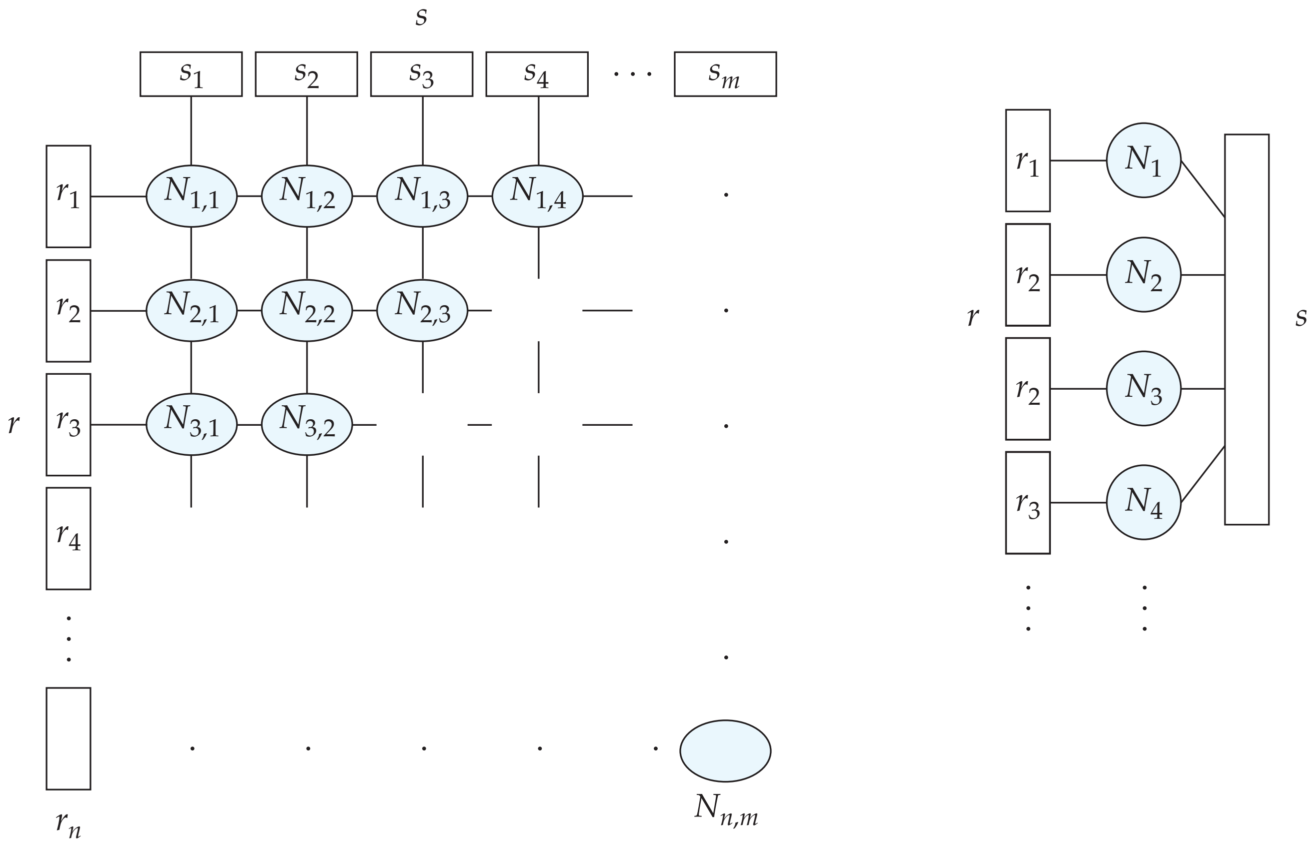
- Note: Hash-join optimizations can be applied to the parallel case
  - e.g., the hybrid hash-join algorithm can be used to cache some of the incoming tuples in memory and avoid the cost of writing them and reading them back in.

Fragment-and-Replicate Join

- Partitioning is not possible for some join conditions
  - E.g., non-equijoin conditions, such as r.A > s.B.
- For joins where partitioning is not applicable, parallelization can be accomplished by **fragment and replicate technique**
  - Fragment r into n relations, and s into m relations
  - Use n*m processors to compute joins between all partitions
  - This requires replicating the fragments
  - Make the union of the obtained joins
- Special case – **asymmetric fragment-and-replicate**:
  - One of the relations, say r, is partitioned; any partitioning technique can be used.
  - The other relation, s, is replicated across all the processors.
  - Node N_i then locally computes the join of r_i with all of s using any join technique.
  - Also referred to as **broadcast join**

Fragment-and-Replicate Joins

Symmetric and Asymmetric Fragment-and-Replicate Joins



Fragment-and-Replicate Join (Cont.)

- Both versions of fragment-and-replicate work with any join condition, since every tuple in *r* can be tested with every tuple in *s*.
- Usually has a higher cost than partitioning, since one of the relations (for asymmetric fragment-and-replicate) or both relations (for general fragment-and-replicate) must be replicated.
- Sometimes asymmetric fragment-and-replicate is preferable even though partitioning could be used.
  - E.g., if *s* is small and *r* is large, and *r* is already partitioned, it may be cheaper to replicate *s* across all nodes, rather than repartition *r* and *s* on the join attributes.

Other Relational Operations

Selection $\sigma_\theta(r)$

- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - If r is partitioned on a_i the selection is performed at a single node.
- If θ is of the form $l \leq a_i \leq u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
  - Selection is performed at each node whose partition overlaps with the specified range of values.
- In all other cases: the selection is performed in parallel at all the nodes.

Other Relational Operations (Cont.)

- **Duplicate elimination**
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash- partitioning) and perform duplicate elimination locally at each node.
- **Projection**
  - Projection without duplicate elimination can be performed as tuples are read from disk, in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- **Step 1**: Partition the relation on the grouping attributes
- **Step 2**: Compute the aggregate values locally at each node.
- **Optimization:** Can reduce cost of transferring tuples during partitioning by **partial aggregation** before partitioning
    - For distributive aggregate
    - Can be done as part of run generation
    - Consider the **sum** aggregation operation:
        - Perform aggregation operation at each node N_i on those tuples stored in its local disk
            - results in tuples with partial sums at each node.
        - Result of the local aggregation is partitioned on the grouping attributes, and the aggregation performed again at each node N_i to get the final result.
    - Fewer tuples need to be sent to other nodes during partitioning.

Query processing in Oracle

- Oracle implements most of the algorithms we've seen for query processing of the various operations
  - For selection
    - Full table scan (i.e. linear search algorithm)
    - Index scan (as described above in these slides)
    - Index fast full scan (transverse the whole table using the B+ index)
    - Cluster and hash cluster access (access data by using cluster key)
  - For join
    - (Block) nested loop join (possibly using indexes, when available)
    - Sort-merge join
    - Hash join

Query processing in Oracle (Cont.)

- The choice of the algorithm to be used in each operation is done by the query optimiser (we will see more on this later today)
- However, the user can force specific algorithms by providing hints when writing the query:

  select /*+ *here_comes_the_hint* ***/** …
- Many hints can be added. Here we name just a few.
  - Hints for forcing a specific algorithm (or index file) to be used in a selection
  - Hints for ordering the relation in a join (i.e. to choose the outer and the inner relations)
  - Hints for choosing an algorithm for join, and possibly an index file if appropriate;
  - Hints for forcing parallelism; Hints for guiding the optimiser
  - …
- See more at the Oracle's reference manual under the section on "Comments"
  - It may seem a strange place to have it, but there is where it is!

Some query hints in Oracle

- **full**(*table_name*)
  - Instructs the processor to use a full table scan
- **index**(*table_name index_name*)
  - The processor uses *index_name* to scan the table
- **no_index**(*table_name index_name*)
  - Forbids the use of *index_name* to scan the table
- **index_combine**(*table_name index_names*)
  - Builds a bitmap for the index files specified, and uses it in the scan of the table

- **ordered**
  - Instructs the processor to use the relations in a join in the exact order they appear in the select query
- **star_transformation**
  - The order chosen for join has the smaller relations first

Some (more) query hints in Oracle

- **use_nl**(*table_name table_name table_name …*)
  - Uses (block) nested loop for the join between the tables
- **use_nl_with_index**(*table_name index_name*)
  - The table is to be used as the inner relation of the join, in a nested loop algorithm, using the index file in the inner loop
- **use_merge**(*table_name table_name*)
  - Uses merge-join algorithm for the join between the tables
- **use_hash**(*table_name table_name*)
  - Uses hash join algorithm for the join between the tables

- **first_rows**(*n*)
  - Instructs the processor to use the fastest plan for providing the first n tuple of the result (as in pipelining)
- **all_rows**
  - The optimisation is made assuming taking into account the cost of obtaining all the tuples of the result

Parallel query processing in Oracle

- Oracle allows for parallelism in query processing, whenever multiple processors are available:

  alter session enable parallel query [**parallel_max servers** *n*]

- This enables the use of parallel algorithms for intraoperation and for parallelism in pipelining, using *n* processors
  - Other options are available for specifying parallelism groups, number of threads per cpu, etc.
  - Pipelining can be explicitly enforced by using **pipelined** PL/SQL table functions
- Parallelism can be specified to be used in all algorithms, depending on a specific table, with

  alter table table_name **parallel**

- There are also hints for imposing parallelism:
  - **parallel**(*table_name,n*)
    - Use *n* processors in partitioning parallel algorithms involving *table_name* in the query where the hint is given
    - If *n* is not given, the default value for the session is used

Chapter 16: Query Optimization

Sistemas de Bases de Dados 2020/21

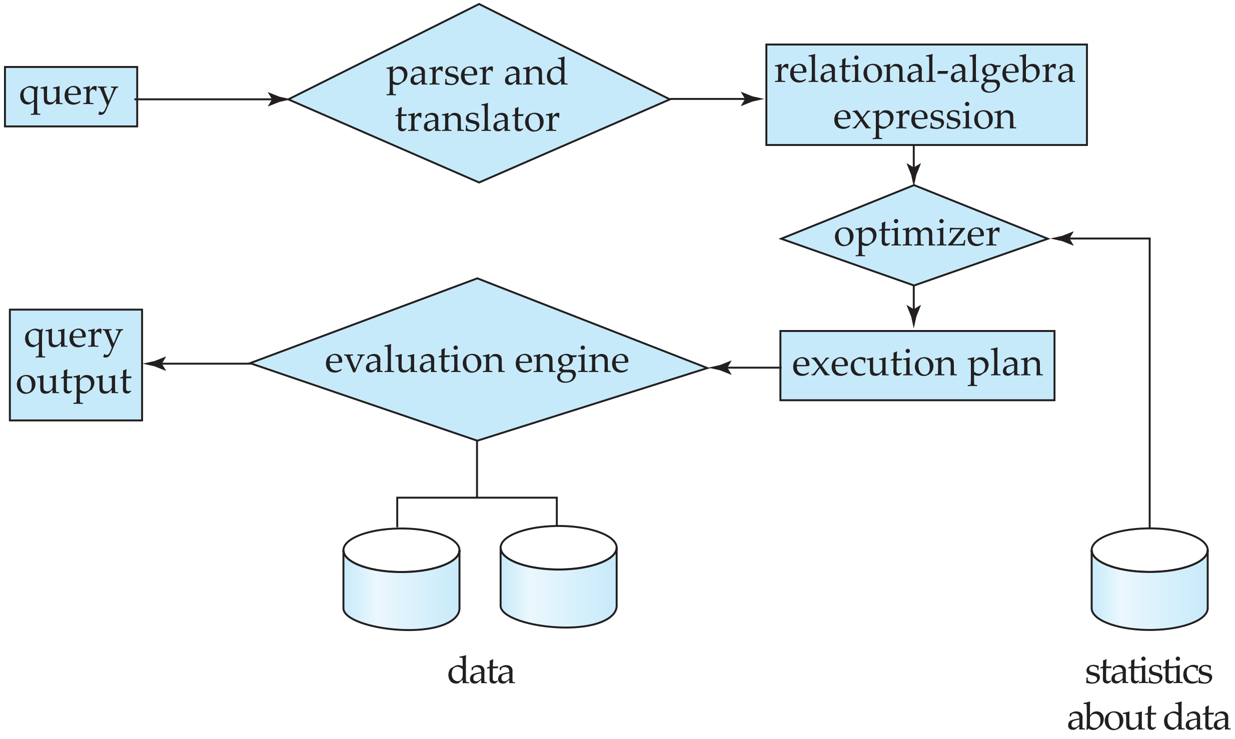
Capítulo refere-se a: Database System Concepts, 7th Ed

Outline

- Introduction
- Transformation of Relational Expressions
- Catalog Information for Cost Estimation
- Statistical Information for Cost Estimation
- Cost-based optimization
- Dynamic Programming for Choosing Evaluation Plans
- Materialized views

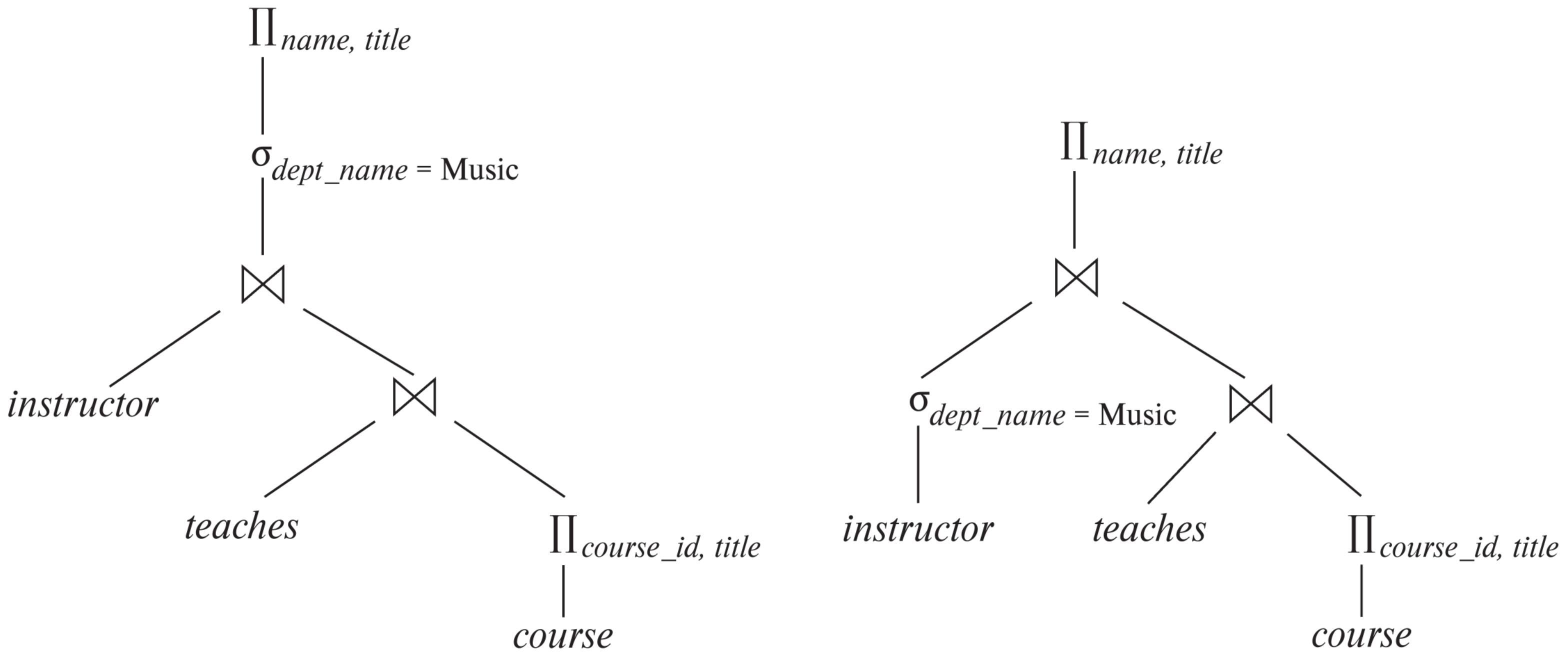
Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Introduction

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

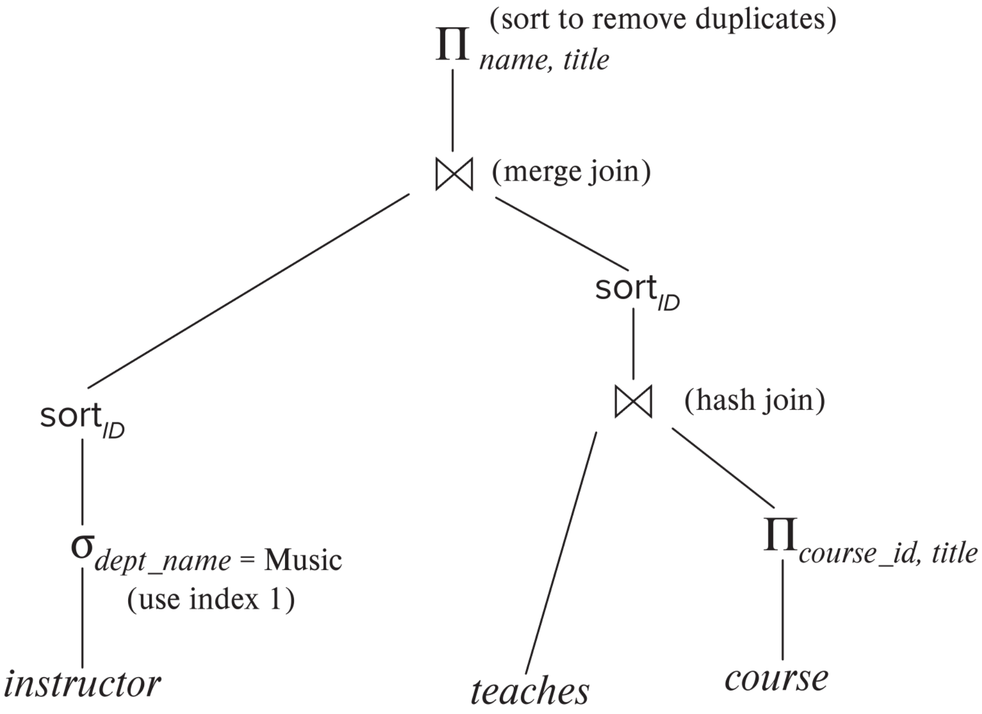

$\Pi_{name,\ title}$
$\sigma_{dept\_name\ =\ \text{Music}}$
$\bowtie$
instructor
$\bowtie$
teaches
$\Pi_{course\_id,\ title}$
course
$\Pi_{name,\ title}$
$\bowtie$
$\sigma_{dept\_name\ =\ \text{Music}}$
$\bowtie$
instructor
teaches
$\Pi_{course\_id,\ title}$
course

(a) Initial expression tree

(b) Transformed expression tree

Introduction (Cont.)

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



- We've seen how to view the plans in Oracle

Introduction (Cont.)

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

Generating Equivalent Expressions

Sistemas de Bases de Dados 2020/21

Capítulo refere-se a: Database System Concepts, 7th Ed

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
  - we don’t care if they generate different results on databases that violate integrity constraints
- 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
  - One can replace expression of the 1st form by the 2nd, or vice versa

Equivalence Rules

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

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

2. Selection operations are commutative.

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

3. In a sequence of projection operations, only the most restrictive is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) \equiv \Pi_{L_1}(E)$$

where $L_1 \subseteq L_2 \ldots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins.

   a. $\sigma_\theta(E_1 \text{ x } E_2) \equiv E_1 \bowtie_\theta E_2$

   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Equivalence Rules (Cont.)

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

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

6. (a) Natural join operations are associative:

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

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

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:

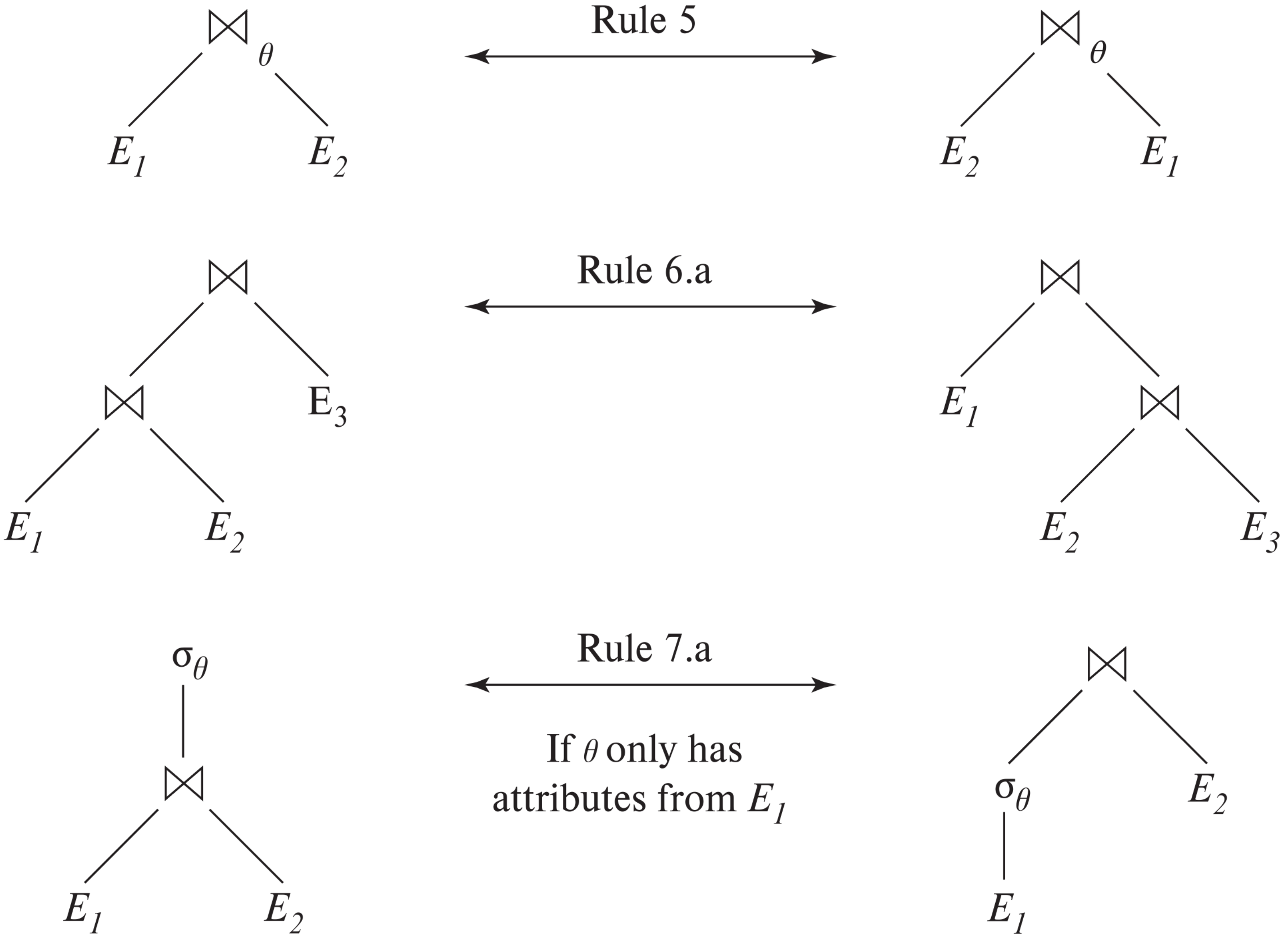
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined:

   $$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2:

   $$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

Pictorial Depiction of Equivalence Rules



Rule 5

Rule 6.a

Rule 7.a

If θ only has attributes from E_1

Equivalence Rules (Cont.)

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

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

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

   (b) In general, 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 $L_1 \cup L_2$.

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

Similar equivalences hold for outerjoin operations: ⟕, ⟖, and ⟗

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1$$
$$E_1 \cap E_2 \equiv E_2 \cap E_1$$

(set difference is not commutative).

10. Set union and intersection are associative.

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

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

a. $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$

b. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta(E_2)$

c. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta(E_2)$

d. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$

e. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2$

preceding equivalence does not hold for $\cup$

12. The projection operation distributes over union

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

Equivalence Rules (Cont.)

13. Selection distributes over aggregation as below

    $\sigma_\theta({}_G\gamma_A(E)) \quad \equiv \quad {}_G\gamma_A(\sigma_\theta(E))$

    provided θ only involves attributes in G

14. a. Full outerjoin is commutative:

    $E_1 ⟗ E_2 \quad \equiv \quad E_2 ⟗ E_1$

    b. Left and right outerjoin are not commutative, but:

    $E_1 ⟕ E_2 \quad \equiv \quad E_2 ⟖ E_1$

15. Selection distributes over left and right outerjoins as below, provided θ_1 only involves attributes of E_1

    a. $\sigma_{\theta_1} (E_1 ⟕_\theta E_2) \quad \equiv \quad (\sigma_{\theta_1} (E_1)) ⟕_\theta E_2$

    b. $\sigma_{\theta_1} (E_1 ⟖_\theta E_2) \quad \equiv \quad E_2 ⟕_\theta (\sigma_{\theta_1} (E_1))$

16. Outerjoins can be replaced by inner joins under some conditions

    a. $\sigma_{\theta_1} (E_1 ⟕_\theta E_2) \quad \equiv \quad \sigma_{\theta_1} (E_1 \bowtie_\theta E_2)$

    b. $\sigma_{\theta_1} (E_1 ⟖_\theta E_1) \quad \equiv \quad \sigma_{\theta_1} (E_1 \bowtie_\theta E_2)$

    provided θ_1 is null rejecting on E_2

Equivalence Rules (Cont.)

Note that several equivalences that hold for joins do not hold for outerjoins

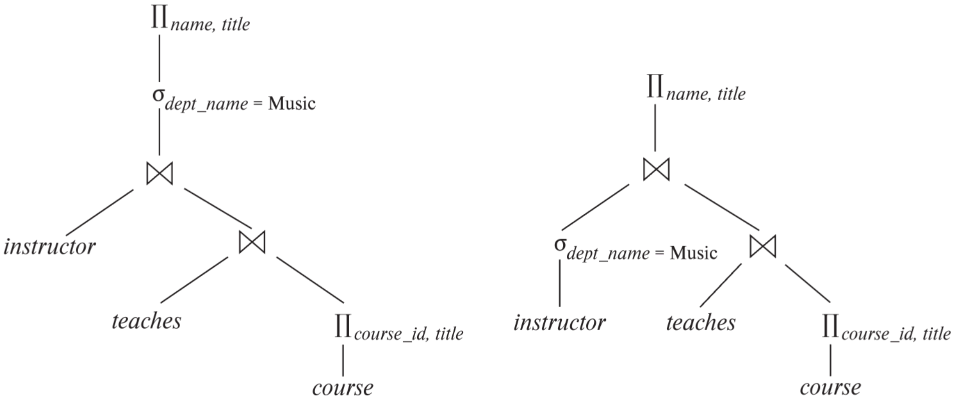
- $\sigma_{year=2017}$(*instructor* ⟕ *teaches*) $\not\equiv$ $\sigma_{year=2017}$(*instructor* ⋈ *teaches*)
- Outerjoins are not associative

  (r ⟕ s) ⟕ t $\not\equiv$ r ⟕ (s ⟕ t)

  - e.g. with r(A,B) = {(1,1)}, s(B,C) = { (1,1)}, t(A,C) = { }

Transformation Example: Pushing Selections

- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\ 'Music'}$ $(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\ (course))))$
- Transformation using rule 7a.
  - $\Pi_{name,\ title}((\sigma_{dept\_name=\ 'Music'}(instructor)) \bowtie$ $(teaches \bowtie \Pi_{course\_id,\ title}\ (course)))$
  - It is usually a good idea since performing the selection as early as possible reduces the size of the relation to be joined.



(a) Initial expression tree (b) Transformed expression tree

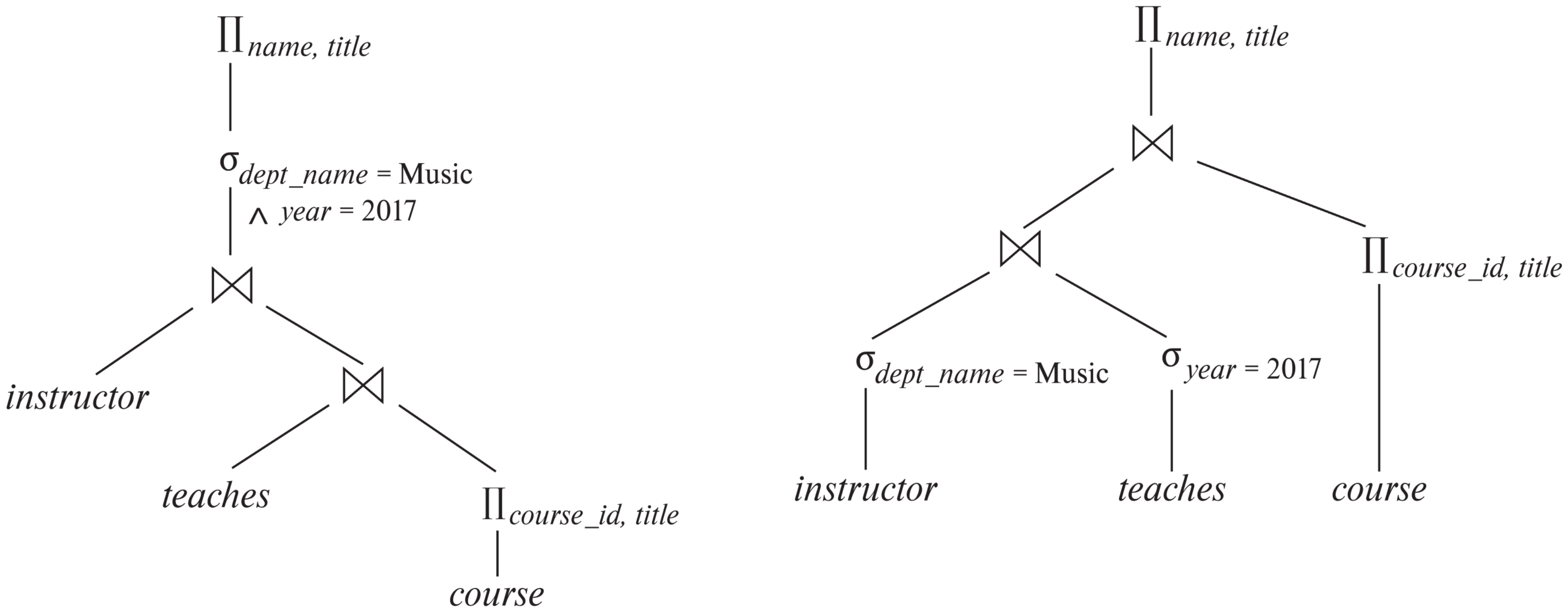
Example with Multiple Transformations

- Query: Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught
  - $\Pi_{name,\ title}(\sigma_{dept\_name= \text{"Music"} \wedge year = 2017}$
    $(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\ (course))))$
- Transformation using join associatively (Rule 6a):
  - $\Pi_{name,\ title}(\sigma_{dept\_name= \text{“Music”} \wedge year = 2017}$
    $((instructor \bowtie teaches) \bowtie \Pi_{course\_id,\ title}\ (course)))$
- Second form provides an opportunity to apply the “perform selections early” rule, resulting in the subexpression

  $\sigma_{dept\_name\ =\ \text{“Music”}}\ (instructor) \bowtie \sigma_{\ year\ =\ 2017}\ (teaches)$

- A sequence of transformations can be useful!

Multiple Transformations (Cont.)


$\Pi_{name,\ title}$
$\sigma_{dept\_name\ =\ \text{Music}\ \wedge\ year\ =\ 2017}$
$\bowtie$
instructor
$\bowtie$
teaches
$\Pi_{course\_id,\ title}$
course
$\Pi_{name,\ title}$
$\bowtie$
$\bowtie$
$\Pi_{course\_id,\ title}$
$\sigma_{dept\_name\ =\ \text{Music}}$
$\sigma_{year\ =\ 2017}$
instructor
teaches
course

(a) Initial expression tree

(b) Tree after multiple transformations

Transformation Example: Pushing Projections

- Consider: $\Pi_{name,\ title}(\sigma_{dept\_name=\text{ “Music”}}(instructor) \bowtie teaches)$
  $\bowtie \ \Pi_{course\_id,\ title}(course))))$
- When we compute

  $$\sigma_{dept\_name\,=\,\text{“Music”}}(instructor \bowtie teaches)$$

  we obtain a relation whose schema is:
  (*ID, name, dept_name, salary, course_id, sec_id, semester, year)*
- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

  $$\Pi_{name,\ title}(\Pi_{name,\ course\_id}(\\ \sigma_{dept\_name=\text{ “Music”}}(instructor) \bowtie teaches)) \\ \bowtie \quad \Pi_{course\_id,\ title}(course))))$$

- Performing the projection as early as possible reduces the size of the relation to be joined.
  - Important also for the size of the intermediate relation, e.g. in materialisation

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) $\bowtie$
- 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 the computed and stored temporary relation (in case no pipelining is used) is smaller

Join Ordering Example (Cont.)

- Consider the expression

  $\Pi_{name,\ title}((\sigma_{dept\_name=\ \text{“Music”}}\ (instructor) \bowtie teaches) \bowtie\ \Pi_{course\_id,\ title}\ (course))$

- Could compute $teaches \bowtie \Pi_{course\_id,\ title}\ (course)$ first, and join result with

  $\sigma_{dept\_name=\ \text{“Music”}}\ (instructor)$

  but the result of the first join is likely to be a large relation.

- Only a small fraction of the university’s instructors are likely to be from the Music department
  - it is better to compute

    $\sigma_{dept\_name=\ \text{“Music”}}\ (instructor) \bowtie teaches$

    first.

Enumeration of Equivalent Expressions

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

    Until no new equivalent expressions are generated above
- The above approach is very expensive in space and time
  - Two approaches
    - Optimized plan generation based on transformation rules
    - Special case approach for queries with only selections, projections and joins

Cost Estimation

- Cost of each operator computer as described in Chapter 15
  - Need statistics of input relations
    - E.g., number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
  - To do so, we require additional statistics
    - E.g., number of distinct values for an attribute

- More on cost estimation next week

Choice of Evaluation Plans

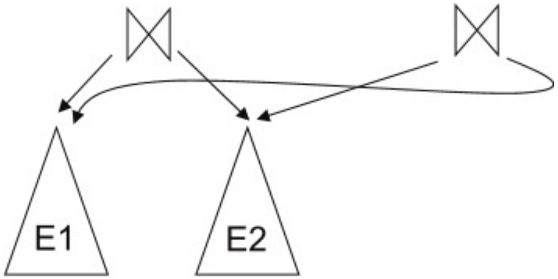
- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining
- 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 $r_1 \bowtie r_2 \bowtie \ . \ . \ . \bowtie r_n$.
- There are $(2(n - 1))!/(n - 1)!$ different join orders for above expression. With $n = 7$, the number is 665 280; 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 $\{r_1, r_2, . . . r_n\}$ is computed only once and stored for future use.

Implementing Transformation Based Optimization

- Space requirements reduced by sharing common sub-expressions:
  - when E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared using pointers
    - E.g., when applying join commutativity



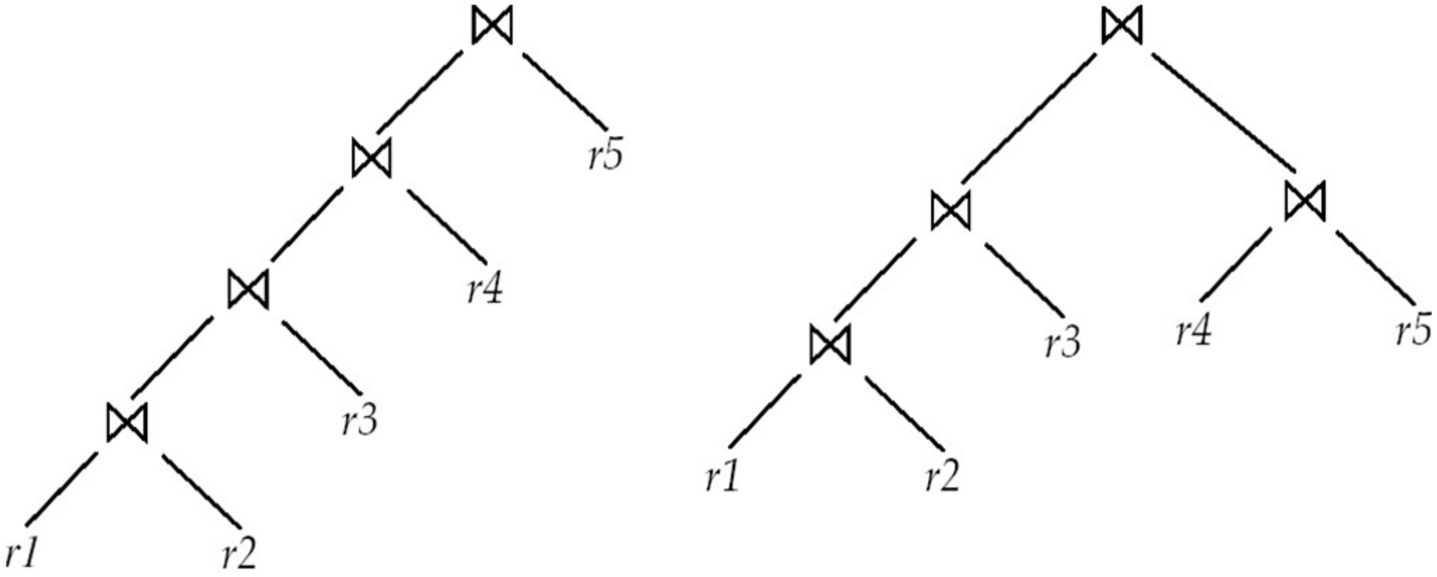
  - Same sub-expression may get generated multiple times
    - Detect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
  - Dynamic programming
    - We will study only the special case of dynamic programming for join order optimization

Dynamic Programming in Optimization

- To find the best join tree for a set of *n* relations:
  - 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 $2^n - 2$ alternatives.
  - Base case for recursion: single relation 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

Left Deep Join Trees

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



(a) Left-deep join tree

(b) Non-left-deep join tree

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- 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 $S1$ of S such that $S1 \neq S$"
    - 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 $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- 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 specific sort order of tuples that could make a later operation (join/group by/order by) cheaper
  - 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
- 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, the 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) improves 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.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

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 is used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimization
  - E.g., nested subqueries

Join Minimisation

```
select r.A, r.B
from r, s
where r.B = s.B
```

- Check if join with s is redundant. If so, drop it!
  - E.g. join condition is in *foreign key* from r to s, no selection on s
  - Other sufficient conditions possible

```
select r.A, s1.B
from r, s as s1, s as s2
where r.B=s1.B and r.B = s2.B and s1.A < 10 and s2.A < 20
```

    - join with s2 is redundant and can be dropped (along with selection on s2)

- There are many special cases where joins can be dropped!

Optimising Nested Subqueries

- Nested query example:

  ```
  select instructor.name
  from instructor
  where exists (select *
              from teaches
              where teaches.instructor_id=
                                   instructor.instructor_id)
  ```

- SQL conceptually treats nested subqueries in the **where** clause as functions that take parameters and return a single value or set of values
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**
- Conceptually, a nested subquery is executed once for each tuple in the cross-product generated by the outer level **from** clause
  - Such evaluation is called **correlated evaluation**
  - Note: other conditions in **where** clause may be used to compute a join (instead of a cross-product) before executing the nested subquery

Structure of Query Optimizers (Cont.)

- 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 the entire query and do not depend on block structure
    - **Optimization cost budget** to stop optimization early (if cost of plan is less than cost of optimization)
    - **Plan caching** to reuse previously computed plan if the query is resubmitted
        - Even with different constants in query
- 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