Overview of Query Processing

- SQL query
  - parse
    - parse tree
  - Query rewriting
    - logical query plan
      - statistics
      - Physical plan generation
        - physical query plan
          - execute
            - result

Query Optimization

Query Execution
SQL query

parse

parse tree

Query rewriting

statistics

logical query plan

Physical plan generation

physical query plan

execute

result

Initial logical plan

Rewrite rules

"Best" logical plan

Logical plan
Query Rewriting

\[ \pi_{B,D} \sigma_{R.A = "c" \land R.C = S.C} \]

We will revisit it towards the end of this lecture
Query rewriting

Best logical query plan

Physical plan generation

Best physical query plan

execute

result
Physical Plan Generation

Best logical plan

$\sigma_{R.A = \text{"c"}}$

$\pi_{B,D}$

Natural join

$\sigma_{R.A = \text{"c"}}$

$S$

$R$

Project

Hash join

Index scan

Table scan

$R$

$S$
Query rewriting

- SQL query → parse → parse tree

Query rewriting

- Statistics → Best logical query plan

Physical plan generation

- Enumerate possible physical plans

- Find the cost of each plan

- Pick plan with minimum cost

- Best physical query plan → execute → result
Physical Plan Generation

Logical Query Plan

P1  P2  ….  Pn

C1  C2  ….  Cn

Pick minimum cost one

Physical plans

Costs
Plans for Query Execution

• Roadmap
  – Path of a SQL query
  – Operator trees
  – Physical Vs Logical plans
  – Plumbing: Materialization Vs pipelining
Modern DBMS Architecture

- Applications
  - SQL
  - DBMS
    - Parser
      - Logical query plan
    - Query Optimizer
      - Physical query plan
    - Query Executor
      - Access method API calls
    - Storage Manager
      - Storage system API calls
      - File system API calls
  - OS
  - Disk(s)
Logical Plans Vs. Physical Plans

Best logical plan

\[ \pi_{B,D} \]

\[ \sigma_{R.A = \text{"c"}} \]

Natural join

\[ \sigma_{R.A = \text{"c"}} \]

\[ S \]

\[ R \]

Project

Hash join

Index scan

Table scan

\[ R \]

\[ S \]
**Operator Plumbing**

- **Materialization**: output of one operator written to disk, next operator reads from the disk
- **Pipelining**: output of one operator directly fed to next operator
Materialization

Materialized here

\[ \pi_{B,D} \]

\[ \sigma_{R.A = "c"} \]

\[ S \]

[Diagram showing a materialization process with a selection operation \( \sigma_{R.A = "c"} \) followed by a projection operation \( \pi_{B,D} \).]
Iterators: Pipelining

Each operator supports:
- Open()
- GetNext()
- Close()
Iterator for Table Scan (R)

Open() {
    /** initialize variables */
    b = first block of R;
    t = first tuple in block b;
}

GetNext() {
    IF (t is past last tuple in block b) {
        set b to next block;
        IF (there is no next block)
            /** no more tuples */
            RETURN EOT;
        ELSE t = first tuple in b;
    }
    /** return current tuple */
    oldt = t;
    set t to next tuple in block b;
    RETURN oldt;
}

Close() {
    /** nothing to be done */
}

Iterator for Select

\[ \sigma_{R.A = "c"} \]

GetNext() {
    LOOP:
    t = Child.GetNext();
    IF (t == EOT) {
        /** no more tuples */
        RETURN EOT;
    }
    ELSE IF (t.A == "c")
        RETURN t;
    ENDLOOP:
}

Open() {
    /** initialize child */
    Child.Open();
}

Close() {
    /** inform child */
    Child.Close();
}
Iterator for Sort

\[ \tau_{R.A} \]

GetNext() {
    IF (more tuples)
        RETURN next tuple in order;
    ELSE RETURN EOT;
}

Open() {
    /** Bulk of the work is here */
    Child.Open();
    Read all tuples from Child and sort them
}

Close() {
    /** inform child */
    Child.Close();
}
Iterator for Tuple Nested Loop Join

Lexp \land Rexp

- TNLJ (conceptually)
  for each \( r \in \text{Lexp} \) do
    for each \( s \in \text{Rexp} \) do
      if \( \text{Lexp.}C = \text{Rexp.}C \), output \( r,s \)
Example 1: Left-Deep Plan

Question: What is the sequence of getNext() calls?
Example 2: Right-Deep Plan

Question: What is the sequence of getNext() calls?
Example

Worked on blackboard
Cost Measure for a Physical Plan

• There are many cost measures
  – Time to completion
  – Number of I/Os (we will see a lot of this)
  – Number of getNext() calls

• Tradeoff: Simplicity of estimation Vs. Accurate estimation of performance as seen by user
Chapter 15

15.1 Physical operators
   - Scan, Sort (Ch. 11.4), Indexes (Ch. 13)

15.2-15.6 Implementing operators + estimating their cost

15.8 Buffer Management

15.9 Parallel Processing
Textbook outline (contd.)

Chapter 16
16.1 Parsing
16.2 Algebraic laws
16.3 Parse tree → logical query plan
16.4 Estimating result sizes
16.5-16.7 Cost based optimization
Background Material

Chapter 5 Relational Algebra
Chapter 6 SQL
Query Processing - In class order

1. **Parse**
   - SQL query
   - Parse tree

2. **Query rewriting**
   - First: A quick look at this
   - 2; 16.1

3. **Statistics**
   - Logical query plan
   - 3; 16.2, 16.3

4. **Physical plan generation**
   - Physical query plan
   - 4; 16.4—16.7

5. **Execute**
   - Result
   - 1; 13, 15
Why do we need Query Rewriting?

- **Pruning** the HUGE space of physical plans
  - Eliminating redundant conditions/operators
  - Rules that will improve performance with very high probability

- **Preprocessing**
  - Getting queries into a form that we know how to handle best

⇒ Reduces optimization time drastically without noticeably affecting quality
Some Query Rewrite Rules

• Transform one logical plan into another
  – Do not use statistics
• Equivalences in relational algebra
• Push-down predicates
• Do projects early
• Avoid cross-products if possible
Equivalences in Relational Algebra

\[ R \bowtie S = S \bowtie R \quad \text{Commutativity} \]
\[ (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \quad \text{Associativity} \]

Also holds for: Cross Products, Union, Intersection

\[ R \times S = S \times R \]
\[ (R \times S) \times T = R \times (S \times T) \]
\[ R \cup S = S \cup R \]
\[ R \cup (S \cup T) = (R \cup S) \cup T \]
Apply Rewrite Rule (1)

\[ \Pi_{B,D} \left[ \sigma_{R.C=S.C} \left[ \sigma_{R.A=\text{"c"}}(R \times S) \right] \right] \]
Rules: Project

Let: $X = \text{set of attributes}$

$Y = \text{set of attributes}$

$XY = X \cup Y$

$\pi_{xy}(R) = \pi_x[\pi_y(R)]$
Rules: $\sigma + \bowtie$ combined

Let $p = \text{predicate with only R attribs}$

$q = \text{predicate with only S attribs}$

$m = \text{predicate with only R,S attribs}$

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

$$\sigma_q (R \bowtie S) = R \bowtie [\sigma_q (S)]$$
Apply Rewrite Rule (2)

\[ \pi_{B,D} \left( \sigma_{R.C = S.C} \sigma_{R.A = "c"} X R \right) S \]

\[ \Pi_{B,D} [ \sigma_{R.C=S.C} [\sigma_{R.A="c"}(R)] X S] \]
Apply Rewrite Rule (3)

\[
\begin{align*}
\pi_{B,D} & \left( \sigma_{R.C = S.C} \left( \sigma_{R.A = "c"} (R) \right) \right) \\
\Pi_{B,D} \left[ \sigma_{R.A = "c"}(R) \right] \bowtie S & \rightarrow \\
\pi_{B,D} & \left( \sigma_{R.A = "c"}(R) \right) \bowtie S
\end{align*}
\]
**Rules:**  \( \sigma + \ltimes \) combined  (continued)

\[
\sigma_{p \land q} (R \ltimes S) = [\sigma_p (R)] \ltimes [\sigma_q (S)]
\]

\[
\sigma_{p \land q \land m} (R \ltimes S) = \sigma_m \left[ (\sigma_p R) \ltimes (\sigma_q S) \right]
\]

\[
\sigma_{p \lor q} (R \ltimes S) = \left[ (\sigma_p R) \ltimes S \right] \cup \left[ R \ltimes (\sigma_q S) \right]
\]
Which are “good” transformations?

- $\sigma_{p_1 \land p_2} (R) \to \sigma_{p_1} [\sigma_{p_2} (R)]$
- $\sigma_p (R \bowtie S) \to [\sigma_p (R)] \bowtie S$
- $R \bowtie S \to S \bowtie R$
- $\pi_x [\sigma_p (R)] \to \pi_x \{ \sigma_p [\pi_{xz} (R)] \}$
Conventional wisdom: do projects early

Example: $R(A, B, C, D, E)$
$P: (A=3) \land (B=\text{“cat”})$

$\pi_E \{\sigma_p (R)\} \quad \text{vs.} \quad \pi_E \{\sigma_p \{\pi_{ABE}(R)\}\}$
But: What if we have A, B indexes?

B = “cat”

Intersect pointers to get
pointers to matching tuples

A=3
Bottom line:

- No transformation is always good
- Some are usually good:
  - Push selections down
  - Avoid cross-products if possible
  - Subqueries → Joins
Avoid Cross Products (if possible)

Select B,D
From R,S,T,U
Where R.A = S.B \land
R.C=T.C \land R.D = U.D

• Which join trees avoid cross-products?
• If you can't avoid cross products, perform them as late as possible
More Query Rewrite Rules

• Transform one logical plan into another
  – Do not use statistics
• Equivalences in relational algebra
• Push-down predicates
• Do projects early
• Avoid cross-products if possible
• Use left-deep trees
• Subqueries → Joins
• Use of constraints, e.g., uniqueness