Database Systems
08 Query Processing

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMVIT endowed chair for Data Management

Last update: June 24, 2019
Announcements/Org

- **#1 Video Recording**
  - Since lecture 03, video/audio recording
  - Link in TeachCenter & TUbe *(video recorder fixed?)*

- **#2 Statistics Exercise 1**
  - *All submissions accepted* *(submitted/draft)*
  - In progress of *grading*, but understaffed

- **#3 Exercise 2**
  - *Submission is crucial* *(modified rule: 1 exercise 2%-50%)*
  - Modified deadline: *May 07 11.59pm*
  - *Please, submit correct file names*
    *(avoid wrong IDs, wrong naming scheme)*

77.4%
Announcements/Org, cont.

- #4 Study Abroad Fair 2019

**Study Abroad Fair**  
**May 22, 2019**

- Your opportunity to find out about exchange programmes and scholarships offered by TU Graz
- Information booths
- Short presentations concerning various study abroad possibilities

[Event Details](tu4u.tugraz.at/go/study-abroad-fair-2019)
**Query Optimization and Query Processing**

**SELECT** * FROM TopScorer
WHERE Count>=4

CREATE VIEW TopScorer AS
SELECT P.Name, Count(*)
FROM Players P, Goals G
WHERE P.Pid=G.Pid
AND G.GOwn=FALSE
GROUP BY P.Name
ORDER BY Count(*) DESC

**WHAT**
Yes, but **HOW to we get there efficiently**

<table>
<thead>
<tr>
<th>Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>James Rodríguez</td>
<td>6</td>
</tr>
<tr>
<td>Thomas Müller</td>
<td>5</td>
</tr>
<tr>
<td>Robin van Persie</td>
<td>4</td>
</tr>
<tr>
<td>Neymar</td>
<td>4</td>
</tr>
</tbody>
</table>

**Goal:** **Basic Understanding of Internal Query Processing**

* Query rewriting and query optimization
* Query processing and physical plan operators
  ➔ **Performance debugging & reuse of concepts and techniques**
  ➔ Overview, detailed techniques discussed in **ADBS**
Agenda

- Query Rewriting and Optimization
- Plan Execution Strategies
- Physical Plan Operators
Query Rewriting and Optimization
Overview Query Optimization

```
SELECT * FROM TopScorer
WHERE Count >= 4
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>James Rodríguez</td>
<td>6</td>
</tr>
<tr>
<td>Thomas Müller</td>
<td>5</td>
</tr>
<tr>
<td>Robin van Persie</td>
<td>4</td>
</tr>
<tr>
<td>Neymar</td>
<td>4</td>
</tr>
</tbody>
</table>

Compile Time

Plan Optimization

Plan Execution

Plan Caching

Runtime
Query Rewrites

- **Query Rewriting**
  - Rewrite query into semantically equivalent form that may be processed more efficiently or give the optimizer more freedom
  - #1 Same query can be expressed differently, prevent hand optimization
  - #2 Complex queries may have redundancy

- **A Simple Example**
  - Catalog meta data: custkey is unique

```sql
SELECT DISTINCT custkey, name
FROM TPCH.Customer
```

- **20+ years of experience on query rewriting**

Standardization and Simplification

- **Normal Forms of Boolean Expressions**
  - **Conjunctive** normal form \((P_{11} \lor \ldots \lor P_{1n}) \land \ldots \land (P_{m1} \lor \ldots \lor P_{mp})\)
  - **Disjunctive** normal form \((P_{11} \land \ldots \land P_{1q}) \lor \ldots \lor (P_{r1} \land \ldots \land P_{rs})\)

- **Transformation Rules for Boolean Expressions**

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commutativity rules</td>
<td>(A \lor B \iff B \lor A) (A \land B \iff B \land A)</td>
</tr>
<tr>
<td>Associativity rules</td>
<td>((A \lor B) \lor C \iff A \lor (B \lor C)) ((A \land B) \land C \iff A \land (B \land C))</td>
</tr>
<tr>
<td>Distributivity rules</td>
<td>(A \lor (B \land C) \iff (A \lor B) \land (A \lor C)) (A \land (B \lor C) \iff (A \land B) \lor (A \land C))</td>
</tr>
<tr>
<td>De Morgan’s rules</td>
<td>(\neg (A \land B) \iff \neg A \lor \neg B) (\neg (A \lor B) \iff \neg A \land \neg B)</td>
</tr>
<tr>
<td>Double-negation rules</td>
<td>(\neg(\neg (A)) \iff A)</td>
</tr>
<tr>
<td>Idempotence rules</td>
<td>(A \lor A \iff A) (A \land A \iff A) (A \lor \neg A \iff \text{TRUE}) (A \land \neg A \iff \text{FALSE}) (A \land (A \lor B) \iff A) (A \lor (A \land B) \iff A) (A \lor \text{FALSE} \iff A) (A \land \text{TRUE} \iff \text{TRUE}) (A \land \text{FALSE} \iff \text{FALSE})</td>
</tr>
</tbody>
</table>
Standardization and Simplification, cont.

- **Elimination of Common Subexpressions**
  - \((A_1=a_{11} \text{ OR } A_1=a_{12}) \text{ AND } (A_1=a_{12} \text{ OR } A_1=a_{11}) \rightarrow A_1=a_{11} \text{ OR } A_1=a_{12}\)

- **Propagation of Constants**
  - \(A \geq B \text{ AND } B = 7 \rightarrow A \geq 7 \text{ AND } B = 7\)

- **Detection of Contradictions**
  - \(A \geq B \text{ AND } B > C \text{ AND } C \geq A \rightarrow A > A \rightarrow \text{FALSE}\)

- **Use of Constraints**
  - A is primary key/unique: \(\pi_A \rightarrow \text{no duplicate elimination necessary}\)
  - Rule \(\text{MAR\_STATUS} = \text{'married'} \rightarrow \text{TAX\_CLASS} \geq 3:\)
    - \(\text{MAR\_STATUS} = \text{'married'} \text{ AND } \text{TAX\_CLASS} = 1) \rightarrow \text{FALSE}\)

- **Elimination of Redundancy**
  - \(R \bowtie R \rightarrow R, \ R \cup R \rightarrow R, \ R - R \rightarrow \emptyset\)
  - \(R \bowtie (\sigma_p R) \rightarrow \sigma_p R, \ \text{RU}(\sigma_p R) \rightarrow R, \ R - (\sigma_p R) \rightarrow \sigma_{\neg p} R\)
  - \((\sigma_{p_1} R) \bowtie (\sigma_{p_2} R) \rightarrow \sigma_{p_1 \lor p_2} R, \ (\sigma_{p_1} R) \cup (\sigma_{p_2} R) \rightarrow \sigma_{p_1 \lor p_2} R\)

**Query Rewriting and Optimization**
Query Unnesting

- **Case 1: Type-A Nesting**
  - Inner block is not correlated and computes an aggregate
  - **Solution:** Compute the aggregate once and insert into outer query

  ```sql
  SELECT OrderNo FROM Order
  WHERE ProdNo =
  (SELECT MAX(ProdNo)
   FROM Product WHERE Price<100)
  \[X\]
  ```

- **Case 2: Type-N Nesting**
  - Inner block is not correlated and returns a set of tuples
  - **Solution:** Transform into a symmetric form (via join)

  ```sql
  SELECT OrderNo FROM Order
  WHERE ProdNo IN
  (SELECT ProdNo
   FROM Product WHERE Price<100)
  ```

  ```sql
  SELECT OrderNo FROM Order O, Product P
  WHERE O.ProdNo = P.ProdNo AND P.Price < 100
  ```
Query Unnesting, cont.

- **Case 3: Type-J Nesting**
  - Un-nesting of correlated sub-queries w/o aggregation

  SELECT OrderNo FROM Order O  
  WHERE ProdNo IN  
  (SELECT ProdNo FROM Project P  
  WHERE P.ProjNo = O.OrderNo  
  AND P.Budget > 100,000)

- **Case 4: Type-JA Nesting**
  - Un-nesting of correlated sub-queries w/ aggregation

  SELECT OrderNo FROM Order O  
  WHERE ProdNo IN  
  (SELECT MAX(ProdNo)  
  FROM Project P  
  WHERE P.ProjNo = O.OrderNo  
  AND P.Budget > 100,000)

  Further un-nesting via case 3 and 2

Selections and Projections

- **Example Transformation Rules**
  1) Grouping of Selections
  \[ \sigma_{x>y} \rightarrow \sigma_{x>y \land p=q} \]
  \[ \sigma_{p=q} \rightarrow R \]
  \[ \pi_A \rightarrow \pi_A \]

  2) Grouping of Projections
  \[ \pi_{A,B} \rightarrow \pi_A \]
  \[ \sigma_{p(R)} \rightarrow \sigma_{p(R)} \]
  \[ \sigma_{A=B} \rightarrow \sigma_{A=B} \]

  3) Pushdown of Selections
  \[ \sigma_{p(R)} \rightarrow \sigma_{p(R)} \]
  \[ \sigma_{A=B} \rightarrow \sigma_{A=B} \]
  \[ \pi_C \rightarrow \pi_C \]

  4) Pushdown of Projections
  \[ \pi_{A,C} \rightarrow \pi_{A,C} \]
  \[ \pi_{B,C} \rightarrow \pi_{B,C} \]

- **Restructuring Algorithm**
  - #1 Split n-ary joins into binary joins
  - #2 Split multi-term selections
  - #3 Push-down selections as far as possible
  - #4 Group adjacent selections again
  - #5 Push-down projections as far as possible

**Input:** Standardized, simplified, and un-nested query graph

**Output:** Restructured query graph
Example Query Restructuring

```
SELECT * FROM TopScorer
WHERE count>=4
AND Pos='FW'

CREATE VIEW TopScorer AS
SELECT P.Name, P.Pos, count(*)
FROM Players P, Goals G
WHERE P.Pid=G.Pid
   AND G.GOwn=FALSE
GROUP BY P.Name, P.Pos
ORDER BY count(*) DESC
```

Additional metadata:
P.Name is unique
Plan Optimization Overview

- **Plan Generation**
  - Selection of **physical access path and plan operators**
  - Selection of **execution order** of plan operators
  - **Input**: logical query plan → **Output**: optimal physical query plan
  - Costs of query optimization should not exceed yielded improvements

- **Different Cost Models**
  - Relies on statistics (cardinalities, selectivities via histograms + estimators)
  - Operator-specific and general-purpose cost models
  
  \[
  C_{out}(T) = \begin{cases} 
  0 & \text{if } T \text{ is a single relation} \\
  |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 
  \end{cases}
  \]
  
  (estimated) (real)

  - **I/O costs** (number of read pages, tuples)
  - **Computation costs** (CPU costs, path lengths)
  - **Memory** (temporary memory requirements)
  - **Beware assumptions of optimizers**
    (no skew, independence, no correlation)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>σModel= ‘Golf’</td>
<td>10</td>
</tr>
<tr>
<td>σMake=‘VW’</td>
<td>1,000</td>
</tr>
<tr>
<td>Cars</td>
<td>10,000</td>
</tr>
</tbody>
</table>
Join Ordering Problem

- **Join Ordering**
  - Given a join query graph, find the optimal join ordering
  - In general, **NP-hard**; but polynomial algorithms exist for special cases

- **Query Types**

- **Search Space**

<table>
<thead>
<tr>
<th></th>
<th>Chain (no CP)</th>
<th>Star (no CP)</th>
<th>Clique / CP (cross product)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>left-deep</td>
<td>zig-zag</td>
<td>left-deep</td>
</tr>
<tr>
<td>n</td>
<td>$2^{n-1}$</td>
<td>$2^{2n-3}$</td>
<td>$2^{n-1}C(n-1)$</td>
</tr>
<tr>
<td>5</td>
<td>16</td>
<td>128</td>
<td>224</td>
</tr>
<tr>
<td>10</td>
<td>512</td>
<td>~131K</td>
<td>~2.4M</td>
</tr>
</tbody>
</table>

C(n) ... Catalan Numbers

Join Order Search Strategies

- **Tradeoff**: Optimal (or good) plan vs compilation time
  
- #1 Naïve Full Enumeration
  - Infeasible for reasonably large queries (long tail up to 1000s of joins)

- #2 Exact Dynamic Programming
  - Guarantees optimal plan, often too expensive (beyond 20 relations)
  - Bottom-up vs top-down approaches

- #3 Greedy / Heuristic Algorithms

- #4 Approximate Algorithms
  - E.g., Genetic algorithms, simulated annealing

- **Example PostgreSQL**
  - Exact optimization (DPSize) if < 12 relations (geqo_threshold)
  - Genetic algorithm for larger queries
  - Join methods: NLJ, SMJ, HJ

---

[Nicolas Bruno, César A. Galindo-Legaria, Milind Joshi: Polynomial heuristics for query optimization. ICDE 2010]
Greedy Join Ordering

- Example
  - Part \(\bowtie\) Lineorder \(\bowtie\) Supplier \(\bowtie\) \(\sigma\)\(\)\(\text{Customer}\) \(\bowtie\) \(\sigma\)\(\)\(\text{Date}\), left-deep plans

<table>
<thead>
<tr>
<th>#</th>
<th>Plan</th>
<th>Costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lineorder (\bowtie) Part</td>
<td>30M</td>
</tr>
<tr>
<td></td>
<td>Lineorder (\bowtie) Supplier</td>
<td>20M</td>
</tr>
<tr>
<td></td>
<td>Lineorder (\bowtie) (\sigma)(\text{Customer})</td>
<td>90K</td>
</tr>
<tr>
<td></td>
<td>Lineorder (\bowtie) (\sigma)(\text{Date})</td>
<td>40K</td>
</tr>
<tr>
<td></td>
<td>Part (\bowtie) Customer</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(Lineorder (\bowtie) (\sigma)(\text{Date})) (\bowtie) Part</td>
<td>150K</td>
</tr>
<tr>
<td></td>
<td>(Lineorder (\bowtie) (\sigma)(\text{Date})) (\bowtie) Supplier</td>
<td>100K</td>
</tr>
<tr>
<td></td>
<td>(Lineorder (\bowtie) (\sigma)(\text{Date})) (\bowtie) (\sigma)(\text{Customer})</td>
<td>75K</td>
</tr>
<tr>
<td>3</td>
<td>(((Lineorder (\bowtie) (\sigma)(\text{Date})) (\bowtie) (\sigma)(\text{Customer})) (\bowtie) Part</td>
<td>120K</td>
</tr>
<tr>
<td></td>
<td>(((Lineorder (\bowtie) (\sigma)(\text{Date})) (\bowtie) (\sigma)(\text{Customer})) (\bowtie) Supplier</td>
<td>105M</td>
</tr>
<tr>
<td>4</td>
<td>(((Lineorder (\bowtie) (\sigma)(\text{Date})) (\bowtie) (\sigma)(\text{Customer})) (\bowtie) Supplier) (\bowtie) Part</td>
<td>135M</td>
</tr>
</tbody>
</table>

Note: Simple \(O(n^2)\) algorithm for left-deep trees; \(O(n^3)\) algorithms for bushy trees existing (e.g., GOO)
Dynamic Programming Join Ordering

- **Exact Enumeration via Dynamic Programming**
  - **#1: Optimal substructure** (Bellman’s Principle of Optimality)
  - **#2: Overlapping subproblems** allow for memoization

→ Approach DPSIZE: Split in independent subproblems (optimal plan per set of quantifiers and interesting properties), solve subproblems, combine solutions

- **Example**

<table>
<thead>
<tr>
<th>{C}</th>
<th>Tbl, IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>{D}</td>
<td>Tbl, IX</td>
</tr>
<tr>
<td>{L}</td>
<td>...</td>
</tr>
<tr>
<td>{P}</td>
<td>...</td>
</tr>
<tr>
<td>{S}</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q1</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>{C}</td>
<td>L⊙C, C⊙L</td>
</tr>
<tr>
<td>{D}</td>
<td>L⊙D, D⊙L</td>
</tr>
<tr>
<td>{L}</td>
<td>L⊙P, P⊙L</td>
</tr>
<tr>
<td>{P}</td>
<td>L⊙S, S⊙L</td>
</tr>
<tr>
<td>{C,D}</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q1+Q1</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>{C,L}</td>
<td>L⊙C, C⊙L</td>
</tr>
<tr>
<td>{D,L}</td>
<td>L⊙D, D⊙L</td>
</tr>
<tr>
<td>{L,P}</td>
<td>L⊙P, P⊙L</td>
</tr>
<tr>
<td>{L,S}</td>
<td>L⊙S, S⊙L</td>
</tr>
<tr>
<td>{C,D}</td>
<td>N/A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q1+Q2, Q2+Q1</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>{C,D,L}</td>
<td>(L⊙C)⊙D, D⊙(L⊙C), (L⊙D)⊙C, C⊙(L⊙D)</td>
</tr>
<tr>
<td>{C,L,P}</td>
<td>(L⊙C)⊙P, P⊙(L⊙C), (P⊙L)⊙C, C⊙(P⊙L)</td>
</tr>
<tr>
<td>{C,L,S}</td>
<td>...</td>
</tr>
<tr>
<td>{D,L,P}</td>
<td>...</td>
</tr>
<tr>
<td>{D,L,S}</td>
<td>...</td>
</tr>
<tr>
<td>{L,P,S}</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q1+Q3, Q2+Q2, Q3+Q1</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>{C,D,L,P}</td>
<td>((L⊙C)⊙D)⊙P, P⊙((L⊙C)⊙D)</td>
</tr>
<tr>
<td>{C,D,L,S}</td>
<td>...</td>
</tr>
<tr>
<td>{C,L,P,S}</td>
<td>...</td>
</tr>
<tr>
<td>{D,L,P,S}</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q1+Q4, Q2+Q3, Q3+Q2, Q4+Q1</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>{C,D,L,P,S}</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q5</th>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>{C,D,L,P,S}</td>
<td>...</td>
</tr>
</tbody>
</table>
Plan Execution Strategies
Overview Query Processing

SELECT * FROM TopScorer
WHERE Count >= 4

Plan Execution Strategies

<table>
<thead>
<tr>
<th>Name</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>James Rodríguez</td>
<td>6</td>
</tr>
<tr>
<td>Thomas Müller</td>
<td>5</td>
</tr>
<tr>
<td>Robin van Persie</td>
<td>4</td>
</tr>
<tr>
<td>Neymar</td>
<td>4</td>
</tr>
</tbody>
</table>

Compile Time

Plan Optimization

Query Rewrites

Query Semantics

Parsing

Runtime

Plan Caching

Plan Execution

QEP

IR

AST/IR
Overview Execution Strategies

- Different execution strategies (processing models) with different pros/cons (e.g., memory requirements, DAGs, efficiency, reuse)

- #1 Iterator Model (mostly row stores)

- #2 Materialized Intermediates (mostly column stores)

- #3 Vectorized (Batched) Execution (row/column stores)

- #4 Query Compilation (row/column stores)

High-level overview, details in ADBS
Plan Execution Strategies

Iterator Model

- **Volcano Iterator Model**
  - Pipelined & no global knowledge
  - Open-Next-Close (ONC) interface
  - Query execution from root node (pull-based)

- **Example $\sigma_{A=7}(R)$**
  ```c
  void open() { R.open(); }
  void close() { R.close(); }
  
  Record next() {
    while( (r = R.next()) != EOF )
      if( p(r) ) //A==7
        return r;
    return EOF;
  }
  
  PostgreSQL: Init(), GetNext(), ReScan(), MarkPos(), RestorePos(), End()
  ```

- **Blocking Operators**
  - Sorting, grouping/aggregation, build-phase of (simple) hash joins

---

Scalable (small memory)
High CPI measures

Iterator Model – Predicate Evaluation

- **Operator Predicates**
  - Examples: arbitrary selection predicates and join conditions
  - Operators parameterized with in-memory expression trees/DAGs
  - **Expression evaluation engine** (interpretation)

- **Example Selection σ**
  - $(A = 7 \land B \neq 8) \lor D = 9$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-----</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>Product 1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>Product 3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>Product 7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Product 2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

```
 &
 /\ 
 /  
 A  B
==  !=
D   9
```

Plan Execution Strategies
Plan Execution Strategies

Materialized Intermediates (column-at-a-time)

```sql
SELECT count(DISTINCT o_orderkey)
FROM orders, lineitem
WHERE l_orderkey = o_orderkey
AND o_orderdate >= date '1996-07-01'
AND o_orderdate < date '1996-07-01'
   + interval '3' month
AND l_returnflag = 'R';
```

- Column-oriented storage
- Efficient array operations
- DAG processing
- Reuse of intermediates
- Memory requirements
- Unnecessary read/write from and to memory

Generalities

Binary Association Tables (BATs:=OID/Val)

Vectorized Execution (vector-at-a-time)

- Idea: Pipelining of vectors (sub columns) s.t. vectors fit in CPU cache

- Column-oriented storage
- Efficient array operations
- Memory/cache efficiency
- DAG processing
- Reuse of intermediates

Workload: TPCH Q1

[Peter A. Boncz, Marcin Zukowski, Niels Nes: MonetDB/X100: Hyper-Pipelining Query Execution. CIDR 2005]
Plan Execution Strategies

Query Compilation

- **Idea:** Data-centric, not op-centric processing + LLVM code generation

**Operator Trees**
(w/o and w/ pipeline boundaries)

**Compiled Query**
(conceptual, not LLVM)

[Thomas Neumann: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB 2011]
Physical Plan Operators
Overview Plan Operators

- **Multiple Physical Operators**
  - **Different physical operators** for different data and query characteristics
  - Physical operators can have vastly different costs

- **Examples** (supported in most DBMS)

  - **Logical Plan Operators**
    - Selection \( \sigma_p(R) \)
    - Projection \( \pi_A(R) \)
    - Grouping \( \gamma_{G:agg(A)}(R) \)
    - Join \( R \bowtie_{R.a=S.b} S \)

  - **Physical Plan Operators**
    - TableScan
    - IndexScan
    - ALL
    - SortGB
    - HashGB
    - NestedLoopJN
    - SortMergeJN
    - HashJN

Lecture 07

This Lecture
Exercise 3
Nested Loop Join

- **Overview**
  - **Most general join operator** (no order, no indexes, arbitrary predicates \( \theta \))
  - **Poor asymptotic behavior** (very slow)

- **Algorithm** (pseudo code)

```plaintext
for each s in S
  for each r in R
    if( r.RID \&\& s.SID )
      emit concat(r, s)
```

How to implement `next()`?

- **Complexity**
  - Complexity: Time: \( O(N \times M) \), Space: \( O(1) \)
  - Pick smaller table as inner if it fits entirely in memory (buffer pool)
Block Nested Loop / Index Nested Loop Joins

- **Block Nested Loop Join**
  - Avoid I/O by blocked data access
  - Read blocks of \( b_R \) and \( b_S \) \( R \) and \( S \) pages
  - Complexity unchanged but potentially much fewer

- **Index Nested Loop Join**
  - Use index to locate qualifying tuples
    \((=, \geq, >, \leq, <)\)
  - Complexity (for equivalence predicates):
    Time: \( O(N \times \log M) \), Space: \( O(1) \)

\[
\begin{align*}
\text{for each block } b_R \text{ in } R \\
&\quad \text{for each block } b_S \text{ in } S \\
&\quad \quad \text{for each } r \text{ in } b_R \\
&\quad \quad \quad \text{for each } s \text{ in } b_S \\
&\quad \quad \quad \quad \text{if}( r.RID \: \theta \: s.SID ) \\
&\quad \quad \quad \quad \quad \text{emit } \text{concat}(r, s)
\end{align*}
\]

\[
\begin{align*}
\text{for each } r \text{ in } R \\
&\quad \text{for each } s \text{ in } S.IX(\theta, r.RID) \\
&\quad \quad \text{emit } \text{concat}(r, s)
\end{align*}
\]
Sort-Merge Join

- **Overview**
  - **Sort Phase:** sort the input tables R and S (w/ external sort algorithm)
  - **Merge Phase:** step-wise merge with lineage scan

- **Algorithm** (Merge, PK-FK)

```java
Record next() {
    while( curR!=EOF && curS!=EOF ) {
        if( curR.RID < curS.SID )
            curR = R.next();
        else if( curR.RID > curS.SID )
            curS = S.next();
        else if( curR.RID == curS.SID ) {
            t = concat(curR, curS);
            curS = S.next(); //FK side
            return t;
        }
    }
    return EOF;
}
```

- **Complexity**
  - Time (unsorted vs sorted): $O(N \log N + M \log M)$ vs $O(N + M)$
  - Space (unsorted vs sorted): $O(N + M)$ vs $O(1)$
Hash Join

- **Overview**
  - **Build Phase:** read table S and build a hash table $H_S$ over join key
  - **Probe Phase:** read table R and probe $H_S$ with the join key

- **Algorithm** (Build+Probe, PK-FK)

```java
Record next() {
    // build phase (first call)
    while (r = R.next() != EOF )
        Hr.put(r.RID, r);

    // probe phase
    while (s = S.next() != EOF )
        if (Hr.containsKey(s.SID))
            return concat(Hr.get(s.SID), s);

    return EOF;
}
```

- **Complexity**
  - Time: $O(N + M)$, Space: $O(N)$
  - Classic hashing: $p$ in-memory partitions of $H_r$ w/ $p$ scans of R and S
Conclusions and Q&A

- **Summary**
  - Query rewriting and query optimization
  - Query processing and physical operators

- **Exercise 2 Reminder**
  - Submission deadline: **May 07 11.59pm** (+ max 7 late days)
  - Modified submission rules, but crucial to submit

- **Next Lectures**
  - May 13: **09 Transaction Processing and Concurrency**