Chapter 7

Query Processing

Introduction

Query processing refers to the range of activities involved in extracting data from database.

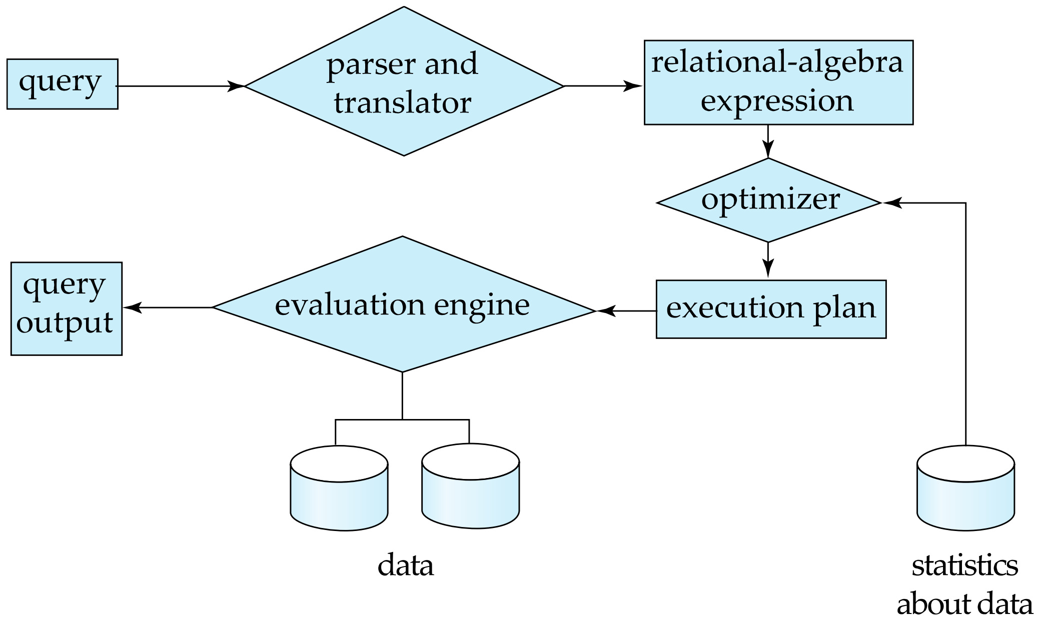
The activities includes:

- ➢ translation of queries in high-level database languages into expressions that can be used at the physical level of the file system
- ➢ a variety of query-optimizing transformations
- ➢ actual evaluation of queries

The steps involved in processing a query appear in Figure below.

The basic steps are:

1. Parsing and translation
2. Optimization
3. Evaluation



1. Parsing and translation

- ✓ In this step query is translated into its internal form. This translation process is similar to the work performed by the parser of the compiler.
- ✓ In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on.
- ✓ The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression.

2. Optimization

A relational algebra expression may have many equivalent expressions.

As an illustration, consider the query:
select *salary*
from *instructor*
where *salary* < 75000;

This query can be translated into either of the following relational-algebra expressions

- $\sigma_{salary<75000}(\Pi_{salary}(instructor))$

  is equivalent to

- $\Pi_{salary}(\sigma_{salary<75000}(instructor))$

Relational algebra expression annotate with instructions specifying how to evaluate each operation which may state the algorithm to be used for a specific operation, or the particular index or indices to use.

A relational algebra operation annotated with instructions on how to evaluate it is called an **evaluation primitive.**

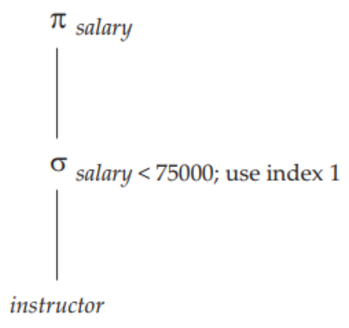


Figure: A query-evaluation plan.

A sequence of primitive operations that can be used to evaluate a query is a query-evaluation plan. The process of choosing the evaluation plan with the lowest cost amongst all equivalent evaluation plans is known as query optimization. The cost is estimated using the statistical information from the database catalog. The different statistical information is number of tuples in each relation, tuple size etc.

3. Evaluation

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

Measures of query cost

There are multiple possible evaluation plans for a query, and it is important to be able to compare the alternatives in terms of their (estimated) cost, and choose the best plan. To do so, we must estimate the cost of individual operations, and combine them to get the cost of a query evaluation plan.
The cost of query evaluation can be measured in terms of a number of different resources, including:

- disk accesses
- CPU time to execute a query
- a distributed or parallel database system, the cost of communication

In large database systems, the cost to access data from disk is usually the most important cost, since disk accesses are slow compared to in-memory operations. Moreover, CPU speeds have been improving much faster than have disk speeds. Thus, it is likely that the time spent in disk activity will continue to dominate the total time to execute a query.

Although real-life query optimizers do take CPU costs into account, for simplicity here we ignore CPU costs and **use only disk-access costs to measure the cost of a query-evaluation plan**.

- We use the **number of block transfers** *from disk and the* **number of disk seeks** to estimate the cost of query evaluation plan.
- If a disk subsystem takes an average of t_T seconds to transfer the block of data and has average block access time of t_S seconds then an operation that transfers b blocks and performs S seeks would take $b * t_T + S * t_S$ seconds.
- The values of t_T and t_S must be calibrated for the disk system used, but typical values for high end disks would be $t_S = 4$ milliseconds and $t_T = 0.1$ milliseconds assuming a 4 kilobyte size and a transfer rate of 40 megabytes per seconds;

Query optimization

Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, especially if the query is complex.

Let us consider the following relation:

depositor(<u>customer_name</u>,account_number)
account(<u>account_number</u>,branch_name,balance)
branch(<u>branch_name</u>,branch_city,assets)
Now, Consider the following relational-algebra expression, for the query "**Find the names of all customers who have an account at some branch located in kathmandu"**

$\Pi_{customer\_name}(\sigma_{branch\_city= \text{"kathmandu"}}$ (depositer $\bowtie$ (account $\bowtie$branch)))

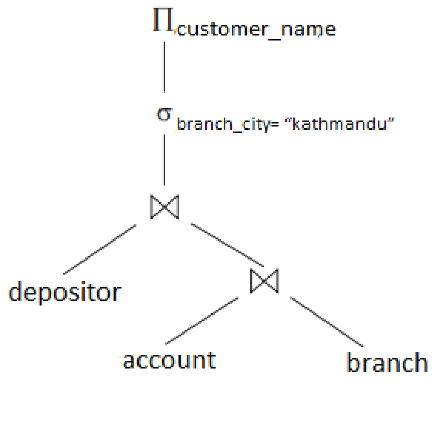
The above expression constructs a large intermediate relation. However we are interested only few records of this relation.(i.e. branch city is kathmandu)

By reducing the number of tuples in branch relation that we need access we reduce the size of intermediate result.

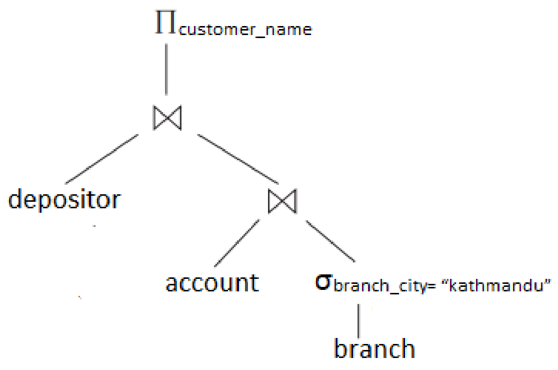
Our query is now represented by:

$\Pi_{customer\_name}($ depositer $\bowtie$ (account $\bowtie$ $\sigma_{branch\_city= \text{"kathmandu"}}$ (branch)))

which is equivalent to previous relational algebra expression, but which generates the smaller intermediate relations. Figure below depicts the initial and transformed expressions.



a) Initial expression tree



b) Transformed expression tree

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

For a given a **relational-algebra expression**, it is the job of the query optimizer to come up with a query-evaluation plan that computes the same result as the given expression, and is the least-costly way of generating the result .

To find the least-costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression, and to choose the least-costly one.

Generation of query-evaluation plans involves three steps:

(1) generating expressions that are logically equivalent to the given expression

(2) annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans

(3) estimating the cost of each evaluation plan, and choosing the one whose estimated cost is the least.

- To implement the first step, the query optimizer must generate expressions equivalent to a given expression by using equivalence rules.
- We must estimate the statistics of results of each operation in a query plan.

For example:

- **catalog information (***The database-system catalog stores the following statistical information about database relations such as the number of tuples in a relation r, number of blocks containing the relation r , the size of a tuple of relation r in bytes)*
- **size of joins estimation**
- **selection size estimation**

Using this statistics cost formulae allows us to estimate the cost of each individual operations. The individual cost is combined to determine the estimated cost of evaluating the given relational algebra expressions.

- Now we can choose query evaluation plan based on the estimated cost of the plans with is likely to be the least costly one or not more much more costly than it.

Equivalence of expressions

- ✓ Any two relational expressions are said to be equivalent if resulting relation generates the same set of tuples.
- ✓ When two expressions are equivalent we can use them interchangeably i.e. we can use either of the expressions which gives the better performance.

Equivalence rule

- ✓ An equivalence rule says that expressions of two forms are equivalent.
- ✓ We can replace an expression of the first form by an expression of the second form, or vice versa i.e., we can replace an expression of the second form by an expression of the first form, since the two expressions generate the same result on any valid database.
- ✓ The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.
- ✓ We now list a number of general equivalence rules on relational-algebra expressions. We use θ1, θ2, θ3, and so on to denote predicates, L1, L2, L3, and so on to denote lists of attributes, and E1, E2, E3, and so on to denote relational-algebra expressions. A relation name r is simply a special case of a relational-algebra expression, and can be used wherever E appears.

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. Only the final operations in a sequence of projection operations is needed, the others can be omitted

   $\prod_{L1}(\prod_{L2}(\ldots(\prod_{Ln}(E))\ldots)) \equiv \prod_{L1}(E)$
   where $L_1 \subseteq L_2 \ldots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins:
   a. $\sigma_\theta (E_1 \times E_2) \equiv E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1} (E_1 \bowtie_{\theta 2} E_2) \equiv E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

5. Theta join operations are commutative:

   $E1 \bowtie_\theta E2 \equiv E2 \bowtie_\theta E1$

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.

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.

$\sigma_{\theta 1 \wedge \theta 2} (E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$

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

(a) if θ involves only attributes from $L_1 \cup L_2$:
$\prod_{L1 \cup L2}(E_1 \bowtie_\theta E_2) \equiv \prod_{L1}(E_1) \bowtie_\theta \prod_{L2}(E_2)$

(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 $L_1 \cup L_2$.

$\prod_{L1 \cup L2}(E_1 \bowtie_\theta E_2) \equiv \prod_{L1 \cup L2}(\prod_{L1 \cup L3}(E_1) \bowtie_\theta \prod_{L2 \cup L4}(E_2))$

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.

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

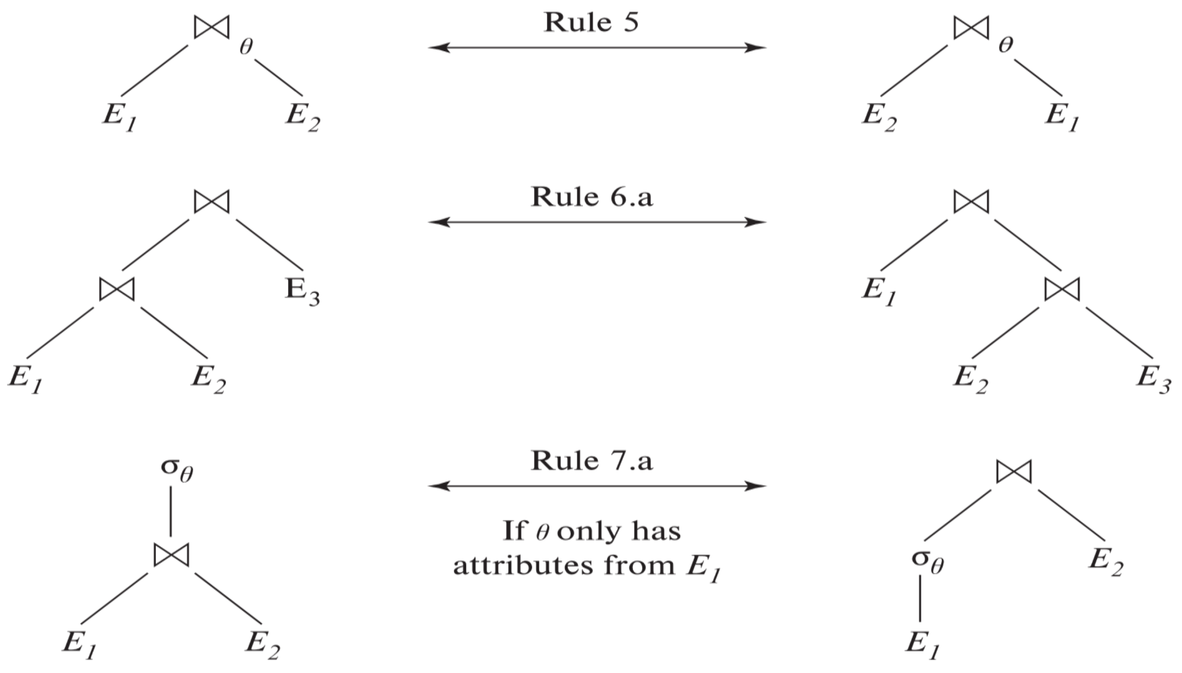
11. The selection operation distributes over the union, intersection, and set-difference operations.

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

12. The projection operation distributes over the union operation.

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

Pictorial depiction of Equivalence rules



Example of transformations:

Let us consider the following relation:

depositor(<u>customer_name</u>,account_number)
account(<u>account_number</u>,branch_name,balance)
branch(<u>branch_name</u>,branch_city,assets)

Example 1:

Query:

Find the name of all customers who have an account at some branch located in Kathmandu.

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

Transformation using rule 7a.

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

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

Example 2:

Query:

Find the name of all customers with an account at a kathmandu city whose account balance is over 1000.

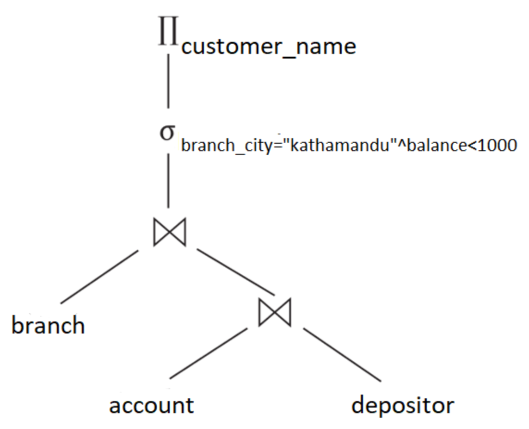
$\Pi_{customer\_name}(\sigma_{branch\_city= \text{“kathmandu”} \wedge\ balance>1000} (branch \bowtie (account \bowtie depositor)))$

Transformation using join associativity (Rule 6a)

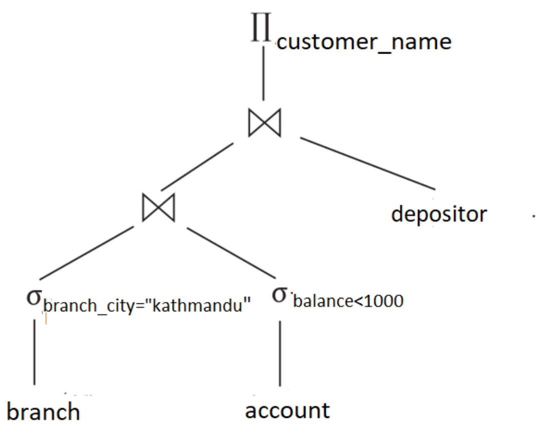
$\Pi_{customer\_name}((\sigma_{branch\_city= \text{“kathmandu”} \wedge\ balance>1000} (branch \bowtie account)) \bowtie depositor)$

second form provides an opportunity to apply the “perform selection early” rule, resulting in the subexpression

$\sigma_{branch\_city= \text{“kathmandu”}} (branch) \bowtie \sigma_{balance>1000} (account)$



a) Initial expression tree



b) Tree after multiple transformations

Evaluation of Expression

Expression cannot exist as a single operation instead typical query combines them.

For example:

```
select customer_name
 from account natural  join customer
 where balance <30000;
```

This example consist of selection, natural join, and a projection.

To evaluate an expression that carries multiple operations in it, computation of each operation is performed one by one.

In the query processing, two methods are used for evaluating an expression that carries multiple operations. These methods are:

1. Materialization
2. Pipelining

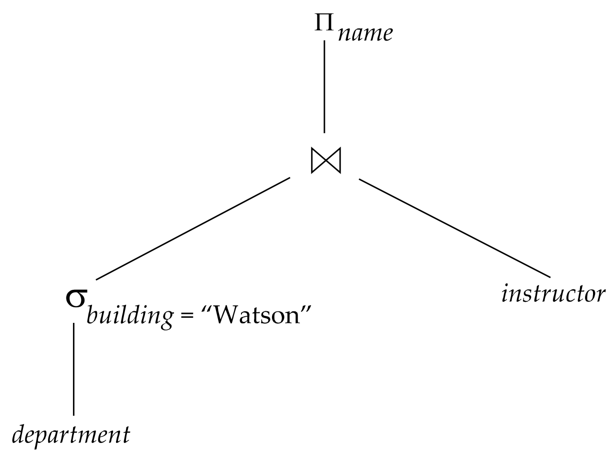
1. Materialization

In materialization, an expression is evaluated one operation at a time in an appropriate order. The result of each evaluation is **materialized** in a temporary relation for subsequent use.

A disadvantage to this approach is the need to construct the temporary relations, which (unless they are small) must be written to disk.

Consider the expression:

$\Pi_{name}((\sigma_{building= \text{“watson”}}(department)) \bowtie instructor)$



If we apply the materialization approach, we start from the lowest-level operations in the expression (at the bottom of the tree). In our example, there is only one such operation: the selection operation on *department*. The inputs to the lowest-level operations are relations in the database. We execute these operations and we store the results in temporary relations. We can use these temporary relations to execute the operations at the next level up in the tree, where the inputs now are either temporary relations or relations stored in the database.

In our example, the inputs to the join are the *instructor* relation and the temporary relation created by the selection on *department*. The join can now be evaluated, creating another temporary relation.

2. Pipelining

- ✓ **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- ✓ For example, as in previous expression tree, don't store result of instead, pass tuples directly to the join. Similarly, don't store result of join, pass tuples directly to projection.

  Creating a pipeline of operations can provide two benefits:

  1. It eliminates the cost of reading and writing temporary relations, reducing the cost of query evaluation.
  2. It can start generating query results quickly, if the root operator of a query evaluation plan is combined in a pipeline with its inputs. This can be quite useful if the results are displayed to a user as they are generated, since otherwise there may be a long delay before the user sees any query results.

Pipelines can be executed in either of two ways:

1. **In a demand-driven pipeline or lazy evaluation**, the system makes repeated requests for tuples from the operation at the top of the pipeline. Each time that an operation receives a request for tuples, it computes the next tuple (or tuples) to be returned, and then returns that tuple.

2. **In a producer-driven or eager pipelining,** operations do not wait for requests to produce tuples, but instead generate the tuples eagerly.

Choice of evaluation plan

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

For a given an evaluation plan, we can estimate its cost using statistics estimated by the techniques coupled with cost estimates for various algorithms and evaluation methods.

Approaches to choose the best evaluation plan for a query are as follows:

- ➢ Search all the plans and choose the best plan on the cost based fashions (cost based optimizer)
- ➢ Use heuristics to choose the plan

Cost based optimizer

- ✓ A cost-based optimizer explores the space of all query-evaluation plans that are equivalent to the given query, and chooses the one with the least estimated cost.
- ✓ Equivalence rules can be used to generate equivalent plans.
- ✓ Exploring the space of all possible plans may be too expensive for complex queries.

Consider the problem of choosing the optimal join order for such a query. For a complex join query, the number of different query plans that are equivalent to the query can be large.

As an illustration, consider the expression:

$$r_1 \bowtie r_2 \bowtie \ . \ . \ . \bowtie r_n$$

where the joins are expressed without any ordering. With n = 3, there are 12 different join orderings:

r1 ⋈ (r2 ⋈ r3)	r1 ⋈ (r3 ⋈ r2)	(r2 ⋈ r3) ⋈ r1	(r3 ⋈ r2) ⋈ r1
r2 ⋈ (r1 ⋈ r3)	r2 ⋈ (r3 ⋈r1)	(r1 ⋈ r3) ⋈r2	r3 ⋈ (r1 ⋈ r2)
r3 ⋈ (r2 ⋈ r1)	(r1 ⋈ r2) ⋈ r3	(r3 ⋈ r1) ⋈ r2	(r2 ⋈ r1) ⋈ r3

In general, with *n* relations, there are $(2(n - 1))!/(n - 1)!$ different join orders. For joins involving small numbers of relations, this number is acceptable; for example, with *n* = 5, the number is 1680. However, as *n* increases, this number rises quickly.

- ➢ With *n* = 7, the number is 665,280
- ➢ with *n* = 10, the number is greater than 17.6 billion!

It is not necessary to generate all the expressions equivalent to a given
expression. For example, suppose we want to find the best join order of the form:
$(r1 \bowtie r2 \bowtie r3) \bowtie r4 \bowtie r5$
which represents all join orders where *r*1,*r*2, and *r*3 are joined first (in some order),
and the result is joined (in some order) with *r*4 and *r*5. There are 12 different join

orders for computing $r1 \bowtie r2 \bowtie r3$, and 12 orders for computing the join of this result with $r4$ and $r5$. Thus, there appear to be 144 join orders to examine. However, once we have found the best join order for the subset of relations $\{r1, r2, r3\}$, we can use that order for further joins with $r4$ and $r5$, and can ignore all costlier join orders of $r1 \bowtie r2 \bowtie r3$. Thus, instead of 144 choices to examine, we need to examine only 12 + 12 choices.
Using this idea, we can develop a *dynamic-programming* algorithm for finding optimal join orders. Dynamic-programming algorithms store results of computations and reuse them, a procedure that can reduce execution time greatly.

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 $O(2^n)$

Heuristics in Optimization

Cost-based optimization is expensive, even with dynamic programming. Systems may use heuristics to reduce the number of choices that must be made in a cost-based fashion. Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:

- Perform selection early (reduces the number of tuples)
- Perform projection early (reduces the number of attributes)
- Perform most restrictive selection and join operations (i.e., with smallest result size) before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.