COSC 404
Database System Implementation

Query Optimization

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Query Optimization Overview

The query processor performs four main tasks:

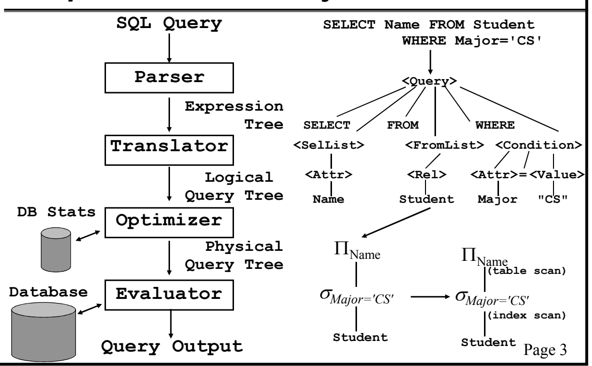
1) Verifies the correctness of an SQL statement

2) Converts the SQL statement into relational algebra

3) Performs heuristic and cost-based optimization to build the more efficient execution plan

4) Executes the plan and returns the results

Components of a Query Processor



Query Processor Components The Parser

The role of the parser is to convert an SQL statement represented as a string of characters into a parse tree.

A ***parse tree*** consists of nodes, and each node is either an:

- ***Atom*** - lexical elements such as words (`WHERE`), attribute or relation names, constants, operator symbols, etc.
- ***Syntactic category*** - are names for query subparts.
  - E.g. <SFW> represents a query in select-from-where form.

Nodes that are atoms have no children. Nodes that correspond to categories have children based on one of the rules of the grammar for the language.

A Simple SQL Grammar

A ***grammar*** is a set of rules dictating the structure of the language. It exactly specifies what strings correspond to the language and what ones do not.

- Compilers are used to parse grammars into parse trees.
  - Same process for SQL as programming languages, but somewhat simpler because the grammar for SQL is smaller.

Our simple SQL grammar will only allow queries in the form of `SELECT-FROM-WHERE`.

- We will not support grouping, ordering, or `SELECT DISTINCT`.
- We will support lists of attributes in the `SELECT` clause, lists of relations in the `FROM` clause, and conditions in the `WHERE` clause.

Simple SQL Grammar

```
<Query> ::= <SFW>
<Query> ::= ( <Query> )

<SFW> ::= SELECT <SelList> FROM <FromList> WHERE
                                          <Condition>
<SelList> ::= <Attr>
<SelList> ::= <Attr> , <SelList>

<FromList> ::= <Rel>
<FromList> ::= <Rel> , <FromList>

<Condition> ::= <Condition> AND <Condition>
<Condition> ::= <Tuple> IN <Query>
<Condition> ::= <Attr> = <Attr>
<Condition> ::= <Attr> LIKE <Value>
<Condition> ::= <Attr> = <Value>
<Tuple> ::= <Attr> // Tuple may be 1 attribute
```

A Simple SQL Grammar Discussion

The syntactic categories of `<Attr>`, `<Rel>`, and `<Value>` are special because they are not defined by the rules of the grammar.

- `<Attr>` - must be a string of characters that matches an attribute name in the database schema.
- `<Rel>` - must be a character string that matches a relation name in the database schema.
- `<Value>` - is some quoted string that is a legal SQL pattern or a valid numerical value.

Query Example Database

Student(Id,Name,Major,Year)
Department(Code,DeptName,Location)

Student Relation

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
11589	T. Allen	BA	2
15973	M. Smith	CS	3
29579	B. Zimmer	BS	1
34596	T. Atkins	ME	4
75623	J. Wong	BA	3
84920	S. Allen	CS	4
96256	P. Wright	ME	2

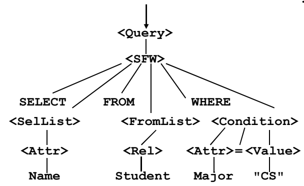
Department Relation

Code	DeptName	Location
BA	Bachelor of Arts	English Building
BS	Bachelor of Science	Physics Building
CS	Computer Science	MacLean Hall
ME	Mechanical Engineering	Engineering Building

Query Parsing Example

Return all students who major in computer science.

SELECT Name FROM Student WHERE Major='CS'

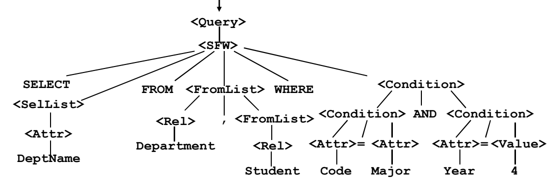



```
Rules applied:
<Query> ::= <SFW>
<SFW> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>
<SelList> ::= <Attr>                  (<Attr> = "Name")
<Condition> ::= <Attr> = <Value>  (<Attr>="Major", <Value>="CS")
<FromList> ::= <Rel>                  (<Rel> = "Student")
```


Query Parsing Example 2

Return all departments who have a 4th year student.

SELECT DeptName FROM Department, Student WHERE Code = Major AND Year = 4

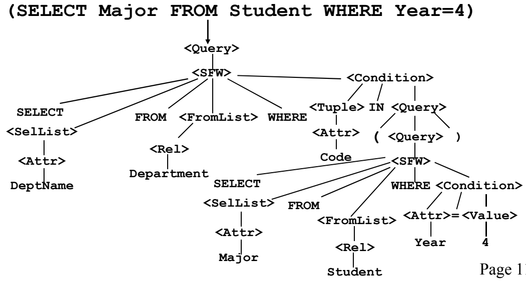


Can you determine what rules are applied?

Query Parsing Example 3

Return all departments who have a 4th year student.

SELECT DeptName FROM Department WHERE Code IN (SELECT Major FROM Student WHERE Year=4)



Query Processor Components
The Parser Functionality

The parser converts an SQL string to a parse tree.

- This involves breaking the string into tokens.
- Each token is matched with the grammar rules according to the current parse tree.
- Invalid tokens (not in grammar) generate an error.
- If there are no rules in the grammar that apply to the current SQL string, the command will be flagged to have a syntax error.

We will not concern ourselves with how the parser works. However, we will note that the parser is responsible for checking for ***syntax errors*** in the SQL statement.

- That is, the parser determines if the SQL statement is valid according to the grammar.

Query Processor Components
The Preprocessor

The preprocessor is a component of the parser that performs ***semantic validation***.

The preprocessor runs ***after*** the parser has built the parse tree. Its functions include:

- Mapping views into the parse tree if required.
- Verify that the relation and attribute names are actually valid relations and attributes in the database schema.
- Verify that attribute names have a corresponding relation name specified in the query. (Resolve attribute names to relations.)
- Check types when comparing with constants or other attributes.

If a parse tree passes syntax and semantic validation, it is called a ***valid parse tree***.

A valid parse tree is sent to the logical query processor, otherwise an error is sent back to the user.

Query Parsing Question

Question: Select a true statement.

A) The SQL grammar contains information to validate if a given field name is a valid field in the database.

B) The preprocessor runs before the parsing process.

C) SQL syntax errors are checked by the preprocessor.

D) Errors indicating a table does not exist are generated by the preprocessor.

Query Processor Components
Translator

The ***translator***, or ***logical query processor***, is the component that takes the parse tree and converts it into a logical query tree.

A ***logical query tree*** is a tree consisting of relational operators and relations. It specifies what operations to apply and the order to apply them. A logical query tree does *not* select a particular algorithm to implement each relational operator.

We will study some rules for how a parse tree is converted into a logical query tree.

Parse Trees to Logical Query Trees

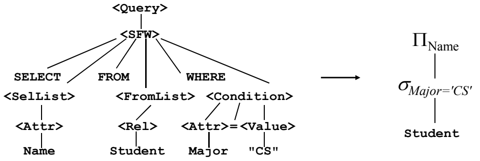
The simplest parse tree to convert is one where there is only one select-from-where (`<SFW>`) construct, and the `<Condition>` construct has no nested queries.

The logical query tree produced consists of:

- 1) The cross-product (×) of all relations mentioned in the `<FromList>` which are inputs to:
- 2) A selection operator, σ_C, where *C* is the `<Condition>` expression in the construct being replaced which is the input to:
- 3) A projection, π_L, where *L* is the list of attributes in the `<SelList>`.

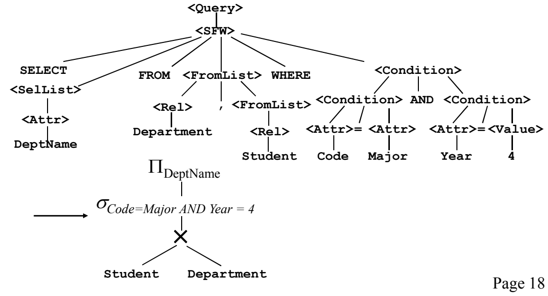
Parse Tree to Logical Tree Example

```
SELECT Name FROM Student WHERE Major='CS'
```



Parse Tree to Logical Tree Example 2

```
SELECT DeptName FROM Department, Student
   WHERE Code = Major AND Year = 4
```



Converting Nested Parse Trees to Logical Query Trees

Converting a parse tree that contains a nested query is slightly more challenging.

A nested query may be ***correlated*** with the outside query if it must be re-computed for every tuple produced by the outside query. Otherwise, it is ***uncorrelated***, and the nested query can be converted to a non-nested query using joins.

We will define a two-operand selection operator σ that takes the outer relation ***R*** as one input (left child), and the right child is the condition applied to each tuple of ***R***.

- The condition is the subquery involving `IN`.

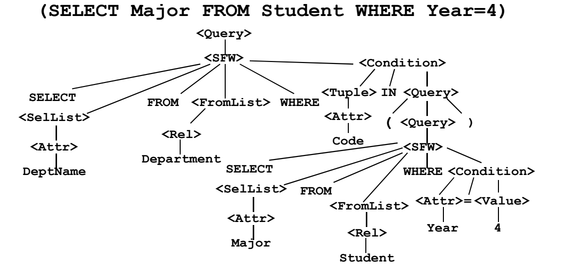
Converting Nested Parse Trees to Logical Query Trees (2)

The nested subquery translation algorithm involves defining a tree from root to leaves as follows:

- 1) Root node is a projection, π_L, where *L* is the list of attributes in the `<SelList>` of the outer query.
- 2) Child of root is a selection operator, σ_C, where *C* is the `<Condition>` expression in the outer query ignoring the subquery.
- 3) The two-operand selection operator σ with left-child as the cross-product ($\times$) of all relations mentioned in the `<FromList>` of the outer query, and right child as the `<Condition>` expression for the subquery.
- 4) The subquery itself involved in the `<Condition>` expression is translated to relational algebra.

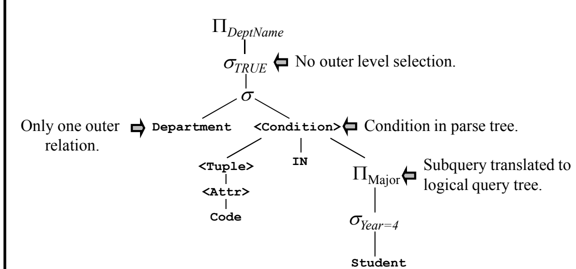
Parse Tree to Logical Tree Example 3

```
SELECT DeptName FROM Department WHERE Code IN
   (SELECT Major FROM Student WHERE Year=4)
```



Parse Tree to Logical Tree Example 3 (2)

```
SELECT DeptName FROM Department WHERE Code IN
   (SELECT Major FROM Student WHERE Year=4)
```



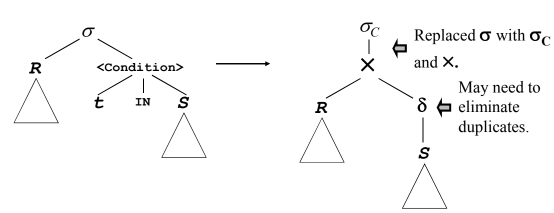
Converting Nested Parse Trees to Logical Query Trees (3)

Now, we must remove the two-operand selection and replace it by relational algebra operators.

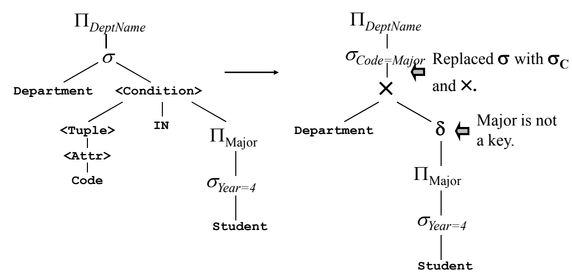
Rule for replacing two-operand selection (uncorrelated):

- Let *R* be the first operand, and the second operand is a `<Condition>` of the form *t* IN *S*. (*S* is uncorrelated subquery.)
- 1) Replace `<Condition>` by the tree that is expression for *S*.
  - May require applying duplicate elimination if expression has duplicates.
- 2) Replace two-operand selection by one-argument selection, σ_C, where *C* is the condition that equates each component of the tuple *t* to the corresponding attribute of relation *S*.
- 3) Give σ_C an argument that is the product of *R* and *S*.

Parse Tree to Logical Tree Conversion



Parse Tree to Logical Tree Example 3 (3)



COSC 404 - Dr. Ramon Lawrence

Correlated Nested Subqueries

Translating correlated subqueries is more difficult because the result of the subquery depends on a value defined outside the query itself.

Correlated subqueries may require the subquery to be evaluated for each tuple of the outside relation as an attribute of each tuple is used as the parameter for the subquery.

- We will not study translation of correlated subqueries.

Example:

```
Return all students that are more senior than the
average for their majors.

SELECT Name FROM Student s WHERE year >
  (SELECT Avg(Year) FROM student AS s2
     WHERE s.major = s2.major)
```


COSC 404 - Dr. Ramon Lawrence

Logical Query Tree Question

Question: True or False: A logical query tree has relational algebra operators and specifies the algorithm used for each of them.

A) True

B) False

COSC 404 - Dr. Ramon Lawrence

Logical Query Tree Question (2)

Question: True or False: A logical query tree is the final plan used for executing the query.

A) True

B) False

COSC 404 - Dr. Ramon Lawrence

Parsing Review Question

Build the parse tree for the following SQL query then convert it into a logical query tree.

```
SELECT Name, DeptName FROM Department, Student
      WHERE Code = Major and Code = 'CS'
```


COSC 404 - Dr. Ramon Lawrence

Optimizing the Logical Query Plan

The translation rules converting a parse tree to a logical query tree do not always produce the best logical query tree.

It is possible to optimize the logical query tree by applying relational algebra laws to convert the original tree into a more efficient logical query tree.

Optimizing a logical query tree using relational algebra laws is called ***heuristic optimization*** because the optimization process uses common conversion techniques that result in more efficient query trees in most cases, but not always.

- The optimization rules are heuristics.

We begin with a summary of relational algebra laws.

Relational Algebra Laws

Just like there are laws associated with the mathematical operators, there are laws associated with the relational algebra operators.

These laws often involve the properties of:

- ***commutativity*** - operator can be applied to operands independent of order.
  - ⇨E.g. A + B = B + A - The "+" operator is commutative.
- ***associativity*** - operator is independent of operand grouping.
  - ⇨E.g. A + (B + C) = (A + B) + C - The "+" operator is associative.

Associative and Commutative Operators

The relational algebra operators of cross-product (×), join (⋈), set and bag union ($\cup_S$ and $\cup_B$), and set and bag intersection ($\cap_S$ and $\cap_B$) are all associative and commutative.

Commutative	Associative
$R \times S = S \times R$	$(R \times S) \times T = R \times (S \times T)$
$R \bowtie S = S \bowtie R$	$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
$R \cup S = S \cup R$	$(R \cup S) \cup T = R \cup (S \cup T)$
$R \cap S = S \cap R$	$(R \cap S) \cap T = R \cap (S \cap T)$

Laws Involving Selection

1) Complex selections involving `AND` or `OR` can be broken into two or more selections: (***splitting laws***)

$$\sigma_{C_1 \text{ AND } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R))$$
$$\sigma_{C_1 \text{ OR } C_2}(R) = (\sigma_{C_1}(R)) \cup_S (\sigma_{C_2}(R))$$

2) Selection operators can be evaluated in any order:

$$\sigma_{C_1 \text{ AND } C_2}(R) = \sigma_{C_2}(\sigma_{C_1}(R)) = \sigma_{C_1}(\sigma_{C_2}(R))$$

3) Selection can be done before or after set operations and joins:

$$\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$$
$$\sigma_C(R - S) = \sigma_C(R) - S = \sigma_C(R) - \sigma_C(S)$$
$$\sigma_C(R \cap S) = \sigma_C(R) \cap S = \sigma_C(R) \cap \sigma_C(S)$$
$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$$

Laws Involving Selection and Joins

1) Selection and cross-product can be converted to a join:

$$\sigma_C(R \times S) = R \bowtie_C S$$

2) Selection and join can also be combined:

$$\sigma_C(R \bowtie_D S) = R \bowtie_{C \text{ AND } D} S$$

Laws Involving Selection Examples

1) Example relation is *R(a,b,c)*.

Given expression: $\sigma_{(a=1 \text{ OR } a=3) \text{ AND } b<c}(R)$

Can be converted to: $\sigma_{a=1 \text{ OR } a=3}(\sigma_{b<c}(R))$

then to: $\sigma_{a=1}(\sigma_{b<c}(R)) \cup \sigma_{a=3}(\sigma_{b<c}(R))$

There is another way to divide up the expression. What is it?

2) Given relations *R(a,b)* and *S(b,c)*.

Given expression: $\sigma_{(a=1 \text{ OR } a=3) \text{ AND } b<c}(R \bowtie S)$

Can be converted to: $\sigma_{(a=1 \text{ OR } a=3)}\ \sigma_{b<c}(R \bowtie S))$

then to: $\sigma_{(a=1 \text{ OR } a=3)}(R \bowtie \sigma_{b<c}(S))$

finally to: $\sigma_{(a=1 \text{ OR } a=3)}(R) \bowtie \sigma_{b<c}(S)$

Is there anything else we could do?

Laws Involving Projection

Like selections, it is also possible to push projections down the logical query tree. However, the performance gained is less than selections because projections just reduce the number of attributes instead of reducing the number of tuples.

- Unlike selections, it is common for a pushed projection to also remain where it is.

General principle: We may introduce a projection anywhere in an expression tree, as long as it eliminates only attributes that are never used by any of the operators above, and are not in the result of the entire expression.

Note that discussion considers ***bag projection*** as normally implemented in SQL (duplicates are not eliminated).

Laws Involving Projection (2)

1) Projections can be done before joins as long as all attributes required are preserved.

$$\pi_L(R \times S) = \pi_L(\pi_M(R) \times \pi_N(S))$$
$$\pi_L(R \bowtie S) = \pi_L((\pi_M(R) \bowtie \pi_N(S))$$

⇨ *L* is a set of attributes to be projected. *M* is the attributes of *R* that are either join attributes or are attributes of *L*. *N* is the attributes of *S* that are either join attributes or attributes of *L*.

2) Projection can be done before bag union but *NOT* before set union or set/bag intersection and difference.

$$\pi_L(R \cup_B S) = \pi_L(R) \cup_B \pi_L(S)$$

3) Projection can be done before selection.

$$\pi_L(\sigma_C(R)) = \pi_L(\sigma_C(\pi_M(R)))$$

4) Only the last projection operation is needed:

$$\pi_L(\pi_M(R)) = \pi_L(R)$$

Laws Involving Projection Examples

1) Given relations *R(a,b,c)* and *S(c,d,e)*.

Given expression: $\pi_{b,d}(R \bowtie S)$

Can be converted to: $\pi_{b,d}(\pi_{b,c}(R) \bowtie \pi_{c,d}(S))$

2) Using *R(a,b,c)* and the expression: $\pi_b(\sigma_{a=5}(R))$

Can be converted to: $\pi_b(\sigma_{a=5}(\pi_{a,b}(R))$

Laws Involving Duplicate Elimination

Duplicate elimination (δ) can be done before many operators.

Note that δ(*R*) = *R* occurs when *R* has no duplicates:

- 1) *R* may be a stored relation with a primary key.
- 2) *R* may be the result after a grouping operation.

Laws for pushing duplicate elimination operator (δ):

$$\delta(R \times S) = \delta(R) \times \delta(S)$$
$$\delta(R \bowtie S) = \delta(R) \bowtie \delta(S)$$
$$\delta(R \bowtie_D S) = \delta(R) \bowtie_D \delta(S)$$
$$\delta(\sigma_C(R) = \sigma_C(\delta(R))$$

Duplicate elimination (δ) can also be pushed through bag intersection, but not across union, difference, or projection.

$$\delta(R \cap_B S) = \delta(R) \cap_B \delta(S)$$

Laws Involving Grouping

The grouping operator (γ) laws depend on the aggregate operators used.

There is one general rule, however, that grouping subsumes duplicate elimination:

$$\delta(\gamma_L(R)) = \gamma_L(R)$$

The reason is that some aggregate functions are unaffected by duplicates (`MIN` and `MAX`) while other functions are (`SUM`, `COUNT`, and `AVG`).

Relational Algebra Question

Question: How many of the following equivalences are **true**? Let *C* = predicate with only *R* attributes, *D* = predicate with only *S* attributes, and *E* = predicate with only *R* and *S* attributes.

$$\sigma_{C \text{ AND } D}(R \bowtie S) = \sigma_C(R) \bowtie \sigma_D(S)$$
$$\sigma_{C \text{ AND } D \text{ AND } E}(R \bowtie S) = \sigma_E(\sigma_C(R) \bowtie \sigma_D(S))$$
$$\sigma_{C \text{ OR } D}(R \bowtie S) = [\sigma_C(R) \bowtie S] \cup_S [R \bowtie \sigma_D(S)]$$
$$\pi_L(R \cup_S S) = \pi_L(R) \cup_S \pi_L(S)$$

A) 0
B) 1
C) 2
D) 3
E) 4

Relational Algebra Question

Give examples to show that:

- a) *Bag* projection cannot be pushed below set union.

$$\pi_L(R \cup_S S) \mathrel{!=} \pi_L(R) \cup_S \pi_L(S)$$

- b) Duplicate elimination cannot be pushed below *bag* projection.

$$\delta(\pi_L(R)) \mathrel{!=} \pi_L(\delta(R))$$

Heuristic Query Optimization

Heuristic query optimization takes a logical query tree as input and constructs a more efficient logical query tree by applying equivalence preserving relational algebra laws.

Equivalence preserving transformations insure that the query result is identical before and after the transformation is applied. Two logical query trees are ***equivalent*** if they produce the same result.

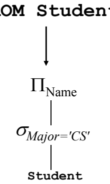
Note that heuristic optimization does not always produce the most efficient logical query tree as the rules applied are only heuristics!

Rules of Heuristic Query Optimization

1. Deconstruct conjunctive selections into a sequence of single selection operations.
2. Move selection operations down the query tree for the earliest possible execution.
3. Replace Cartesian product operations that are followed by a selection condition by join operations.
4. Execute first selection and join operations that will produce the smallest relations.
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed.

Heuristic Optimization Example

```
SELECT Name FROM Student WHERE Major="CS"
```

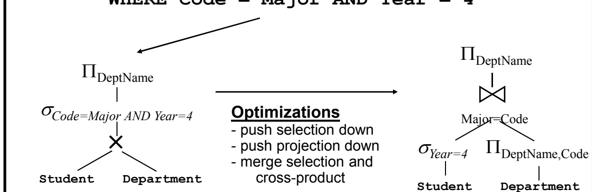


No optimization possible.

$\pi_{Name}(\sigma_{Major="CS"}(Student))$

Heuristic Optimization Example 2

```
SELECT DeptName FROM Department, Student
     WHERE Code = Major AND Year = 4
```



Original:

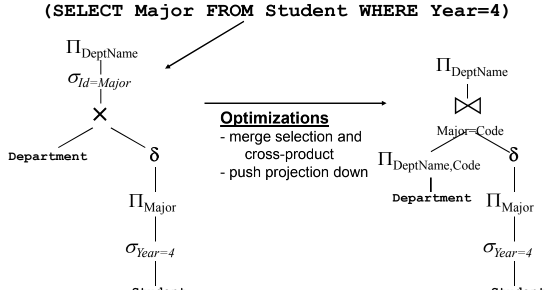
$\pi_{DeptName}(\sigma_{Code=Major\ AND\ Year=4}(Student \times Department))$

Optimized:

$\pi_{DeptName}((\sigma_{Year=4}(Student)) \bowtie_{Code=Major} (\pi_{DeptName,Code}(Department)))$

Heuristic Optimization Example 3

```
SELECT DeptName FROM Department WHERE Id IN
   (SELECT Major FROM Student WHERE Year=4)
```



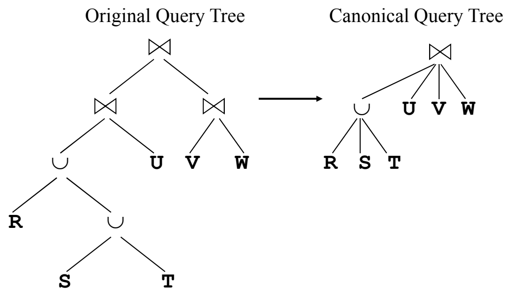
Canonical Logical Query Trees

A ***canonical logical query tree*** is a logical query tree where all associative and commutative operators with more than two operands are converted into multi-operand operators.

- This makes it more convenient and obvious that the operands can be combined in any order.

This is especially important for joins as the order of joins may make a significant difference in the performance of the query.

Canonical Logical Query Tree Example



COSC 404 - Dr. Ramon Lawrence

Canonical Query Tree Question

Question: What does the original logical query tree imply that the canonical tree does not?

A) an order of operator execution

B) the algorithms used for each relational operator

C) the sizes of each input

COSC 404 - Dr. Ramon Lawrence

Query Optimization Physical Query Plan

A ***physical query plan*** is derived from a logical query plan by:

- 1) Selecting an order and grouping for operations like joins, unions, and intersections.
- 2) Deciding on an algorithm for each operator in the logical query plan.
  - e.g. For joins: Nested-loop join, sort join or hash join
- 3) Adding additional operators to the logical query tree such as sorting and scanning that are not present in the logical plan.
- 4) Determining if any operators should have their inputs materialized for efficiency.

Whether we perform cost-based or heuristic optimization, we eventually must arrive at a physical query tree that can be executed by the evaluator.

COSC 404 - Dr. Ramon Lawrence

Query Optimization Heuristic versus Cost Optimization

To determine when one physical query plan is better than another, we must have an estimate of the cost of the plan.

Heuristic optimization is normally used to pick the best logical query plan.

Cost-based optimization is used to determine the best physical query plan given a logical query plan.

Note that both can be used in the same query processor (and typically are). Heuristic optimization is used to pick the best logical plan which is then optimized by cost-based techniques.

COSC 404 - Dr. Ramon Lawrence

Query Optimization Estimating Operation Cost

To determine when one physical query plan is better than another for cost-based optimization, we must have an estimate of the cost of a physical query plan.

Note that the query optimizer will very rarely know the exact cost of a query plan because the only way to know is to execute the query itself!

- Since the cost to execute a query is much greater than the cost to optimize a query, we cannot execute the query to determine its cost!

It is important to be able to estimate the cost of a query plan without executing it based on statistics and general formulas.

COSC 404 - Dr. Ramon Lawrence

Query Optimization Estimating Operation Cost (2)

Statistics for ***base relations*** such as ***B(R)***, ***T(R)***, and ***V(R,a)*** are used for optimization and can be gathered directly from the data, or estimated using statistical gathering techniques.

One of the most important factors determining the cost of the query is the size of the intermediate relations. An ***intermediate relation*** is a relation generated by a relational algebra operator that is the input to another query operator.

- The final result is not an intermediate relation.

The goal is to come up with general rules that estimate the sizes of intermediate relations that give accurate estimates, are easy to compute, and are consistent.

- There is no one set of agreed-upon rules!

Estimating Operation Cost
Estimating Projection Sizes

Calculating the size of a relation after the projection operation is easy because we can compute it directly.

- Assuming we know the size of the input, we can calculate the size of the output based on the size of the input records and the size of the output records.
- The projection operator decreases the size of the tuples, not the number of tuples.

For example, given relation ***R(a,b,c)*** with size of *a* = size of *b* = 4 bytes, and size of *c* = 100 bytes. *T(R)* = 10000 and unspanned block size is 1024 bytes. If the projection operation is $\Pi_{a,b}$, what is the size of the output ***U*** in blocks?

```
T(U) = 10000.  Output tuples are 8 bytes long.
bfr = 1024/8 = 128  B(U) = 10000/128 = 79
B(R) = 10000 / (1024/108) = 1112
Savings = (B(R) - B(U))/B(R)*100% = 93%
```


Estimating Operation Cost
Estimating Selection Sizes

A selection operator generally decreases the number of tuples in the output compared to the input. By how much does the operator decrease the input size?

The ***selectivity*** (***sf***) is the fraction of tuples selected by a selection operator. Common cases and their selectivities:

- 1) Equality: $S = \sigma_{a=v}(R)$ - $sf = 1/V(R,a)$ $T(S) = T(R)/V(R,a)$
  - Reason: Based on the assumption that values occur equally likely in the database. However, estimate is still the best *on average* even if the values *v* for attribute *a* are not equally distributed in the database.
- 2) Inequality: $S = \sigma_{a<v}(R)$ - $sf = 1/3$ $T(S) = T(R)/3$
  - Reason: On average, you would think that the value should be *T(R)*/2. However, queries with inequalities tend to return less than half the tuples, so the rule compensates for this fact.
- 3) Not equals: $S = \sigma_{a!=v}(R)$ - $sf = 1$ $T(S) = T(R)$
  - Reason: Assume almost all tuples satisfy the condition.

Estimating Operation Cost
Estimating Selection Sizes (2)

Simple selection clauses can be connected using AND or OR.

A complex selection operator using AND ($\sigma_{a=10 \text{ AND } b<20}(R)$) is the same as a cascade of simple selections ($\sigma_{a=10}(\sigma_{b<20}(R))$).

The selectivity is the ***product*** of the selectivity of the individual clauses.

Example: Given *R(a,b,c)* and $S = \sigma_{a=10 \text{ AND } b<20}(R)$, what is the best estimate for *T(S)*? Assume *T(R)*=10,000 and *V(R,a)* = 50.

```
The filter a=10 has selectivity of 1/V(R,a)=1/50.
The filter b<20 has selectivity of 1/3.
Total selectivity = 1/3 * 1/50 = 1/150.
T(S) = T(R)* 1/150 = 67
```


Estimating Operation Cost
Estimating Selection Sizes (3)

For complex selections using OR ($S = \sigma_{C1 \text{ OR } C2}(R)$), the # of output tuples can be estimated by the ***sum*** of the # of tuples for each condition.

- Measuring the selectivity with OR is less precise, and simply taking the sum is often an overestimate.

A better estimate assumes that the two clauses are independent, leading to the formula:

$$n * (1 - (1-m_1/n) * (1 - m_2/n))$$

- m_1 and m_2 are the # of tuples that satisfy C_1 and C_2 respectively.
- *n* is the number of tuples of *R* (i.e. *T(R)*).
- $1-m_1/n$ and $1-m_2/n$ are the fraction of tuples that do not satisfy C_1 (resp. C_2). The product of these numbers is the fraction that do not satisfy either condition.

Estimating Operation Cost
Estimating Selection Sizes (4)

Example: Given *R(a,b,c)* and $S = \sigma_{a=10 \text{ OR } b<20}(R)$, what is the best estimate for *T(S)*? Assume *T(R)*=10,000 and *V(R,a)* = 50.

```
The filter a=10 has selectivity of 1/V(R,a)=1/50.
The filter b<20 has selectivity of 1/3.
Total selectivity = (1 - (1 - 1/50)(1 - 1/3)) = .3466
T(S) = T(R) *.3466 = 3466

Simple method results in T(S) = 200 + 3333 = 3533.
```


Estimating Operation Cost
Estimating Join Sizes

We will only study estimating the size of natural join.

- Other types of joins are equivalent or can be translated into a cross-product followed by a selection.

The two relations joined are ***R(X,Y)*** and ***S(Y,Z)***.

- We will assume *Y* consists of only one attribute.

The challenge is we do not know how the set of values of *Y* in *R* relate to the values of *Y* in *S*. There are some possibilities:

- 1) The two sets are disjoint. Result size = **0**.
- 2) *Y* may be a foreign key of *R* joining to a primary key of *S*. Result size in this case is ***T(R)***.
- 3) Almost all tuples of *R* and *S* have the same value for *Y*, so result size in the worst case is ***T(R)*T(S)***.

Estimating Join Sizes (2)

The result size of joining relations ***R(X,Y)*** and ***S(Y,Z)*** can be approximated by:

$$\frac{T(R) * T(S)}{\max(V(R,Y), V(S,Y))}$$

- Argument:
  - ⇨ Every tuple of *R* has a 1/*V(S,Y)* chance of joining with every tuple of *S*. On average then, each tuple of *R* joins with *T(S)*/*V(S,Y)* tuples. If there are *T(R)* tuples of *R*, then the expected size is *T(R)* * *T(S)*/*V(S,Y)*.
  - ⇨ A symmetric argument can be made from the perspective of joining every tuple of *S*. Each tuple has a 1/*V(R,Y)* chance of joining with every tuple of *R*. On average, each tuple of *R* joins with *T(R)*/*V(R,Y)* tuples. The expected size is then *T(S)* * *T(R)*/*V(R,Y)*.
  - ⇨ In general, we choose the smaller estimate for the result size (divide by the maximum value).

Estimating Operation Cost Estimating Join Sizes Example

Example:
- *R(a,b)* with *T(R)* = 1000 and *V(R,b)* = 20.
- *S(b,c)* with *T(S)* = 2000, *V(S,b)* = 50, and *V(S,c)* = 100
- *U(c,d)* with *T(U)* = 5000 and *V(U,c)* = 500

Calculate the natural join $R \bowtie S \bowtie U$.

1) $(R \bowtie S) \bowtie U$ -

$T(R \bowtie S) = T(R)T(S)/\max(V(R,b),V(S,b))$

= 1000 * 2000 / 50 = 40,000

Now join with *U*.

Final size = $T(R \bowtie S)*T(U)/\max(V(R \bowtie S,c),V(U,c))$

= 40000 * 5000 / 500 = 400,000

Now, calculate the natural join like this: $R \bowtie (S \bowtie U)$.
- Which of the two join orders is better?

Estimating Join Sizes Estimating V(R,a)

The database will keep statistics on the number of distinct values for each attribute *a* in each relation *R*, ***V(R,a)***.

When a sequence of operations is applied, it is necessary to estimate *V(R,a)* on the intermediate relations.

For our purposes, there will be three common cases:
- *a* is the primary key of *R* then **V(R,a) = T(R)**
  - ⇨ The number of distinct values is the same as the # tuples in *R*.
- *a* is a foreign key of *R* to another relation *S* then **V(R,a) = T(S)**
  - ⇨ In the worst case, the number of distinct values of *a* cannot be larger than the number of tuples of *S* since *a* is a foreign key to the primary key of *S*.
- If a selection occurs on relation R before a join, then **V(R,a)** after the selection is the same as **V(R,a)** before selection.
  - ⇨ This is often strange since V(R,a) may be greater than # of tuples in intermediate result! V(R,a) <> # of tuples in result.

Estimating Operation Cost Estimating Sizes of Other Operators

The size of the result of set operators, duplicate elimination, and grouping is hard to determine. Some estimates are below:
- Union
  - ⇨ bag union = sum of two argument sizes
  - ⇨ set union = minimum is the size of the largest relation, maximum is the sum of the two relations sizes. Estimate by taking average of min/max.
- Intersection
  - ⇨ minimum is 0, maximum is size of smallest relation. Take average.
- Difference
  - ⇨ Range is between *T(R)* and *T(R)* - *T(S)* tuples. Estimate: *T(R)* - 1/2**T(S)*
- Duplicate Elimination
  - ⇨ Range is 1 to *T(R)*. Estimate by either taking smaller of 1/2**T(R)* or product of all $V(R,a_i)$ for all attributes a_i.
- Grouping
  - ⇨ Range and estimate is similar to duplicate elimination.

Query Optimization Cost-Based Optimization

Cost-based optimization is used to determine the best physical query plan given a logical query plan.

The cost of a query plan in terms of disk I/Os is affected by:
- 1) The logical operations chosen to implement the query (the logical query plan).
- 2) The sizes of the intermediate results of operations.
- 3) The physical operators selected.
- 4) The ordering of similar operations such as joins.
- 5) If the inputs are materialized.

Cost-Based Optimization Obtaining Size Estimates

The cost calculations for the physical operators relied on reasonable estimates for *B(R)*, *T(R)*, and *V(R,a)*.

Most DBMSs allow an administrator to explicitly request these statistics be gathered. It is easy to gather them by performing a scan of the relation. It is also common for the DBMS to gather these statistics independently during its operation.
- Note that by answering one query using a table scan, it can simultaneously update its estimates about that table!

It is also possible to produce a histogram of values for use with *V(R,a)* as not all values are equally likely in practice.
- Histograms display the frequency that attribute values occur.

Since statistics tend not to change dramatically, statistics are computed only periodically instead of after every update.

Using Size Estimates in Heuristic Optimization

Size estimates can also be used during heuristic optimization.

In this case, we are not deciding on a physical plan, but rather determining if a given logical transformation will make sense.

By using statistics, we can estimate intermediate relation sizes (independent of the physical operator chosen), and thus determine if the logical transformation is useful.

Using Size Estimates in Cost-based Optimization

Given a logical query plan, the simplest algorithm to determine the best physical plan is an exhaustive search.

In an ***exhaustive search***, we evaluate the cost of every physical plan that can be derived from the logical plan and pick the one with minimum cost.

The time to perform an exhaustive search is extremely long because there are many combinations of physical operator algorithms, operator orderings, and join orderings.

Using Size Estimates in Cost-based Optimization (2)

Since exhaustive search is costly, other approaches have been proposed based on either a top-down or bottom-up approach.

Top-down algorithms start at the root of the logical query tree and pick the best implementation for each node starting at the root.

Bottom-up algorithms determine the best method for each subexpression in the tree (starting at the leaves) until the best method for the root is determined.

Cost-Based Optimization Choosing a Selection Method

In building the physical query plan, we will have to pick an algorithm to evaluate each selection operator.

Some of our choices are:

- table scan
- index scan

There also may be several variants of each choice if there are multiple indexes.

We evaluate the cost of each choice and select the best one.

Cost-Based Optimization Choosing a Join Method

In building the physical query plan, we will have to pick an algorithm to evaluate each join operator:

- **nested-block join** - one-pass join or nested-block join used if reasonably sure that relations will fit in memory.
- **sort-join** is good when arguments are sorted on the join attribute or there are two or more joins on the same attribute.
- **index-join** may be used when an index is available.
- **hash-join** is generally used if a multipass join is required, and no sorting or indexing can be exploited.

Cost-Based Optimization Pipelining versus Materialization

The default action for iterators is ***pipelining*** when the inputs to the operator provide results a tuple-at-a-time.

However, some operators require the ability to scan the inputs multiple times. This requires the input operator to be able to support ***rescan***.

An alternative to using rescan is to materialize the results of an input to disk. This has two benefits:

- Operators do not have to implement rescan.
- It may be more efficient to compute the result once, save it to disk, then read it from disk multiple times than to re-compute it each time.

Plans can use a materialization operator at any point to materialize the output of another operator.

Selecting a Join Order

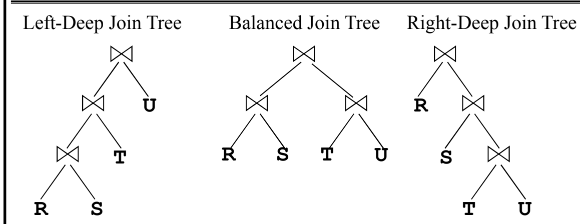
Since joins are the most costly operation, determining the best possible join order will result in more efficient queries.

Selecting a join order is most important if we are performing a join of three or more relations. However, a join of two relations can be evaluated in two different ways depending on which relation is chosen to be the left argument.

- Some algorithms (such as nested-block join and one-pass join) are more efficient if the left argument is the smaller relation.

A ***join tree*** is used to graphically display the ***join order***.

Join Tree Examples



Join Tree Question

Question: How many possible join tree shapes (different trees ignoring relations at leaves) are there for joining 4 nodes?

A) 3
B) 4
C) 5
D) 6
E) 8

Join Tree Question (2)

Question: Assuming that every relation can join with every other relation, how many distinct join trees (considering different relations at leaf nodes) are there for joining 4 nodes?

A) 256
B) 120
C) 60
D) 20
E) 5

Cost-Based Optimization Selecting a Join Order

Dynamic programming is used to select a join order.

Algorithm to find best join tree for a set of *n* relations:

- 1) Find the best plan for each relation.
  - File scan, index scan
- 2) Find the best plan to combine pairs of relations found in step #1. If have two plans for *R* and *S*, test
  - *R* ⋈ *S* and *S* ⋈ *R* for all types of joins.
  - May also consider interesting sort orders.
- 3) Of the plans produced involving two relations, add a third relation and test all possible combinations.

In practice the algorithm works top down recursively and remembers the best subplans for later use.

Join Order Dynamic Programming Algorithm

```
// S is set of relations to join
procedure findBestPlan(S)
{       if (bestplan[S].cost ≠ ∞)          // bestplan stores computed plans
                return bestplan[S];
        // else bestplan[S] has not been computed. Compute it now.
        for each non-empty subset S1 of S such that S1 ≠ S
        {       P1= findBestPlan(S1);
                P2= findBestPlan(S - S1);
                A = best algorithm for join of P1 and P2;
                cost = P1.cost + P2.cost + cost of A;
                if (cost < bestplan[S].cost)
                {       bestplan[S].cost = cost;
                        bestplan[S].plan = P1 ⋈ P2 using A;
                }
        }
        return bestplan[S];
}
```

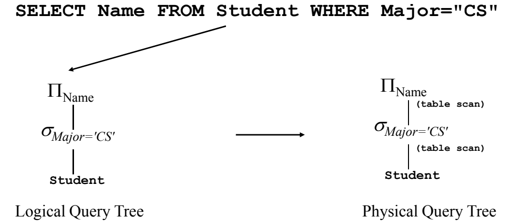
Cost-Based Optimization Example

We will perform cost-based optimization on the three example queries giving the following statistics:

- *T(Student)* = 200,000 ; *B(Student)* = 50,000
- *T(Department)* = 4 ; *B(Department)* = 4
- *V(Student, Major)* = 4 ; *V(Student, Year)* = 4
- *Student* has B+-tree secondary indexes on *Major* and *Year*, and primary index on *Id*.
- *Department* has a primary index on *Code*.

Cost-Based Optimization Example

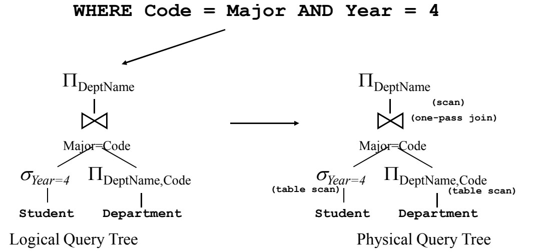
```
SELECT Name FROM Student WHERE Major="CS"
```



Selection will return *T(Student)*/*V(Student,Major)* = 200,000/4 = 50,000 tuples.
Since tuples are not sorted by *Major*, each read may potentially require reading another block (results in another seek + rotational latency).
Thus, table scan will be more efficient.
Projection performed using table scan of pipelined output from selection.

Cost-Based Optimization Example 2

```
SELECT DeptName FROM Department, Student
WHERE Code = Major AND Year = 4
```



Selection uses table scan again due to high selectivity.
One-pass join chosen as result from *Department* subtree is small. Index-join cannot be used as already performed projection on base relation.

Cost-Based Optimization Example 3

Consider a query involving the join of relations:

- *Enrolled(StudentID, Year, CourseID)*
- *Course(CID, Name)*
- and the relations *Student* and *Department*.
- That is, *Student* ⋈ *Department* ⋈ *Enrolled* ⋈ *Course*.

Determine the best join ordering given this information:

- *T(Enrolled)* = 1,000,000; *B(Enrolled)* = 200,000
- *V(Enrolled,StudentID)* = 180,000 ; *V(Enrolled,CourseID)* = 900
- *T(Course)* = 1000 ; *B(Course)* = 100

The best join ordering would have the minimum sizes for the intermediate relations, and we would like to perform the join with the greatest selectivity first.

Cost-Based Optimization Example 3 (2)

Possible join pairs and intermediate result sizes:

- *Student* ⋈ *Department* = 200,000 * 4 / max(4,4) = **200,000**
- *Student* ⋈ *Enrolled*
  = 200,000*1,000,000 / max(200,000,180,000) = **1,000,000**
- *Enrolled* ⋈ *Course*
  =1,000,000 * 1,000 / max(900,1000) = **1,000,000**

Conclusion: Join *Student* and *Department* first as it results in smallest intermediate relation. Then, join that result with *Enrolled*, finally join with *Course*.

Cost-based Optimization Question

Question: Would it be better or worse if we joined *Enrolled* with *Course* then joined that with the result of *Student* and *Department*?

A) same

B) better

C) worse

Join Ordering Example

Query:

```
SELECT * FROM Course C, Enrolled E, Student S
   WHERE Year = 4 AND C.cid = 'COSC404' AND
       E.cid = E.cid and E.sid = S.sid
```

Relation statistics:

- *B(C) = 100, B(E) = 200,000, B(S) = 20,000*
- *T(C)* = 1,000 ; *T(E)* = 1,000,000 ; T(S) = 200,000
- Assume block size = 1000 bytes.
- *Tuple sizes: C = 100 bytes ; E = 200 bytes ; S = 100 bytes*
- *V(E,sid)* = 180,000 ; *V(E,cid)* = 900
- *Student* has secondary B-tree index on *Year*.
- *Course* has primary B-tree index on *cid*.

Join Ordering Example (2)

The first step is to calculate best plan for each relation:

Enrolled

- only choice is file scan at cost = 200,000

Course with filter `cid = 'COSC404'`:

- file scan cost = 100
- index scan cost = 1 (assume get record in 1 block with index)
- Best plan = index scan with cost = 1

Student with filter `Year = 4`:

- file scan cost = 20,000
- index scan will return approximately ¼ of records (50,000). If assume each does a block access that is 50,000 cost.
- Best plan = file scan with cost = 20,000

Join Ordering Example (3)

Now calculate all pairs of relations (sets of size two). Test all types of joins (sort, hash, block). Assume left is build input and M= 1000.

Enrolled, Course: (output size tuples = 1111 blocks = 334)

- Enrolled ⋈ Course
  - ⇨ Sort = 600,003 ; Hash = 598,003 ; Block nested = 200,201
- Course ⋈ Enrolled
  - ⇨ Sort = 600,003 ; **Hash = 200,001**; Block nested = 200,001

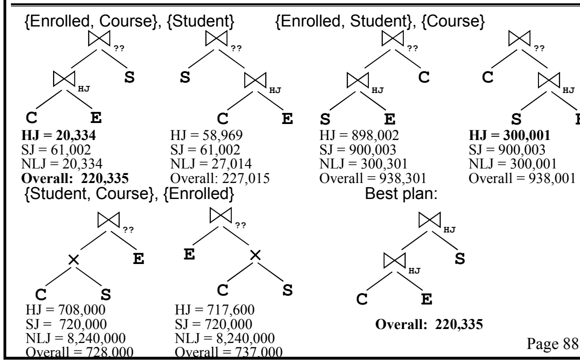
Enrolled, Student: (output size tuples = 1,000,000 blocks = 300,000)

- Enrolled ⋈ Student
  - ⇨ Sort = 660,000 ; Hash = 657,800 ; Block nested = 4,040,000
- Student ⋈ Enrolled
  - ⇨ Sort = 660,000 ; **Hash = 638,000** ; Block nested = 4,220,000

Student, Course (Note: This may not be done if cross-products are not allowed.)

- Student X Course cost = 20,000 output size = 40,000 blocks

Join Ordering Example (4)



Conclusion

A query processor first parses a query into a parse tree, validates its syntax, then translates the query into a relational algebra ***logical query plan***.

The logical query plan is optimized using ***heuristic optimization*** that uses equivalence preserving transformations.

Cost-based optimization is used to select a join ordering and build an execution plan which selects an implementation for each of the relational algebra operations in the logical tree.

Major Objectives

The "One Things":

- Convert an SQL query to a parse tree using a grammar.
- Convert a parse tree to a logical query tree.
- Use heuristic optimization and relational algebra laws to optimize logical query trees.
- Convert a logical query tree to a physical query tree.
- Calculate size estimates for selection, projection, joins, and set operations.

Major Theme:

- The query optimizer uses heuristic (relational algebra laws) and cost-based optimization to greatly improve the performance of query execution.

Objectives

- Explain the difference between syntax and semantic validation and the query processor component responsible for each.
- Define: valid parse tree, logical query tree, physical query tree
- Explain the difference between correlated and uncorrelated nested queries.
- Define and use canonical logical query trees.
- Define: join-orders: left-deep, right-deep, balanced join trees
- Explain issues in selecting algorithms for selection and join.
- Compare/contrast materialization versus pipelining and know when to use them when building physical query plans.