



The Query Compiler

Chapter 16



Objectives ...

- Convert an SQL query to a parse tree using a grammar.
- 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
- Convert parse tree to logical query tree for regular and nested queries.
- Explain the difference between correlated and uncorrelated nested queries.
- Use heuristic optimization (6 rules) and relational algebra laws to optimize logical query trees.
 - selection laws (splitting law), projection laws, laws for joins, duplicate elimination, and grouping, equivalence preserving transformations



... Objectives

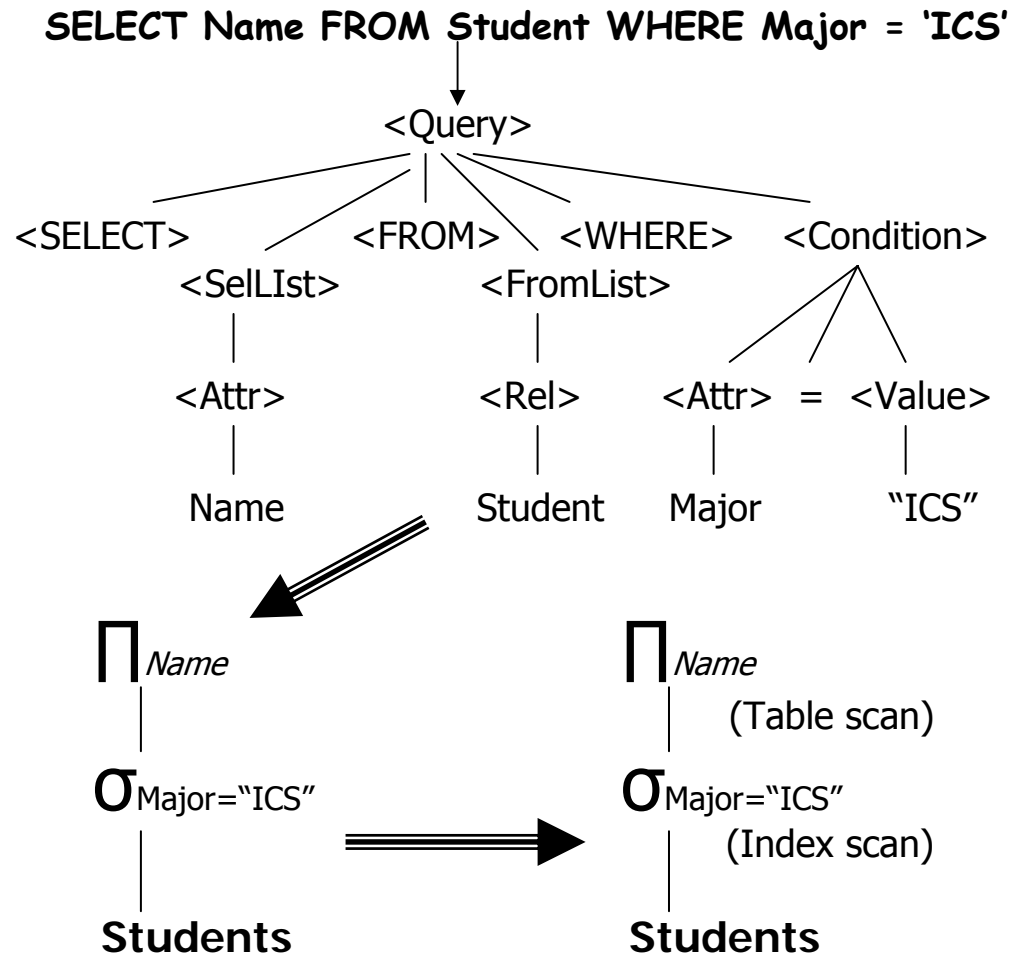
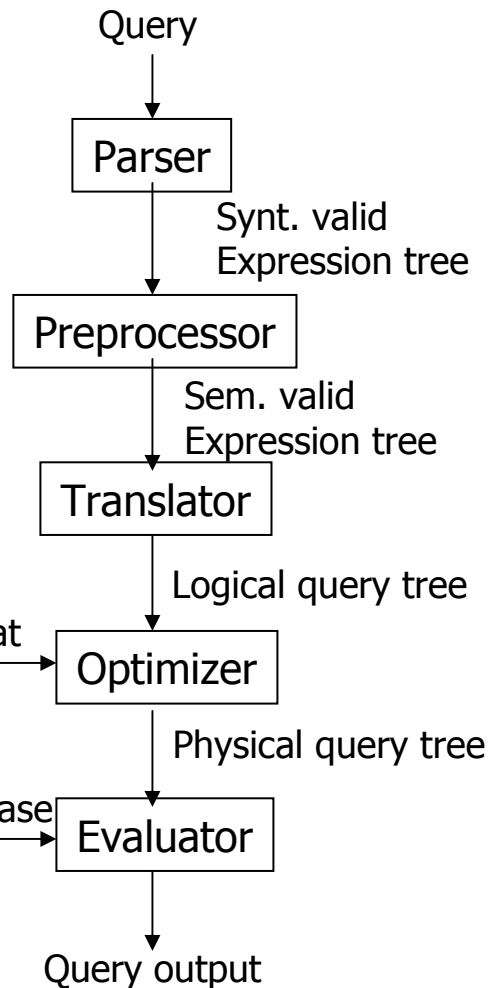
- Define and use canonical logical query trees.
- Convert logical query tree to physical query tree.
- Calculate estimates for estimating operation costs/sizes for selection, projection, joins, and set operations.
- List the different approach to finding an "optimal" physical query plan.
- Define: join-orders: 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.



- Lecture outline

- Components of Query Processor
- Optimizing the Logical Query Plan

- Components of Query Processor





-- The Parser

- The role of the parser is to convert a 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 have to support lists of attributes in the SELECT clause, lists of relations in the FROM clause, and conditions in the WHERE clause.



-- Simple SQL Grammar

- $\langle \text{Query} \rangle ::= \langle \text{SFW} \rangle$
- $\langle \text{Query} \rangle ::= (\langle \text{Query} \rangle)$
- $\langle \text{SFW} \rangle ::= \text{SELECT } \langle \text{SelList} \rangle \text{ FROM } \langle \text{FromList} \rangle \text{ WHERE } \langle \text{Condition} \rangle$
- $\langle \text{SelList} \rangle ::= \langle \text{Attr} \rangle$
- $\langle \text{SelList} \rangle ::= \langle \text{Attr} \rangle , \langle \text{SelList} \rangle$
- $\langle \text{FromList} \rangle ::= \langle \text{Rel} \rangle$
- $\langle \text{FromList} \rangle ::= \langle \text{Rel} \rangle , \langle \text{FromList} \rangle$
- $\langle \text{Condition} \rangle ::= \langle \text{Condition} \rangle \text{ AND } \langle \text{Condition} \rangle$
- $\langle \text{Condition} \rangle ::= \langle \text{Tuple} \rangle \text{ IN } \langle \text{Query} \rangle$
- $\langle \text{Condition} \rangle ::= \langle \text{Attr} \rangle = \langle \text{Attr} \rangle$
- $\langle \text{Condition} \rangle ::= \langle \text{Attr} \rangle \text{ LIKE } \langle \text{Value} \rangle$
- $\langle \text{Condition} \rangle ::= \langle \text{Attr} \rangle = \langle \text{Value} \rangle$
- $\langle \text{Tuple} \rangle ::= \langle \text{Attr} \rangle$ // 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

ID	Name	Major	Year
11111	Aaaa	ICS	4
22222	Bbbb	EE	3
33333	Ccccc	ME	3
44444	Ddddd	ICS	1
55555	Eeeee	AE	4
66666	Ffffff	EE	1
77777	Gggggg	AE	2
88888	Hhhhh	ME	2

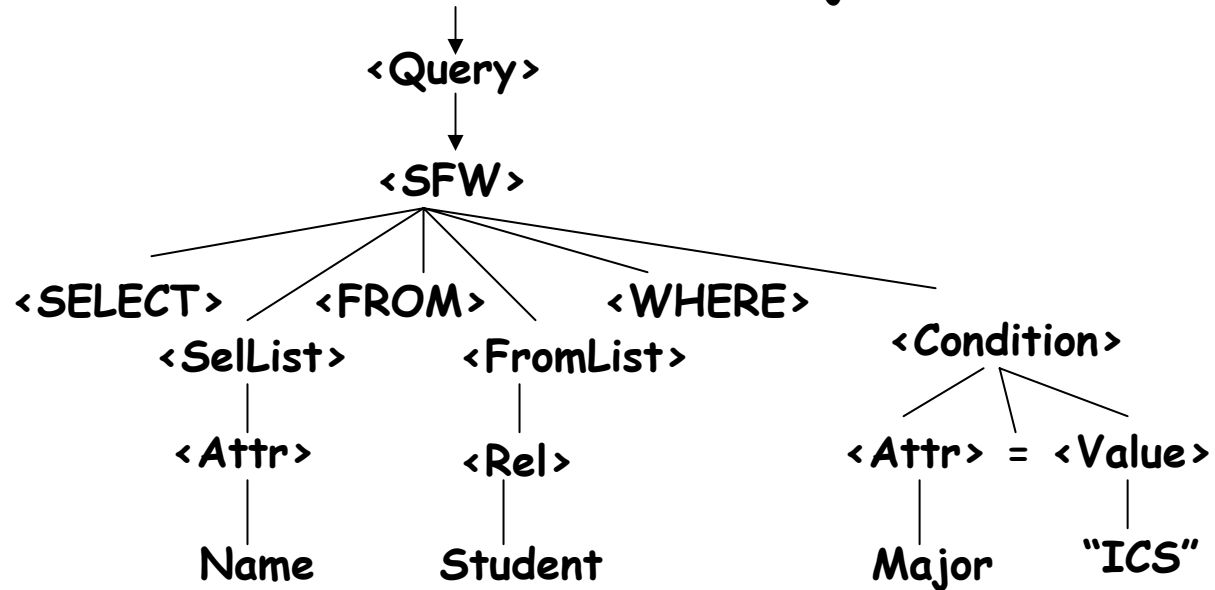
Department

Code	DeptName	Location
EE	Electrical Engineering	14
ME	Mechanical Engineering	22
ICS	Computer Science	23
AE	Architecture Engineering	19

-- Query Parsing Example

Return all students who major in computer science.

SELECT Name FROM Student WHERE Major = "ICS";



Rules applied:

$\langle \text{Query} \rangle ::= \langle \text{SFW} \rangle$

$\langle \text{SFW} \rangle ::= \text{SELECT } \langle \text{SelList} \rangle \text{ FROM } \langle \text{FromList} \rangle \text{ WHERE } \langle \text{Condition} \rangle$

$\langle \text{SelList} \rangle ::= \langle \text{Attr} \rangle$ ($\langle \text{Attr} \rangle = \text{"Name"}$)

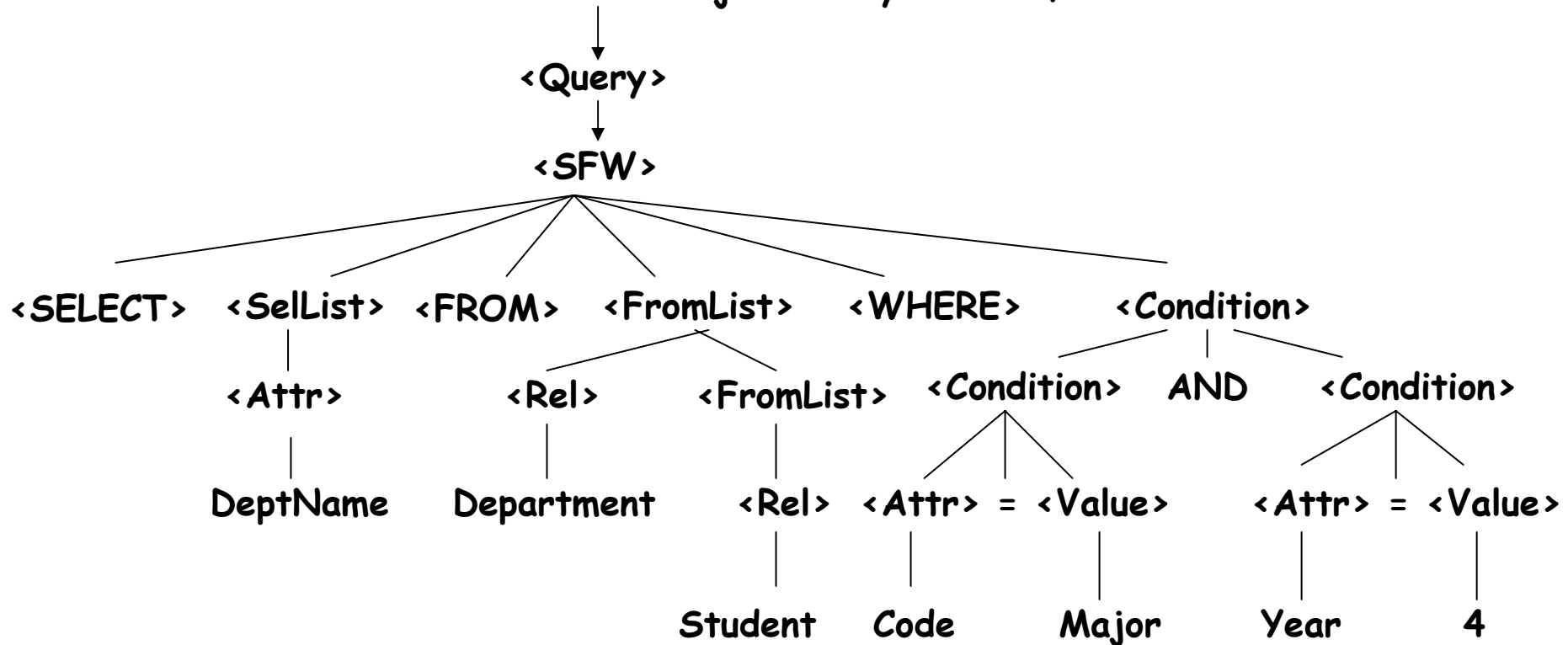
$\langle \text{Condition} \rangle ::= \langle \text{Attr} \rangle = \langle \text{Value} \rangle$ ($\langle \text{Attr} \rangle = \text{"Major"}$, $\langle \text{Value} \rangle = \text{"ICS"}$)

$\langle \text{FromList} \rangle ::= \langle \text{Rel} \rangle$ ($\langle \text{Rel} \rangle = \text{"Student"}$)

-- Query Parsing Example

Return all departments who have a 4th year student.

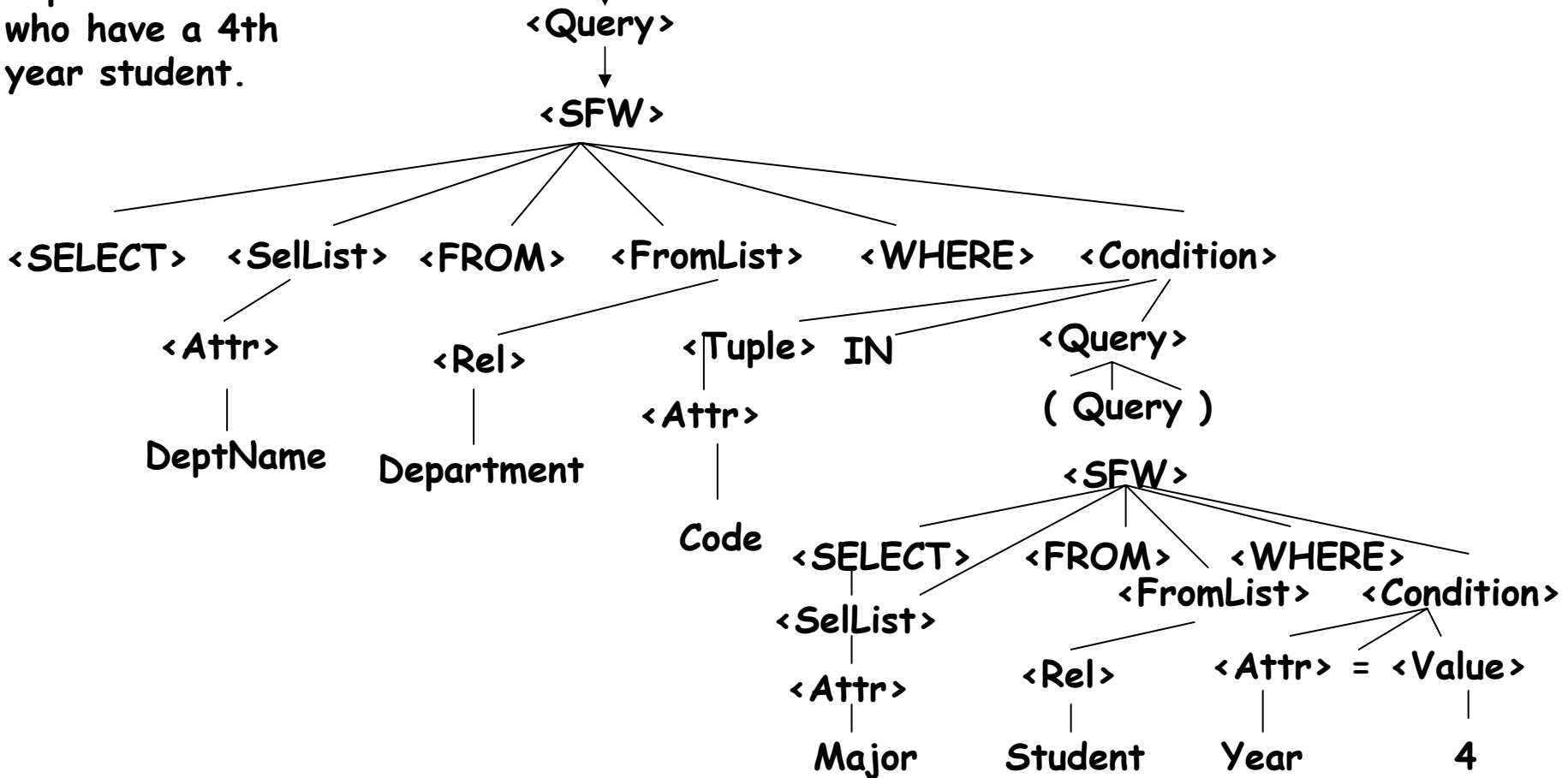
```
SELECT DeptName FROM Student, Department
WHERE Code = Major AND year = 4 ;
```



-- Query Parsing Example

Return all departments who have a 4th year student.

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





-- 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.



-- 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.



-- The 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, but not how to actually implement the operations.
 - A logical query tree does not select a particular algorithm to implement each relational operator.
- We will give some informal rules explaining how the parse tree is converted into a logical query tree.

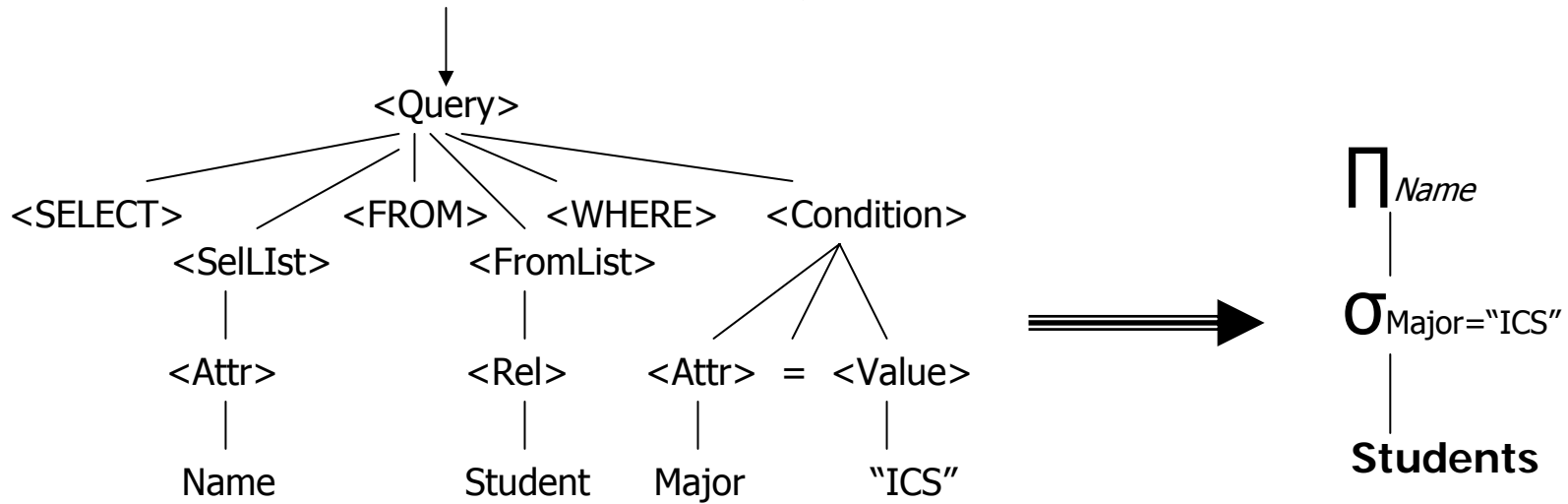


-- Parse Trees to Logical Query Trees

- 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 (\times) 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 1

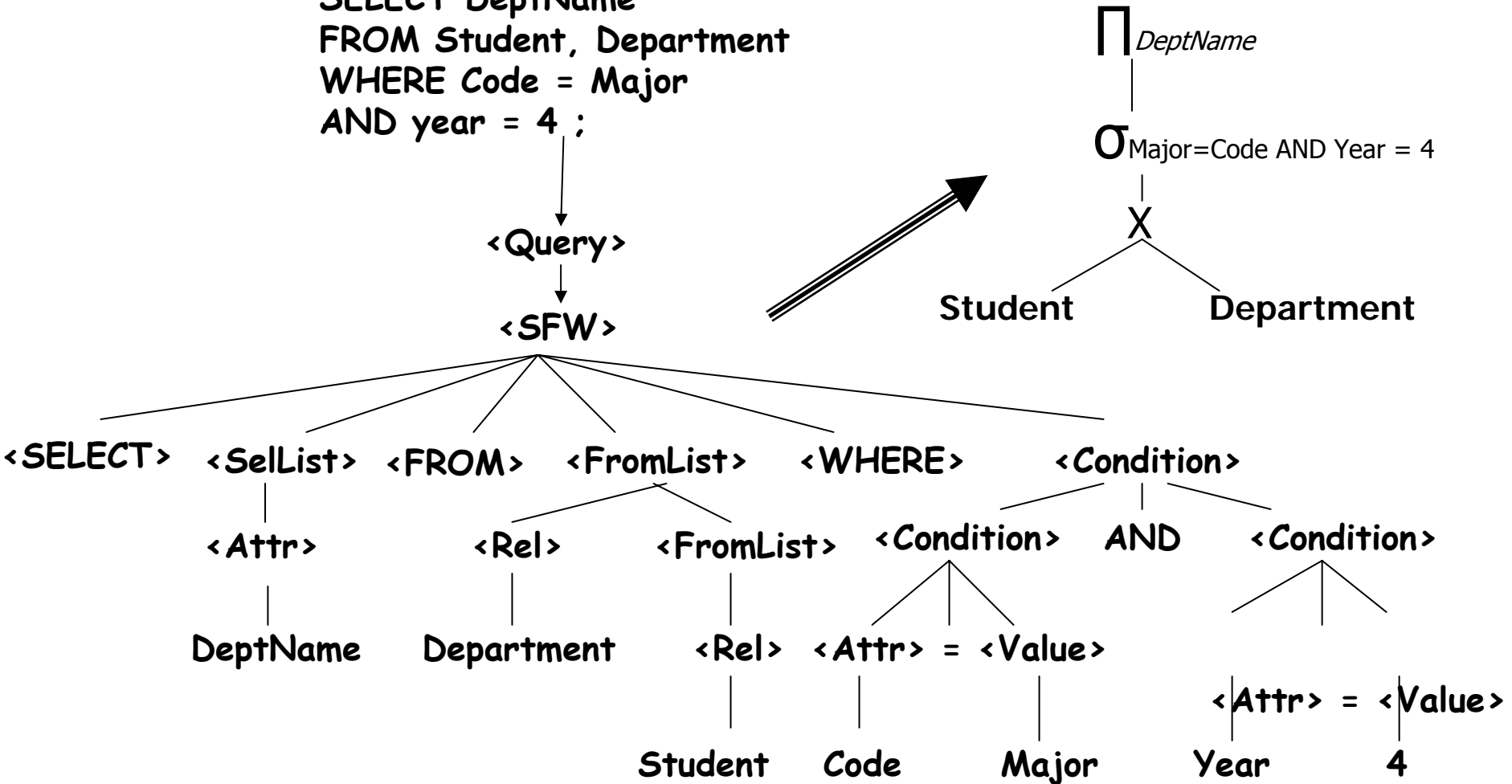
SELECT Name FROM Student WHERE Major = 'ICS'



-- Parse Tree to Logical Tree: Example 2

```

SELECT DeptName
FROM Student, Department
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.



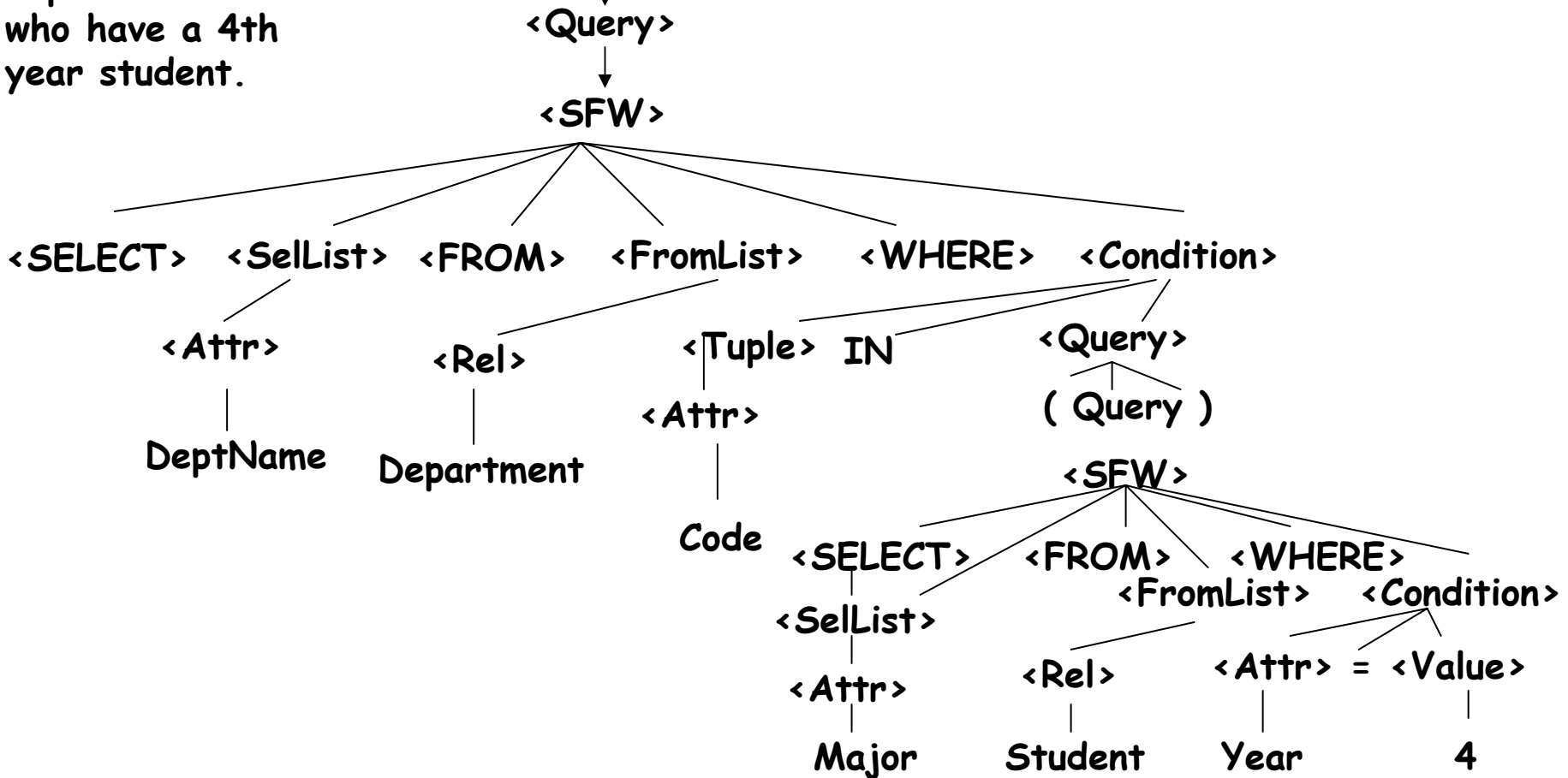
... -- Converting Nested Parse Trees to Logical Query Trees

- 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 $\langle \text{SelList} \rangle$ of the outer query.
 2. Child of root is a selection operator, σ_C , where C is the $\langle \text{Condition} \rangle$ 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 $\langle \text{FromList} \rangle$ of the outer query, and right child as the $\langle \text{Condition} \rangle$ expression for the subquery.
 4. The subquery itself involved in the $\langle \text{Condition} \rangle$ expression is translated to relational algebra.

-- Parse Tree to Logical Tree: Example 3 ...

Return all departments who have a 4th year student.

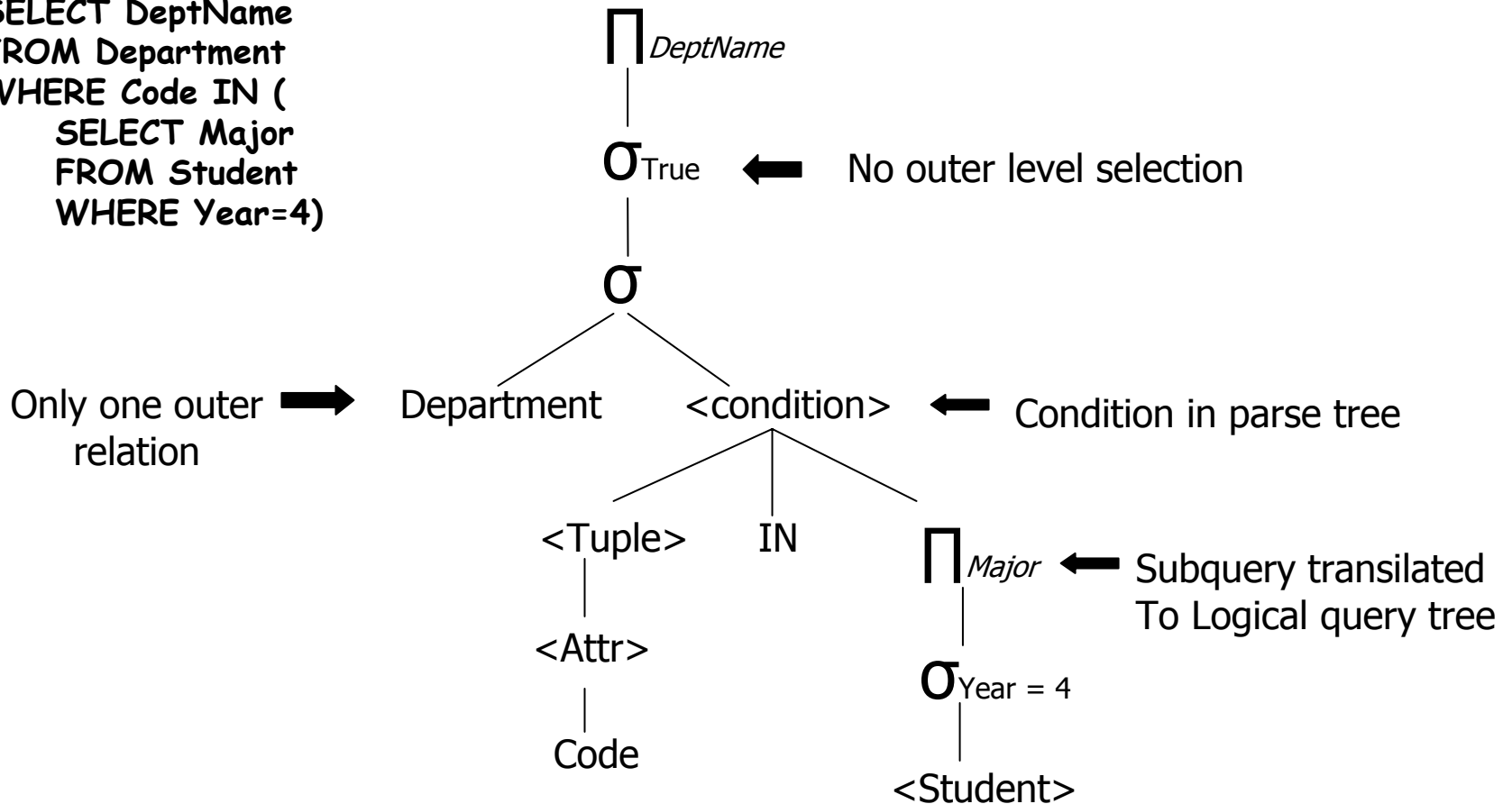
SELECT DeptName FROM Department WHERE Code IN
(SELECT Major FROM Student WHERE Year=4)



... -- Parse Tree to Logical Tree: Example 3

```

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

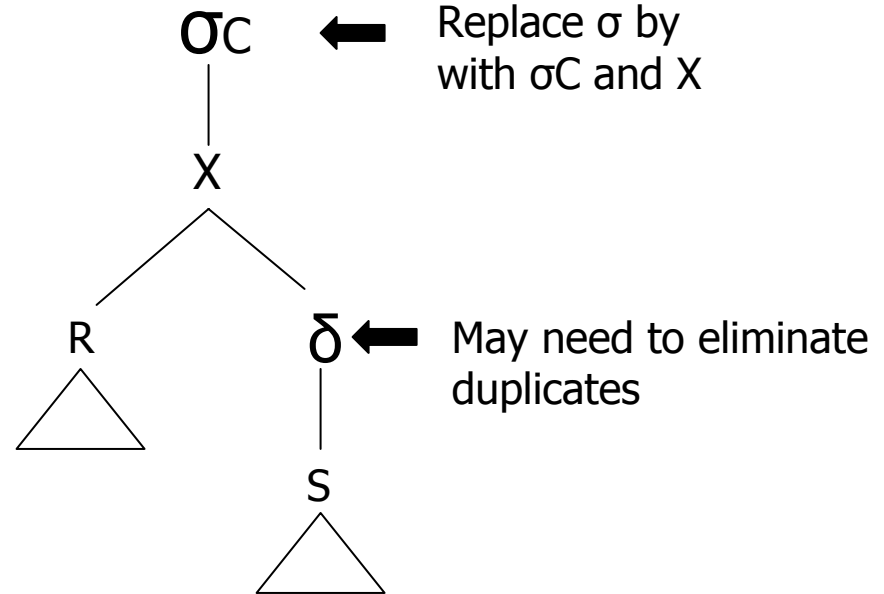
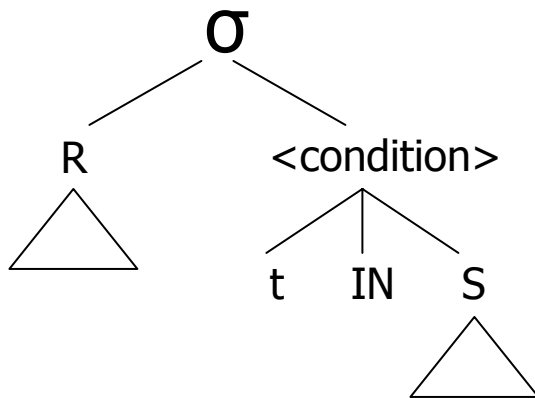




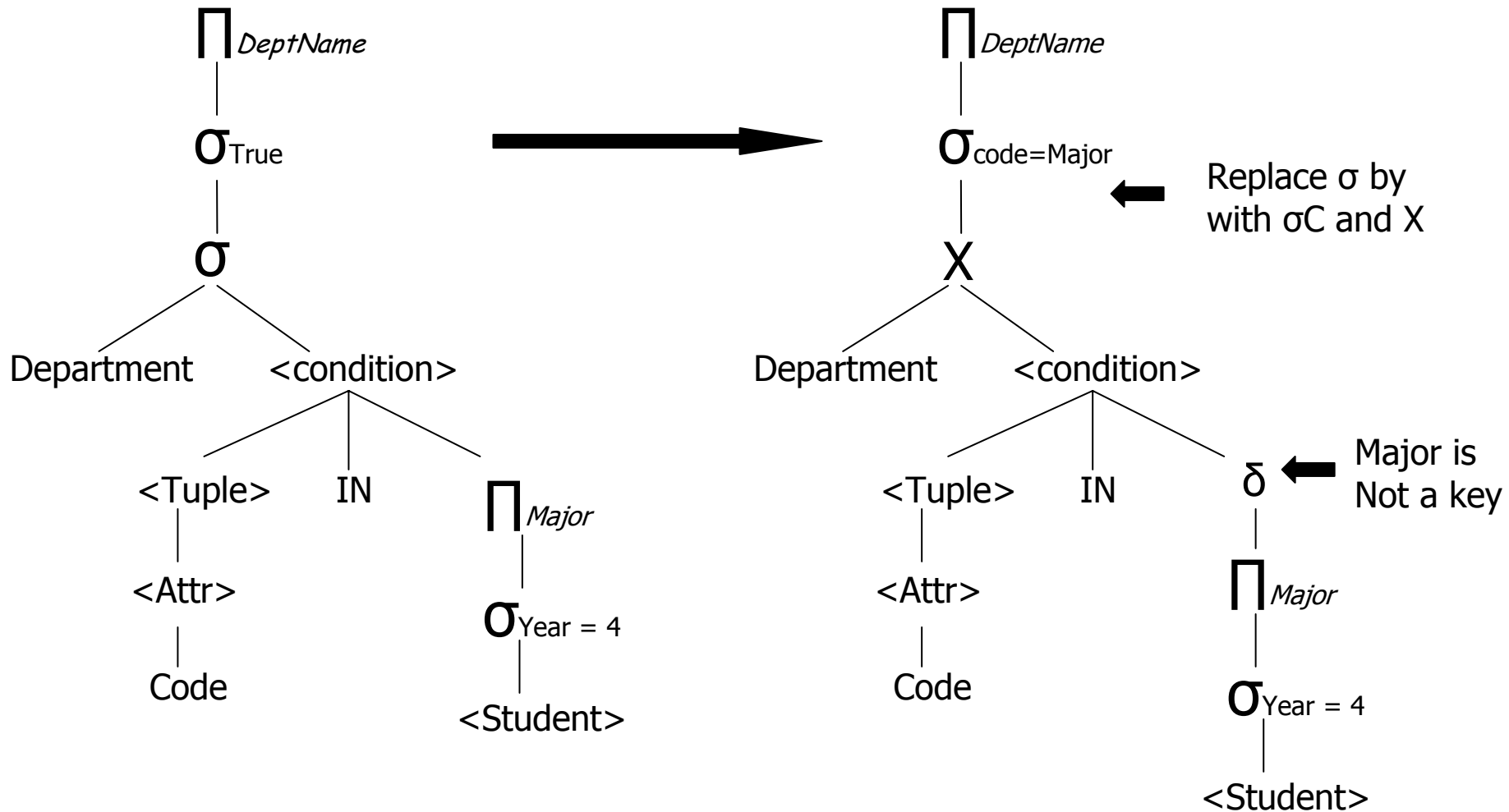
-- Converting Nested Parse Trees to Logical Query Trees

- 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 $\langle \text{Condition} \rangle$ of the form $t \text{ IN } S$. (S is uncorrelated subquery.)
 - Replace $\langle \text{Condition} \rangle$ by the tree that is expression for S .
 - May require applying duplicate elimination if expression has duplicates.
 - 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 .
 - 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





-- Correlated Nested Subqueries

- Translating correlated subqueries is more difficult because the result of the subquery depends on a value defined outside the query itself.
- In general, 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)
```



- 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 often 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 looking at 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 (\times), join (\bowtie), set and bag union (U_S and U_B), and set and bag intersection (\cap_S and \cap_B) are all associative and commutative.

Commutative

$$R \times S = S \times R$$

$$R \bowtie S = S \bowtie R$$

$$R \cup S = S \cup R$$

$$R \cap S = S \cap R$$

Associative

$$(R \times S) \times T = S \times (R \times T)$$

$$(R \bowtie S) \bowtie T = S \bowtie (R \bowtie T)$$

$$(R \cup S) \cup T = S \cup (R \cup T)$$

$$(R \cap S) \cap T = S \cap (R \cap T)$$



-- Laws Involving Selection ...

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

$$\sigma_{c_1 \text{ AND } c_2}(R) = \sigma_{c_1}(\sigma_{c_2}(R))$$

$$\sigma_{c_1 \text{ OR } c_2}(R) = (\sigma_{c_1}(R)) \cup_S (\sigma_{c_2}(R))$$

- Notes:

1. Second law only works if R is a set.
2. The ordering that selections are applied in first law is flexible:

$$\sigma_{c_1 \text{ AND } c_2}(R) = \sigma_{c_2}(\sigma_{c_1}(R))$$

3. In general, we may evaluate selection operators in any order.



... -- Laws Involving Selection ...

- Pushing selections through binary operations is possible and often results in much more efficient logical query trees because a selection reduces the size of the relation.

- Selection is pushed through both arguments for union:

$$\sigma_c(R \cup S) = \sigma_c(R) \cup \sigma_c(S)$$

- Selection is pushed to the first argument and optionally the second for difference:

$$\sigma_c(R - S) = \sigma_c(R) - S$$

$$\sigma_c(R - S) = \sigma_c(R) - \sigma_c(S)$$



... -- Laws Involving Selection ...

- All other operators require selection to be pushed to only one of the arguments. For joins, may not be able to push selection to both if argument does not have attributes selection requires.

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

$$\sigma_C(R \cap S) = \sigma_C(R) \cap S$$

$$\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$$

$$\sigma_C(R \bowtie_D S) = \sigma_C(R) \bowtie_D S$$

- Notes:

1. Laws shown are only pushing to one of the arguments.
2. D is a condition for the join. (Not a natural join.)
3. Assumed that R has all attributes of C .
4. Selection can sometimes be applied to both operands.



... -- Laws Involving Selection

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

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

- Selection and theta join can also be combined:

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



-- 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.



... -- Laws Involving Projection ...

- Laws for pushing projections with joins:

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

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

$$\pi_L(R \bowtie_D S) = \pi_L((\pi_M(R) \bowtie_D \pi_N(S)))$$

- Notes:

- L is a set of attributes to be projected. M is the list of all attributes of R that are either join attributes or are attributes of L . N is the list of attributes of S that are either join or attributes or attributes of L . D is a condition for the join.



... -- Laws Involving Projection ...

- Laws for pushing projections with set operations.
- Projection can be performed entirely before bag union.

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

- Projections can not be pushed below set unions or either the set or bag versions of intersection and difference.
- Projection can be pushed below selection as long as we also keep all attributes needed for the selection ($\mathbf{M} = \mathbf{L} \cup \mathbf{attr}(\mathbf{C})$).

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



... -- Laws Involving Projection ...

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

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

- Notes:
 - Set M must be a superset of set L .



-- Laws Involving Join ...

- We have previously seen these important rules about joins:
 1. Joins are commutative and associative.
 2. Selection can be distributed into joins.
 3. Projection can be distributed into joins.



... -- Laws Involving Duplicate Elimination ...

- The duplicate elimination operator (δ) can be pushed through many operators.
- First, realize that $\delta(R) = R$ occurs when R has no duplicates:
 - R may be a stored relation with a primary key.
 - R may be the result after a grouping operation.
- Laws for pushing duplicate elimination operator (δ):

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

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

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

$$\delta(\sigma_C(R)) = \sigma_C(\delta(R))$$



... -- Laws Involving Duplicate Elimination

- The duplicate elimination operator (δ) can also be pushed through bag intersection, but not across union, difference, or projection in general.

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



-- 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).



-- Rules of Heuristic Query Optimization

- Deconstruct conjunctive selections into a sequence of single selection operations.
- Move selection operations down the query tree for the earliest possible execution.
- Execute first those selection and join operations that will produce the smallest relations.
- Replace Cartesian product operations that are followed by a selection condition by join operations.
- Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed.
- Identify those subtrees whose operations can be pipelined, and execute them using pipelining.



-- Parse Tree to Logical Tree: Example 1

SELECT Name FROM Student WHERE Major = 'ICS'



Π_{Name}



$\sigma_{Major="ICS"}$



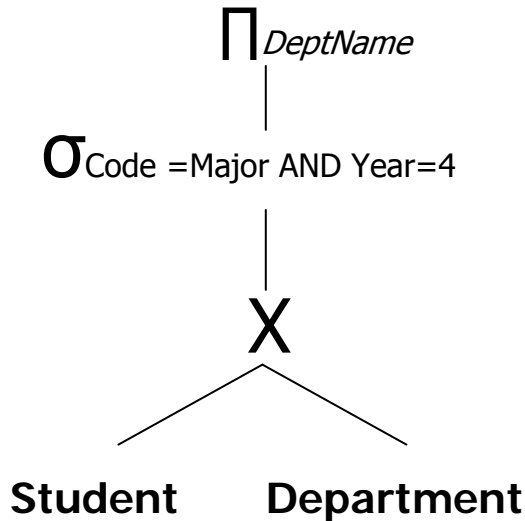
Students

$\Pi_{Name}(\sigma_{Major="ICS"}(Student))$

-- Parse Tree to Logical Tree: Example 1

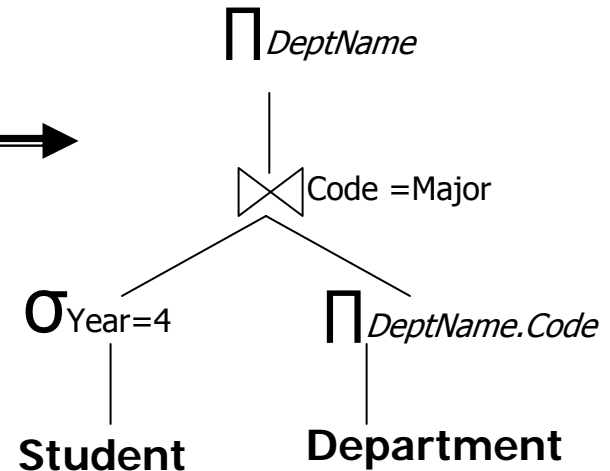
```

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



Optimizations

- push selection down
- push projection down
- merge selection and cross-product



Original:

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

Optimized:

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

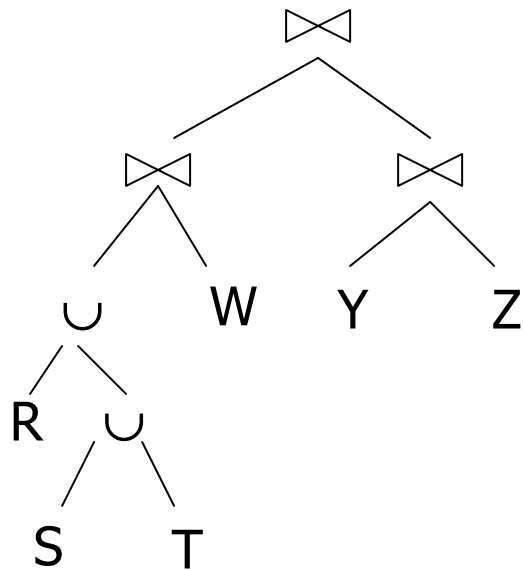


-- Canonical Logical Query Trees

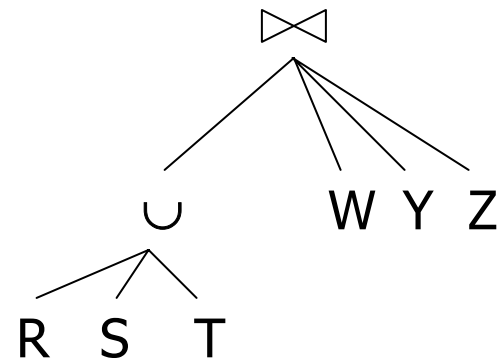
- Most parsers do not produce trees which have unlimited # of children because most operations are unary or binary.
 - However, associative and commutative operators could be considered as having many operands as the order is irrelevant.
- This is especially important for joins as the order of joins may make a significant difference in the performance of the query.
- 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 will make it more convenient and obvious that the operands can be combined in any order.

-- Canonical Logical Query Tree: Example

Original Query Tree



Canonical Query Tree





--Physical Query Plan

- A *physical query plan* is derived from a logical query plan by:
 - Selecting an order and grouping for operations like joins, unions, and intersections.
 - Deciding on an algorithm for each operator in the logical query plan.
 - e.g. Nested-loop join, sort join or hash join
 - Adding additional operators to the logical query tree such as sorting and scanning that are not present in the logical plan.
 - Determining how arguments are passed from one operator to the next. Involves deciding between pipelining and materialization.
- Whether we perform cost-based or heuristic optimization, we eventually must arrive at a physical query tree that can be executed by the evaluator.



-- Heuristic versus Cost Optimization

- In order 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.



-- Estimating Operation Cost ...

- In order 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!
- Thus, it is important to be able to estimate the cost of a query plan without executing it based on statistics and general formulas.



... -- Estimating Operation Cost

- 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 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 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:
 - 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.
 - 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.
 - Not equals: $S = \sigma_{a \neq v}(R)$ - $sf = 1$ $T(S) = T(R)$
 - Reason: Assume almost all tuples satisfy the condition.



... -- Estimating Selection Sizes ...

- 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 Selection Sizes ...

- 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:

$$n(1-(1-m1/n)(1-m2/n))$$

- $m1$ and $m2$ are the # of tuples that satisfy $C1$ and $C2$ respectively.
- n is the number of tuples of R (i.e. $T(R)$).
- $1-m1/n$ and $1-m2/n$ are the fraction of tuples that do not satisfy $C1$ and $C2$ respectively. The product of these numbers is the fraction that do not satisfy either condition.



... -- Estimating Selection Sizes

- 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 Join Sizes ...

- We will study only 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:
 - The two sets are disjoint. Result size = 0.
 - Y may be a foreign key of R joining to a primary key of S . Result size in this case is $T(R)$.
 - 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

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

$$(T(R) * T(S)) / (\text{MAX}(V(R, Y), V(S, Y)))$$

- 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 S 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 $V(R,a)$

- The database will keep statistics on the number of distinct values for each attribute a in each relation R , which we denote as $V(R,a)$.
- However, 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 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.



- 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:
 - The logical operations chosen to implement the query (the logical query plan).
 - The sizes of the intermediate results of operations.
 - The physical operators selected.
 - The ordering of similar operations such as joins.
 - The method of passing arguments from one operator to another (pipelining versus materialization).



-- 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 ...

- 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.



... -- Using Size Estimates in Cost-based Optimization

- Heuristic Selection
 - Use the same approach to select physical plan that is generally used for selecting a logical plan.
- Branch-and-bound
 - Begin with heuristic to find a plan. Let the cost of this plan be C .
 - Then consider other plans for subqueries and ignore if the cost of the subquery is higher than C and so on.
- Hill climbing
 - Start with heuristically selected plan. Let the cost of this plan be C .
 - Change the plan slightly. If the new plan is better than C then consider this plan.
 - Repeat the process again till certain threshold is reached.
- Dynamic programming
- Selinger-Style Optimization
 - Like Dynamic programming but keeps the least cost plan plus some more.

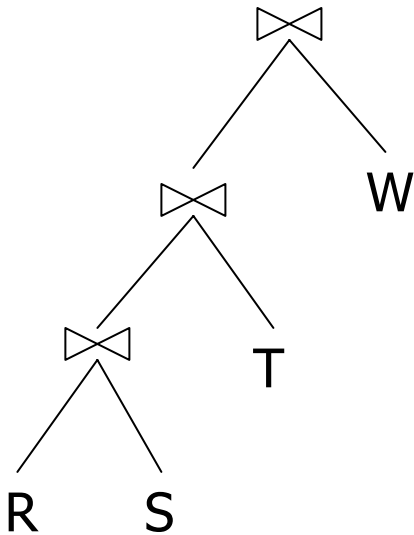


-- Selecting a Join Order

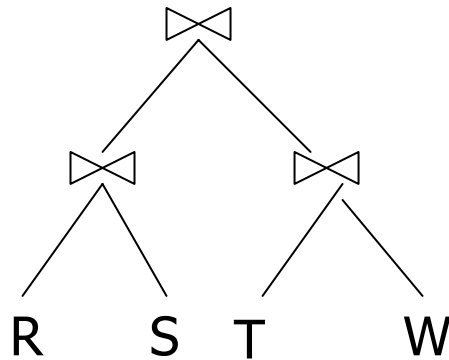
- Since joins are the most costly operation, determining the best possible join order will result in more efficient queries.
- Selecting a join order is most important if we are performing a join of three or more relations. However, a join of two relations can be evaluated in two different ways depending 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

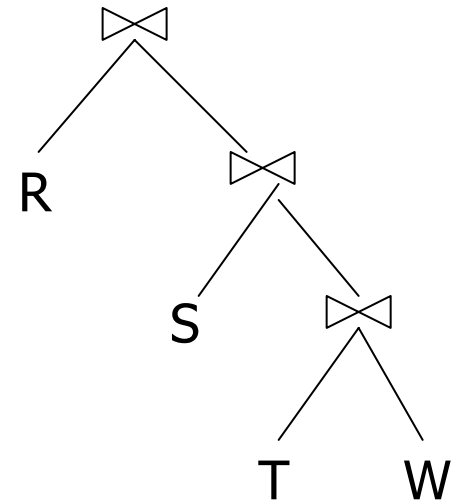
Left-deep join tree



Balanced join tree



Right-deep join tree





-- 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.



-- 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.



-- Other Operators

- Determining the algorithms to select for the other operators is similar. This includes the set operators.
- Projection is always implemented as a table scan, so no decisions must be made for that operator.



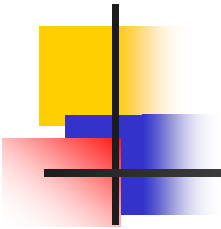
-- Pipelining versus Materialization

- The final decision to be made is how results are passed from one operator to the next.
- There are two choices:
 - ***Materialization*** - the entire operator result is computed before any tuples are sent to the next operator.
 - ***Pipelining*** - the operator result is provided to the next operator a tuple-at-a-time as it is calculated (using iterators).
- Materialization reduces parallelism and requires intermediate results to be stored on disk. Pipelining increases parallelism and reduces disk I/Os, but may be less efficient if many operators compete for the same resources or the pipelined algorithms are less efficient than regular algorithms.



-- Pipelining Operators

- Pipelining unary operators like selection and projection is easy as they operate a tuple-at-a-time.
- Binary operators such as join can also be pipelined. However, there is a potential that pipelining may reduce the performance.
 - For example, consider pipelining the join of $R(a,b)$, $S(b,c)$, and $U(c,d)$ with a fixed number of memory buffers M .
 - One solution is to materialize the join on R and S , then join the result with U using a regular hash-join.
 - The other approach is to perform both hash joins simultaneously, but this would reduce the amount of memory available to each individual operation.
 - Potentially, pipelining would reduce the performance for certain relation sizes.



To be continued