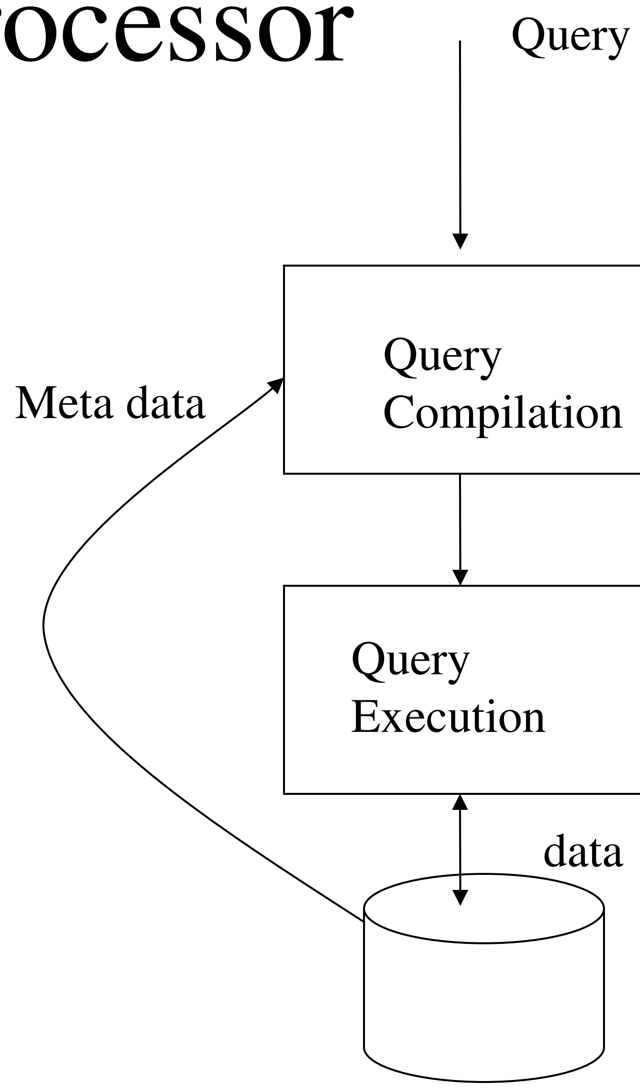
The Query Compiler

Query Processor

- Query compilation
  - Chapter 7
- Query execution
  - Chapter 6



Query Execution

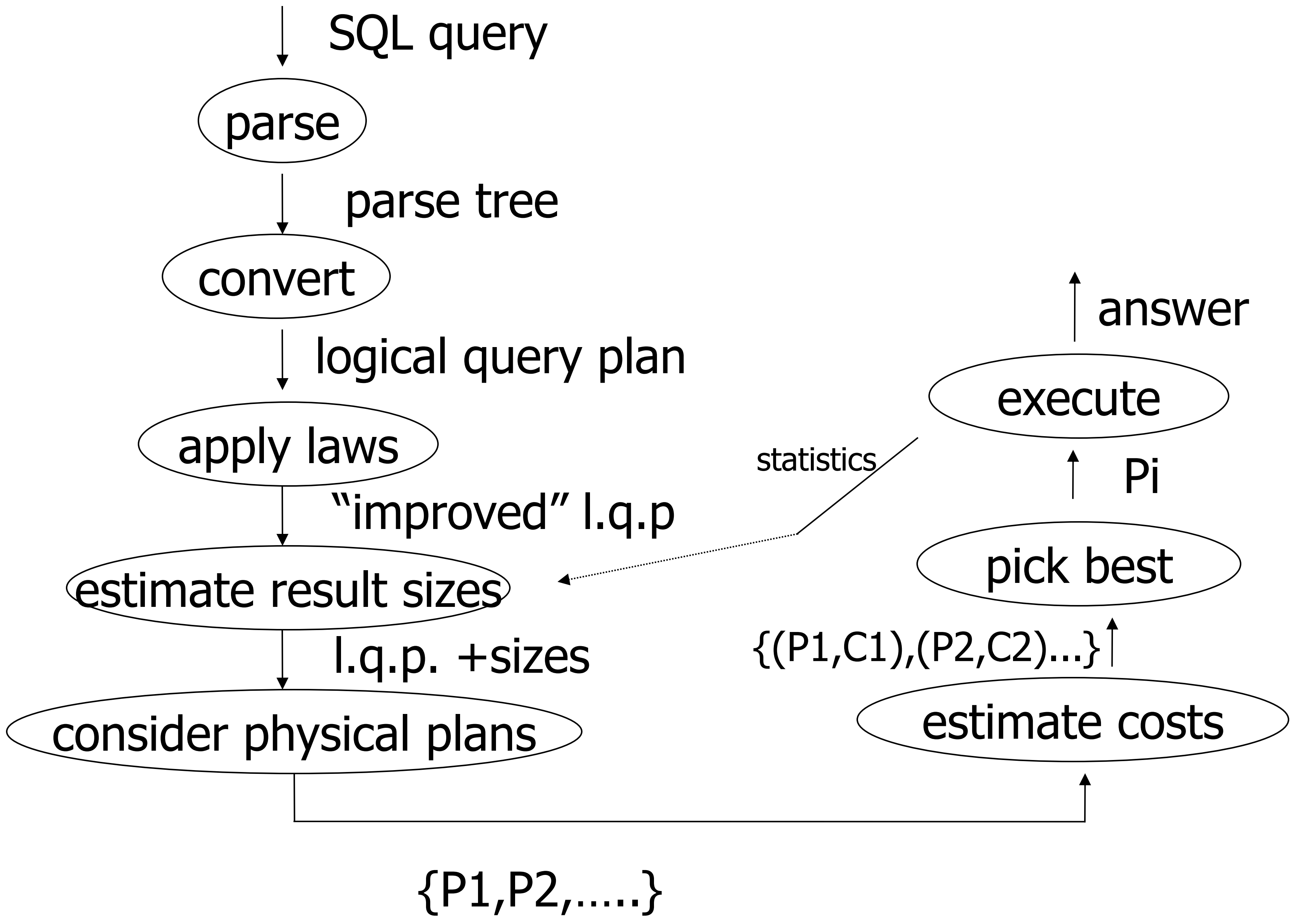
- Execution of relational algebra operations
  - Union
  - Intersection
  - Difference
  - Selection
  - Projection
  - Joins
  - sorting

Query compliation Outline

- Parsing
- Algebraic laws for improving query plans
- From parse trees to logical query plans
- Estimating the cost of operations
- Cost-based plan selection
- Choosing an order for joins
- Completing the physical query plan selection

Overview of Query Compilation

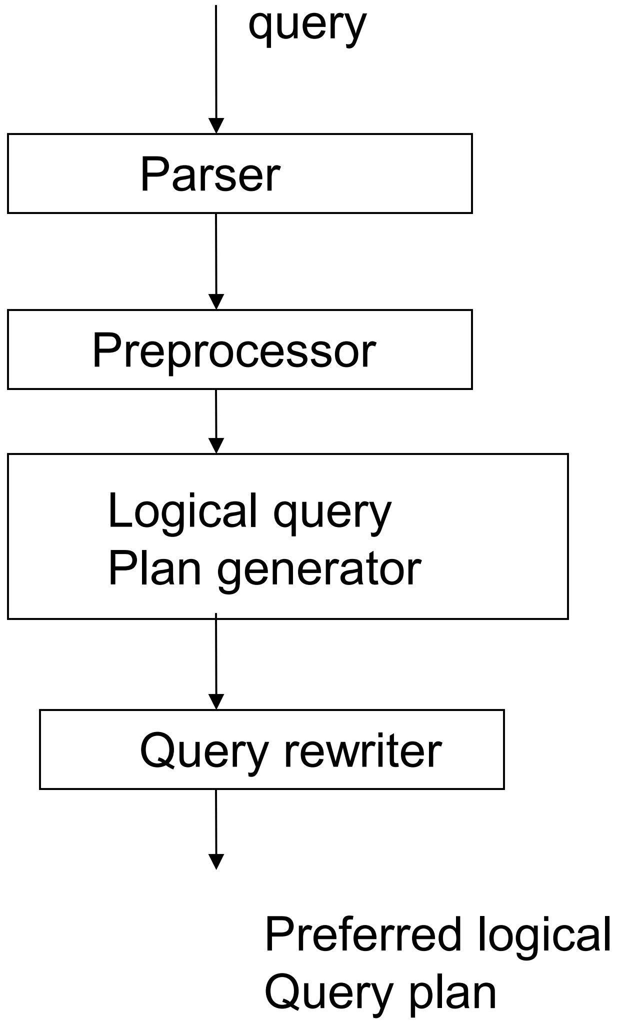
- SQL query → query expression tree →logical query plan tree → physical query plan tree
- Three major steps
  - Parsing
    - Query and its structure is constructed
  - Query rewrite (Selecting a logical query plan)
    - Parse tree is converted to initial query plan
    - Algebraic representation of a query
  - Physical plan generation (Selecting a physical plan)
    - Selecting algorithms for each operations
    - Selecting the order of operations
- Query rewrite and physical plan generation are called query optimizer.
- Choice depends on meta data


SQL query
parse
parse tree
convert
logical query plan
apply laws
"improved" l.q.p
estimate result sizes
l.q.p. +sizes
consider physical plans
{P1,P2,…..}
estimate costs
{(P1,C1),(P2,C2)...}
pick best
Pi
execute
answer
statistics

Outline

- **Parsing**
- Algebraic laws for improving query plan
- From Parse Trees to Logical Query Plans
- Estimating the cost of operations
- Cost-based plan selection
- Choosing the order of joins
- Completing the physical query plan selection

Query to Logical Query Plan



Parsing

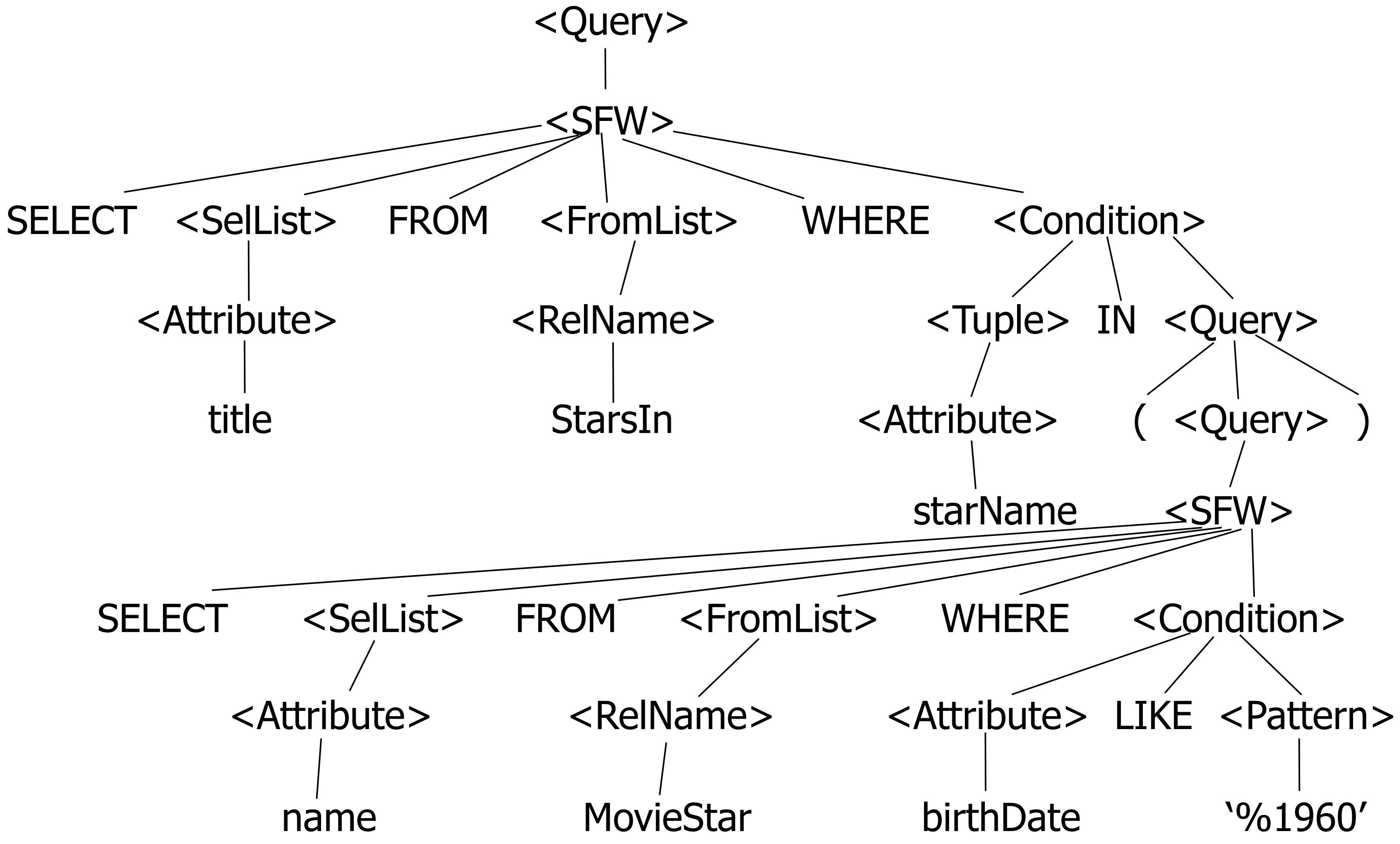
- Syntax analysis of parse tree
  - SQL is converted into parse tree where nodes correspond to
    - Atoms:
      - key words, attributes, constants, parentheses, operators
    - Syntactic categories:
      - names of sub parts < >
        - <SFW> or <condition>
  - If the node is an atom, there are no children
  - If the node is a syntactic category, the children are described by one of the rules of the grammar.
- Parsing turns query into parse tree
  - Subject of compilation

Example: SQL query

```
SELECT title
FROM StarsIn
WHERE starName IN (
        SELECT name
        FROM MovieStar
        WHERE birthdate LIKE ‘%1960’
);
```

(Find the movies with stars born in 1960)

Example: Parse Tree



Preprocessor

- Responsible for semantic checking
  - Check relation uses
  - Check and resolve attribute uses
  - Check types
- If the parse tree satisfies all the above tests, it is valid
- Otherwise, processing stops.

Outline

- Parsing
- **Algebraic laws for improving query plan**
- From Parse Trees to Logical Query Plans
- Estimating the cost of operations
- Cost-based plan selection
- Choosing the order of joins
- Completing the physical query plan selection

Algebraic laws for Improving Query Plans

- Commutative and associative laws
  - Commutative law
    - Order does not matter (+,* are associative)
  - Associative law
    - Group the operators either from left to right or right to left.

Rules: Natural joins & cross products & union

$R \bowtie S = S \bowtie R$

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

$R \times S = S \times R$

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

$R \cup S = S \cup R$

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

Similarly for intersection also

Laws Involving Selection

$\sigma_{p1 \wedge p2}(R) = \quad \sigma_{p1} [\sigma_{p2} (R)]$

$\sigma_{p1 v p2}(R) = \quad [\sigma_{p1} (R)] \cup [\sigma_{p2} (R)]$

- Order can be changed
- The second law do not work for bags

Selection and Union

- For a union, the selection must be pushed to both arguments.
- For a difference, the selection must be pushed to the first argument and optionally pushed to the second.
- For other operators, it is only required that the selection be pushed to one argument.

Rules σ, U combined:

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

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

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

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

Pushing Selections

- Powerful features of query optimizer
  - Replacing the left side of one of the rules by its right side

Laws Involving projections

- Projections, like selections, can be pushed down through many operators.
- Pushing selections reduces number of tuples.
- Pushing projections keeps the same number of tuples.
- 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.

Rules: π, σ combined

Let x = subset of R attributes

z = attributes in predicate

P (subset of R attributes)

$\pi_x[\sigma_{p}(R)] =$

Rules: π, σ combined

Let x = subset of R attributes

z = attributes in predicate

P (subset of R attributes)

$\pi_x[\sigma_{p}(R)] = \quad \{\sigma_p[\pi_x(R)]\}$

Rules: π,σ combined

Let x = subset of R attributes

z = attributes in predicate P (subset of R attributes)

$$\pi_x[\sigma_p(R)] = \pi_x\{\sigma_p[\overset{\pi_{xz}}{\cancel{\pi_x}}(R)]\}$$

Rules: π, $\bowtie$ combined

Let x = subset of R attributes

y = subset of S attributes

z = intersection of R,S attributes

$\pi_{xy}(R \bowtie S) =$

Rules: π, $\bowtie$ combined

Let x = subset of R attributes
y = subset of S attributes
z = intersection of R,S attributes

$$\pi_{xy}(R \bowtie S) =$$

$$\pi_{xy}\{[\pi_{xz}(R)] \bowtie [\pi_{yz}(S)]\}$$

$$\pi_{xy}\left\{\sigma_p\,(R \bowtie S)\right\} =$$

$$\pi_{xy}\{\sigma_p\,(R \bowtie S)\} =$$

$$\pi_{xy}\{\sigma_p\,[\pi_{xz'}(R) \bowtie \pi_{yz'}(S)]\}$$

$z' = z \cup \{$attributes used in P $\}$

<u>Rules</u> for σ, π combined with X

similar...

e.g., σ_p (R X S) = ?

Laws Involving Duplicate Elimination

- $\delta(R \times S) = \delta(R) \times \delta(S)$
- $\delta(R \text{ NJ } S) = \delta(R) \text{ NJ } \delta(S)$
- $\delta(\sigma_C(R)) = \sigma_C(\delta(R))$

Laws Involving Grouping and Aggregation

- γ absorbs δ
  - $\delta(\gamma_L(R)) = \gamma_L(R)$
- $\gamma_L(R)=\gamma_L(\pi_M(R))$

Outline

- Parsing
- Algebraic laws for improving query plan
- **From Parse Trees to Logical Query Plans**
- Estimating the cost of operations
- Cost-based plan selection
- Choosing the order of joins
- Completing the physical query plan selection

From Parse Trees to Logical Query Plan

- First Step:
  - Replace the nodes and structures of parse tree, by an operators of relational algebra
- Second step
  - Take the relational algebra expression produced in the first step and turn it to an expression with the preferred logical query plan.

Conversion into relational algebra

- Input is SFW query and output is relational algebra expression
  - Product of all relations in the From list
  - Selection σ_C where C is the <condition> expression which is the argument of π_L, where L is the selection list.

Removing the subqueries

Improving the Logical query plan

- Push the selections as far as they go
- Push the projections or add new projections
- Remove the duplicate elimination or move into more convenient position.
- Certain selections can be combined with a product below to turn the pair of operations into an equijoin.

Grouping Associative/Commutative Operators

- Conventional parsers do not produce trees with many children.
  - Unary/ binary
- Associative and commutative operators have many operands.
  - Opportunity to reorder
- Group associative and commutative operators.

Outline

- Parsing
- Algebraic laws for improving query plan
- From Parse Trees to Logical Query Plans
- **Estimating the cost of operations**
- Cost-based plan selection
- Choosing the order of joins
- Completing the physical query plan selection

Estimating the cost of Operations

- We enumerate the possible physical plans for a given logical plan
- For each physical plan, we select
  - An order and grouping for associative-and-commutative operations like joins, unions and intersections.
  - An algorithm for each operator
    - Nested loop or a hash join
  - Additional operators
    - Scanning, sorting and so on
  - How the arguments are passed from one operator to next by storing the intermediate result on disk or by using iterators passing one main memory buffer at a time.
- Estimation of costs is an important step

Estimating the Sizes of Intermediate Relations

- Physical plan cost= # of disk I/Os
  - In some cases processor time and communication time is also important.
- Rules for estimating the number of tuples in an intermediate relation.
  - Give accurate estimates
  - Are easy to compute
  - Are logically consistent
    - Independent of the order of computation.

Notation

- B(R) → # of blocks needed to hold all tuples of R
- T(R) → number of tuples of R
- V(R,a) → number of district values of “a” in R.

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

A: 20 byte string

B: 4 byte integer

C: 8 byte date

D: 5 byte string

T(R) = 5

V(R,A) = 3 V(R,C) = 5

V(R,B) = 1 V(R,D) = 4

Estimating the size of projection

- Reduces the size of a tuple
- Some times size can be increased due to addition of attributes.

Estimating the Size of Selection

Example

R

A	B	C	D
cat	1	10	a
cat	1	20	b
dog	1	30	a
dog	1	40	c
bat	1	50	d

$V(R,A)=3$

$V(R,B)=1$

$V(R,C)=5$

$V(R,D)=4$

$W = \sigma_{z=val}(R) \quad T(W) = \dfrac{T(R)}{V(R,Z)}$

Selection

- Size estimate is more problematic when the selection involves inequality comparison.
  - Example $S=\sigma_{a<10}(R)$
  - We assume that typical inequality will return one third of tuples
  - $T(S)=T(R)/3$.
  - For not equal, $T(S)=T(R)$
- For OR
  - $S=\sigma_{c1}OR_{c2}(R)$
  - If R has n tuples, m1 of which satisfy C1 and m2 will satisfy C2, the estimate is equal to
  - $n(1-(1-m1/n)(1-m2/n)$

Estimating the Size of Join

- We consider only natural join
- Estimate for other joins has the following outline
  - Equijoin can be computed similar to natural join.
  - Estimate for Theta join can be computes as if there was a selection following a product.
    - Number of tuples= product
    - Equality is similar to natural join
    - Inequality comparison is similar to selection.

Natural Join

- R(X,Y) $\bowtie$ S(Y,Z)
  - If two relations have disjoint sets of Y values T(R natural join S)=0
  - Y might be the key of S and foreign key of R
    - T(R $\bowtie$ S)=T(R)
- All the tuples of R and S could have the same Y-value
  - =T(R)T(S)
- For above common situations, we make the following assumptions.
  - Containment of value sets: V(R,Y) <= V(S,Y), every Y value of R will be a Y-value of S.
    - This can be violated, if Y is not a key in S and not a foreign key in R
  - Preservation of value sets: If R is joined with another relation and attribute A is not a join attribute, we do not lose value from the set of values of other attributes, V(S join R, A)=V(R,A)
    - This can be violated if there are dangling tuples in R

Natural Join

- But, for most common situations we assume the containment of value sets and preservation of value sets.

- If the above two rules are violated
  - T(R NATURAL JOIN S) = (T(R) T(S))/max(V(R,Y),V(S,Y))
    - Let V(R,Y) <= V(S,Y). Then every tuple t of R has a chance 1/V(S,Y) of joining with a given tuple of S.
    - Since there are T(S) tuples in S, the expected number of tuples that t joins with is T(S)/V(S,Y)
    - Since there are T(R) tuples of R, the estimated size of R JOIN S= (T(R)T(S))/max(V(R,Y),V(S,Y)).

Example

- Estimate R JOIN S JOIN U
  - R(a,b): T(R)=1000, V(R,b)=20
  - S(b,c): T(S)=2000, V(S,b)=50, V(S,c)=100
  - U(c,d): T(U)=5000, V(U,c)=500
- Strategy 1:
  - Estimate R JOIN S which is equal to = (10,000*2000/max (50, 20) =40,000
  - Note: Due to preservation of value sets assumption V(R JOIN S, c)=V(S,c)=100
  - Estimate (R JOIN S) JOIN T(U)= 40,000*5000/max(100,500)=400,000
- Strategy 2
  - Estimate S JOIN U which is equal to = (2,000*5000/max (500, 100) =20,000
  - Note: Due to preservation of value sets assumption V(S JOIN U, b)=V(S,b)=50
  - Estimate (R JOIN (S JOIN T(U))= 1,000*20,000/max(20,50)=400,000

Natural joins with multiple join attributes

- $R(x, y1, y2) \bowtie S(y1, y2, z)$

- $T(R)T(S)/\max(V(R,y1), V(S,y1)) \max(V(R,y2), V(S,y2))$

Join of many relations

- Start with the product of the number of tuples in each relation. Then for each attribute A appearing at least twice, divide by all but the least of the V(R,A)'s.

- Note:
  - The size of join result is independent of the order.

Estimating the Sizes of Other Operations

- Projections do not change the number of tuples
- Products= product of number of tuples
- Union
  - Bag: sum of sizes
  - Set: sum of sizes or equal to larger relation
    - Middle is chosen
- Intersection
  - 0 to size of smaller relation
  - Average of extremes.
- Difference
  - We can have T(R) an T(R) – T(S) tuples. Average of estimate: T(R) – 1/2T(S).

Estimating size of duplicate elimination

- Duplicate elimination
  - Size is equal to size of R (no duplicates)
  - Number of distinct tuples
  - Min(T(R)/2, product of all the V(R,a))
- Grouping and aggregation
  - Similar to duplicate elimination.

Outline

- Parsing
- Algebraic laws for improving query plan
- From Parse Trees to Logical Query Plans
- Estimating the cost of operations
- **Cost-based plan selection**
- Choosing the order of joins
- Completing the physical query plan selection

Cost-based Plan Selection

- Cost of expression is approximated to number of disk I/Os, which in turn are influenced by
  - Logical operators chosen to implement the query, a matter decided when we choose the logical query plan.
  - Size of intermediate relations
    - We can use the estimates
  - Physical operators used to implement the logical operators
  - Ordering of joins
  - The method of passing arguments from one physical operator to the next.

Obtaining Estimates for Size Operators

- DBMS can compute the following by scanning R
  - T(R), V(R, a) for each a
- DBMS computes histogram of the values for each attribute (number of tuples for each value)
  - Equal width
    - A width w is chosen. Counts are provided of the number of tuples with values v such that $v_0 <= v <= v_0 + w$.
  - Equal height
    - Common percentiles.
  - Most frequent values
    - List the most common values and their occurrences.
- With histograms, the size of the joins can be estimated more accurately.
  - If the histogram does not exist, we follow other methods.

Example: Histogram-based estimation

Example

Incremental Computation of Statistics

- Statistics are updated whenever a table is updated.
- T(R): add one whenever a tuple is inserted.
- If there is a B-tree index, we can estimate by counting number of blocks (3/4 full).
- If there is an index on "a", we can estimate V(R,a)
  - Increment the value when the tuple is inserted
- If a is a key for R, V(R,a)=R
- **Sampling can be used to estimate V(R,a).**
  - **Depends on number of assumptions.**
    - **Uniform, Zipfian or some other distribution**

Heuristics For Reducing the Cost of Logical query plan

- Pushing selections, projections, duplication elimination.
  - Estimate the cost before and after the transformation
- Example 7.31:

Approaches to Enumerating Physical Plan

- Each possible physical plan is assigned an estimated cost, and the one with the smallest cost is selected.
- Two approaches
  - Top-down:
    - we work down from the root of the tree. For each possible implementation of the operation at the root, we consider each possible way to evaluate its arguments, and compute the cost of each combination, taking the best.
  - Bottom-up:
    - For each sub-expression E of the logical query plan tree, we compute the costs of all possible ways to compute the sub-expression. Similarly, sub expressions of E are computed.
- Not much difference
- Techniques for bottom-up strategy have been developed Selinger-style (System R)

Commonly Used Heuristics

- Greedy heuristic for join
  - Joining the pair of relations whose results is smallest size, and repeat the process.
- Some of the common heuristics
  - If the logical plan calls for $\sigma_{A=c}(R)$, and stored relation R has an index on attribute A, perform indexed scan.
  - If the selection involves A=c and other conditions, we can implement the selection by indexed scan.
  - If an argument of a join has an index on the join attributes, then use an index-join with that relation in the inner loop.
  - If one argument of the join is sorted on the join attributes, prefer a sort join to a hash join.
  - When computing the union or intersection of three or more relations, group the smallest relations first.

Branch and bound plan enumeration

- Uses good heuristics to find a good physical plan for the entire logical plan. Let the cost of this plan be C.
  - When we consider other plans for sub-queries, we can eliminate any plan greater than C.
  - If we construct a query with a cost less than C, we can replace C.
- If C is already small, stop searching for more plans.
- If C is big, keep searching alternative plans.

Hill Climbing

- Start with heuristically selected physical plan
  - Make small changes to the plan, replacing the one method by another method for an operator, reordering joins with commutative and associative laws, to find nearby plans with low cost.

Dynamic and Selinger Style

- Dynamic programming
  - Keep for each expression only the plan of least cost. (We will discuss later)
- Selinger-Style Optimization
  - It keeps the plans of least cost and other plans of higher costs, but produce the result in a sorted order.
  - Improves upon dynamic programming

Outline

- Parsing
- Algebraic laws for improving query plan
- From Parse Trees to Logical Query Plans
- Estimating the cost of operations
- Cost-based plan selection
- **Choosing the order of joins**
- Completing the physical query plan selection

Choosing a order of joins

- Selecting a join order of two or more relations
- One pass join
  - Left argument fits in main memory (build relation) and right argument (probe relation) is accessed from disk
- Nested-loop join: left argument is the relation of outer loop.
- Index join: right argument has index.

Join Trees

- Algorithms work better of left argument is smaller.
- When join involves more than two relations, the number of join trees grows rapidly.
- n relations n! join trees.

Advantages of left-deep Join Trees

- Number of left-deep trees with a given number of leaves is large but not as that large as the number of all trees.
- Left-deep joins interact well with common join algorithms.
  - Nested-loop joins and one pass join algorithms.

Advantage of left deep join trees…

- If one pass joins are used, and the build relation is on the left, the amount of main memory needed at any one time tends to be smaller than if we used a right-deep tree or a bushy tree for the same relations.
- If we use nested-loop joins, implemented by iterators, then we avoid having to construct any intermediate relation more than once.

Example

- Left deep tree
  - Left arguments will be kept in the memory
  - We compute R join S and keep the result in the main memory
    - We need B(R)+ B(R join S) buffers. If R is small, there will be no problem.
    - After computing R join S, the buffers used by R are not needed and can be used hold the result of (R join S) join T.
- Right deep tree
  - Load R into buffers
    - Load S, T and U for join computation.
  - So, load "n-1" relations in the main memory.
- Bushy tree:
  - neigther left deep nor right deep
- There are !n left deep and !n right deep trees
- The number of left deep trees is less than the number of all trees.
  - Refer book
  - The number of trees with 6 leaves is 30,240 of which only 6!=720 are left deep trees and another 720 are right deep trees.

- Iterators require more disk I/Os in right deep join than left deep join.

Left and right deep trees

- Left deep tree
  - At most two of the relations are in main memory at any time
- Right deep tree
  - All relations should be in main memory simultaneously

Dynamic Programming to select a join order

- We have three choices.
  - Consider them all
  - Consider a subset
  - Use the heuristic to pick one.

Consider them all

- Suppose we want to join R1, R2,…,Rn.
- Estimate size of join of these relations.
- Compute the least cost of computing the join of these relations.
  - Estimate the cost of disk I/Os.
- Compute the expression that yields least cost.
  - The expression joins the set of relations in question, with some grouping.
    - We can restrict to left-deep expressions.

Consider them all

- One relation: The formula is R itself, the cost is zero, no intermediate relations.
- Two relations: {Ri, Rj}, easy to compute. Cost is 0, no intermediate relations.
  - Formula is Ri join Rj or Rj join Ri.
  - Estimate: product of sizes of Ri and Rj.
  - Take smallest of Ri and Rj as left argument.
- More than two relations (r relations)
  - Computing r-{R} and joining with R
  - Cost of join is cost for r-{R} + size of the join.
  - The expression of r has the best join expression for r-{R} as the left argument of final join, and R as the right argument.
- We consider all ways to partition r into disjoint sets in R1 and R2. For each subset, we consider the sum of
  - The best costs of R1 and R2
  - The sizes of R1 and R2.
- See the Example in the book.

Dynamic programming with more Detailed Cost Functions

- Selinger style optimization
  - Compute the least cost of joining R1 and R2 by considering best algorithms.
    - One pass, multi-pass
    - Hash join
    - indexing

Greedy Algorithm for selecting join order

- Dynamic programming results into exponential number of calculations as a function of relations joined.
- To reduce the time, use a join order heuristic.
  - Select a left deep tree
  - Take one decision at one time, but never backtrack
  - Among all those relations not yet included in the current tree, chose the relation that yields the least cost.
    - It may not guarantee the optimum
  - Main heuristic:
    - Keep the intermediate relations small.

Completing the physical plan selection

- Use similar techniques for join for union, intersection or any associative/ commutative operations
- So far we have done the following steps
  - Input is a query
  - Initial logical query plan
  - Improved logical query plan with transformations.
  - Selecting join orders
  - More steps are required.

Further Steps to Complete Physical Query Plan

- Selection of algorithms to implement the operations of the query plan.
- Decisions regarding intermediate results will be materialized (create and store in the disk), and when they will be pipelines (create only in main memory)
- Notations for physical plan operators, include the details regarding access methods for stored relations and algorithms for implementation of relational algebra operators.

Outline

- Parsing
- Algebraic laws for improving query plan
- From Parse Trees to Logical Query Plans
- Estimating the cost of operations
- Cost-based plan selection
- Choosing the order of joins
- **Completing the physical query plan selection**

Choosing a selection method

- Choosing a selection operator
  - The cost of table scan algorithm with a filter step is
    - B(R), if the table is clustered
    - T(R), if the table is not clustered
  - The cost of selection operator with equality
    - B(R)/(V(R,a) if the index is clustering
    - T(R)/V(R,a) if index is not clustering
  - The cost of algorithm with an inequality term such as b<20.
    - B(R)/3 of index is clustering
    - T(R)/3 if the index is not clustering

Choosing a Join Method

- If the estimates are unknown or number of buffers is unknown
  - Call one-pass join, hoping that buffer manager has enough buffers. Alternatively, chose a nested loop join, hoping that left argument can not be granted enough buffers.
  - A sort-join is a good choice
    - One argument is already sorted on their join attributes or there are two more joins on the same attribute.
      - ((R(a,b) join S(a,c)) join T(a,d); reuse the sorted result for the next join
  - If there is an index opportunity, use the index join
  - If there is no opportunity to use already-sorted relations or indexes, and amulti-pass join is needed, hashing is a best choice, because the number of passes requires the size of smaller argument.

Pipelining versus materialization

- Naïve way (materialization)
  - Compute the operation after completing the underlying operation.
    - Intermediate result is materialized to disk.
- Pipelining
  - The tuples produced by the operator are pased to next higher-level operator.
  - No disk I/O, but thrashing might occur as several operations are using main memory.

Pipelining Unary Operations

- Both selection and projection are excellent candidates for pipelining.

Pipelining Binary Operations

- The results of binary operations can be pipelined
  - Use the buffer to pass the results to consumer, one block at a time
- See the example in the book.

Notations for Physical Query Plan

- Operators for leaves: Each leaf will be replaced by scan operator.
  - TableScan(R): All blocks holding tuples of R are read in arbitrary order.
  - SortScan(R,L):Tuples of R are read in order, and stored according to the attribute (L) order.
  - IndexScan(R, C): C is the condition Aθc, where θ is a comparison operator. Hashing or Btree are used.
  - IndexSCan(R,A): A is the attribute of R. The relation R is retrieved via an index on R.A.
- Physical operators for selection
  - Replace σc(R) by Filter(C): uses TableScan(L) or SortScan(L) to access R.

Notations..

- Physical sort operators
  - SortScan(R,L): already introduced.
  - Sort(L): Sort R based on L.
- All operations are replaced by suitable physical operators. Other operations can be given designations as follows.
  - Join or grouping
  - Necessary parameters, the condition in theta-join, or list of elements in a grouping.
  - A general strategy for algorithm: sort-based, hash-based, or in some joins, index-based.
  - One-pass, two-pass, or multi pass
  - Anticipated number of buffers the operation require.

Ordering of Physical Plan Operators

- Physical query plan is represented as a tree
  - Break the subtrees a each edge that represents materialization. The subtrees can will be executed one at a time.
  - Execution the operations left to right
    - Perform the preorder traversal of a tree.
  - Execute the nodes of subtree with the network of interators. GetNext calls determine the order of events.
- Result is an executable code.