Outline

- Query Processing Overview
- Algorithms for basic operations
  - Sorting
  - Selection
  - Join
  - Projection
- Query optimization
  - Heuristics
  - Cost-based optimization
Estimate I/O Cost for Implementations

- Count # of disk blocks that must be read (or written) to execute a query plan

\[ b(R) = \# \text{ of blocks containing } R \text{ tuples} \]
\[ r(R) = \# \text{ of } R \text{ tuples} \]
\[ bfr(R) = \text{ max } \# \text{ of tuples of } R \text{ per block} \]
\[ M = \# \text{ memory blocks available} \]
Implementing the JOIN Operation:
- Join (EQUIJOIN, NATURAL JOIN)
  - two-way join: a join on two files
    \[ R \Join_{A=B} S \]
  - multi-way joins: joins involving more than two files.
    \[ R \Join_{A=B} S \Join_{C=D} T \]

Examples
- (OP6): EMPLOYEE \Join_{DNO=DNUMBER} DEPARTMENT
- (OP7): DEPARTMENT \Join_{MGRSSN=SSN} EMPLOYEE
Join

- Factors affecting performance
  - Tuples of relation stored physically together?
  - Relations sorted by join attribute?
  - Indexes exist?

- Algorithms
  - Nested-loop join
  - Sort-merge join
  - Index join
  - Hash join
Nested Loop Join

- **Nested loop join** (conceptually)
  
  for each $r \in R_1$ do
  
  for each $s \in R_2$ do
  
  if $r.C = s.C$ then output $r,s$ pair
Nested Loop Join – Block based implementation (Nested-block join)

- buffer = M blocks (M-1 for reading, 1 for writing)
- Nested-block join
  M-2 blocks for R1 records, 1 block for R2 records, 1 block for writing
  for each M-2 blocks ∈ R1
  for each block ∈ R2
  output matching pairs

- Disk I/O
  - Number of blocks to read?
  - Number of blocks to write?
  - Which file to use as outer loop file?
**Nested Loop Join – Block based implementation (Nested-block join)**

- buffer = M blocks (M-1 for reading, 1 for writing)
- **Nested-block join**
  - M-2 blocks for R1 records, 1 block for R2 records, 1 block for writing
  - for each M-2 blocks ∈ R1
  - for each block ∈ R2
  - output matching pairs

- **Disk I/O**
  - Number of blocks to read: \( b(R1) + b(R1) \times b(R/2) / (M-2) \)
  - Number of blocks to write: \# of blocks of the join results
  - Use smaller file (with fewer blocks) as the outer-loop file
Sort-Merge Join

- Sort-merge join (conceptually)
  1. if R1 and R2 not sorted, sort them
  2. $i \leftarrow 1; j \leftarrow 1$
     while ($i \leq r(R1)) \&\& (j \leq r(R2))$ do
       if $R1\{i\}.C == R2\{j\}.C$ then output matched tuple
       else if $R1\{i\}.C > R2\{j\}.C$ then $j \leftarrow j+1$
       else if $R1\{i\}.C < R2\{j\}.C$ then $i \leftarrow i+1$

- Disk I/O
  - # of blocks to read if R1 and R2 are sorted?
  - # blocks accesses to sort R1 and R2?
Sort-Merge Join

- **Sort-merge join (conceptually)**
  
  1. if R1 and R2 not sorted, sort them
  2. \( i \leftarrow 1; j \leftarrow 1; \)
     
     while \((i \leq r(R1)) \land (j \leq r(R2))\) do
     
     - if \(R1[i].C == R2[j].C\) then output matched tuple
     - else if \(R1[i].C > R2[j].C\) then \(j \leftarrow j+1\)
     - else if \(R1[i].C < R2[j].C\) then \(i \leftarrow i+1\)

- **Disk I/O**
  
  - # of blocks to read if R1 and R2 are sorted: \(b(R1) + b(R2)\)
  - # blocks accesses to sort R1 and R2: \(2b(R1) \log_M b(R1) + 2b(R2) \log_M b(R2)\)
Index Join (single-loop join)

- Index join (Conceptually)

  For each \( r \in R1 \) do
  
  retrieve tuples from R2 using index search (\( R2.C = r.C \))

- Disk I/O

  - # blocks to read?
  - Which file to use as the loop file?
Index Join (single-loop join)

- Index join (Conceptually)
  
  For each \( r \in R1 \) do
  
  retrieve tuples from \( R2 \) using index search (\( R2.C = r.C \))

- Disk I/O
  
  - \# blocks to read: \( b(R1) + r(R1) \) \( \times \) (Index search cost on \( R2 \))
  - Use smaller file as the loop file
Hash Join (general case)

- Hash join (conceptual)
  (1) Partitioning phase:
  - 1 block for reading and M-1 blocks for the hashed partitions
  - Hash R1 tuples into k buckets (partitions)
  - Hash R2 tuples into k buckets (partitions)
  (2) Joining phase (nested block join for each pair of partitions):
  - M-2 blocks for R1 partition, 1 block for R2 partition, 1 block for writing
  For i = 0 to k do
    join tuples in the ith partition of R1 and R2
**Example**  

hash function: even/odd

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>14</td>
</tr>
</tbody>
</table>

Buckets

<table>
<thead>
<tr>
<th>Even</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4 8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Odd:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 5 9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 12 8 14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 3 13 11</td>
</tr>
</tbody>
</table>
Hash Join (general case)

- Hash join (conceptual)
  1. Partitioning phase:
     - 1 block for reading and M-1 blocks for the hashed partitions
     - Hash R1 tuples into k buckets (partitions)
     - Hash R2 tuples into k buckets (partitions)
  2. Joining phase (nested block join for each pair of partitions):
     - M-2 blocks for R1 partition, 1 block for R2 partition, 1 block for writing
     - For i = 0 to k do
       - join tuples in the ith partition of R1 and R2

- Disk I/O
  - Partitioning phase?
  - Joining phase (if each R1 partition can fit into memory)?
  - Total?

- Memory requirement
  - Partitioning phase?
  - Hashing phase (if each R1 partition can fit into memory)?
  - If both required?
Hash Join (general case)

- Hash join (conceptual)
  1. Partitioning phase:
     - 1 block for reading and M-1 blocks for the hashed partitions
     - Hash R1 tuples into k buckets (partitions)
     - Hash R2 tuples into k buckets (partitions)
  2. Joining phase (nested block join for each pair of partitions):
     - M-2 blocks for R1 partition, 1 block for R2 partition, 1 block for writing
     - For i = 0 to k do
       - join tuples in the ith partition of R1 and R2

- Disk I/O
  - Partitioning phase? \(2b(R1) + 2b(R2)\)
  - Joining phase (if each R1 partition can fit into memory)? \(b(R1) + b(R2)\)
  - Total? \(3b(R1) + 3b(R2)\)

- Memory requirement
  - Partitioning phase: \(M-1 \geq k\)
  - Joining phase (if each R1 partition can fit into memory): \(M-2 \geq b(R1)/k\)
  - If both required: \(M \geq \sqrt{b(R1)} + 1\)
Hybrid Hash Join

- Hybrid hash join (conceptual)
  1. Partitioning phase:
     - 1 block for reading and M-1 blocks for the hashed partitions
     - Hash R1 tuples into k partitions, keep 1st partition in memory
     - Hash R2 tuples into k partitions, join with 1st partition of R2
  2. Joining phase (nested block join for each pair of partitions):
     - M-2 blocks for R1 partition, 1 block for R2 partition, 1 block for writing
     - For \( i = 1 \) to \( k \) do (only \( k-1 \) pairs)
       - join tuples in the ith partition of R1 and R2

- Disk I/O
  - Saves some disk I/O for writing and rereading the 1st partition

- Memory requirement
  - Requires more blocks for storing the 1st partition
## Algorithms for Join - Summary

<table>
<thead>
<tr>
<th>Join algorithm</th>
<th>Disk I/O Cost</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested block join</td>
<td>$b(R1) + b(R1)*b(R2)/(M-2)$</td>
<td>ok for “small” relations (relative to memory size); I/O = $b(R1) + b(R2)$ if R1 can fit into buffer</td>
</tr>
<tr>
<td>Sort-merge join w/o sort</td>
<td>$b(R1) + b(R2)$</td>
<td>best if relations are sorted; good for non-equijoin (e.g., $R1.C &gt; R2.C$)</td>
</tr>
<tr>
<td>Sort-merge join w/ sort</td>
<td>$(2\log_M b(R1) +1) * b(R1) + (2\log_M b(R2) +1) * b(R2)$</td>
<td>best for non-equijoin (e.g., $R1.C &gt; R2.C$)</td>
</tr>
<tr>
<td>Hash join</td>
<td>$3b(R1) + 3b(R2)$</td>
<td>best for equijoin if relations are not sorted and no indexes exist</td>
</tr>
<tr>
<td>Index join</td>
<td>$b(R1) + r(R1) * (Index search cost on R2)$</td>
<td>could be useful if index exists but depends on expected result size</td>
</tr>
</tbody>
</table>
Outline

- Overview
- Algorithms for basic operations
  - Sorting
  - Selection
  - Join
  - Projection
- Query optimization
  - Heuristics
  - Cost-based optimization
Algorithms for PROJECT

- Algorithm for PROJECT operations
  \[ \pi_{\text{<attribute list>}}(R) \]
  
  - Extract all tuples from R with only the values for the attributes in \(<\text{attribute list}>\).
  - Remove duplicate tuples

- Methods for removing duplicates
  - Sorting
  - Hashing

- By default, SQL does not remove duplicates
Algorithms for SET Operations

- **CARTESIAN PRODUCT**
  - Nested loop
  - Result set includes all combinations of records
  - Avoid if possible

- **UNION, INTERSECTION, SET DIFFERENCE**
  - Sort-merge:
    - Sort the two relations on the same attributes
    - Merging based on union, intersection, or set difference
  - Hashing:
    - Hash into partitions
    - Merging partitions based on union, intersection, or set difference
Combining Operations using Pipelining

Motivation

- A query is mapped into a sequence of operations.
- Each execution of an operation produces a temporary result.
- Writing and re-reading the temporary files on disk is time consuming.

Pipelining or stream-based processing

- Pipeline the data through multiple operations - pass the result of a previous operator to the next.
Query Optimization

- Logical level - Heuristics based optimization
  - SQL query -> initial logical query tree -> optimized query tree

- Physical level – cost based optimization
  - Optimized query tree -> multiple query plans -> cost estimation -> “best” query plan
Query Tree

- A tree data structure that corresponds to a relational algebra expression.
  - Input relations as leaf nodes
  - Relational algebra operations as internal nodes.
- Execution of the query tree
  - Start at the leaf nodes
  - Execute an internal node whenever its operands are available and replace that node by the result

\[
\begin{align*}
\Pi & \text{title} \\
\sigma & \text{birthdate LIKE ‘%1960’} \\
\Pi & \text{name} \\
\text{starsIn} & \\
\Pi & \text{name} \\
\text{MovieStar} & \\
\end{align*}
\]
Using Heuristics in Query Optimization

- Heuristics optimization
  - The query parser of a high-level query generates a standard initial query tree
  - Apply heuristics rules to find an optimized equivalent query tree

- Main heuristic: apply first the operations that reduce the size of intermediate results.
  - Apply SELECT and PROJECT operations before JOIN or other set operations.
  - Apply more selective SELECT and JOIN first

- General transformation rules for relational algebra operators
Transformation Rules for
Relational Algebra Operations

1. Cascade of $\sigma$: A conjunctive selection condition can be broken up into a cascade (sequence) of individual $\sigma$ operations:
   $\sigma_{c1} \text{ AND } c_2 \text{ AND } \ldots \text{ AND } c_n (R) = \sigma_{c1} (\sigma_{c2} (\ldots (\sigma_{cn}(R))\ldots ))$

2. Commutativity of $\sigma$: The $\sigma$ operation is commutative:
   $\sigma_{c1} (\sigma_{c2}(R)) = \sigma_{c2} (\sigma_{c1}(R))$

3. Cascade of $\pi$: In a cascade (sequence) of $\pi$ operations, all but the last one can be ignored:
   $\pi_{List1} (\pi_{List2} (\ldots (\pi_{Listn}(R))\ldots )) = \pi_{List1}(R)$

4. Commuting $\sigma$ with $\pi$: If the selection condition $c$ involves only the attributes $A_1, \ldots, A_n$ in the projection list, the two operations can be commuted:
   $\pi_{A1, A2, \ldots, An} (\sigma_c (R)) = \sigma_c (\pi_{A1, A2, \ldots, An}(R))$
5. Commutativity of $\Join$ (and $\times$): The $\Join$ operation is commutative as is the $\times$ operation:

- $R \Join C \ S = S \Join C \ R; \ R \times \ S = S \times \ R$

6. Commuting $\sigma$ with $\Join$ (or $\times$): If all the attributes in the selection condition $c$ involve only the attributes of one of the relations being joined—say, $R$—the two operations can be commuted as follows:

- $\sigma_c \ (R \Join S) = (\sigma_c \ (R)) \Join S$
- $\sigma_c \ (R \times S) = (\sigma_{c1} \ (R)) \times (\sigma_{c2} \ (S))$

7. Commuting $\pi$ with $\Join$ (or $\times$): Suppose that the projection list is $L = \{A_1, ..., A_n, B_1, ..., B_m\}$, where $A_1, ..., A_n$ are attributes of $R$ and $B_1, ..., B_m$ are attributes of $S$. If the join condition $c$ involves only attributes in $L$, the two operations can be commuted as follows:

- $\pi_L (R \Join_C S) = (\pi_{A_1, ..., A_n} \ (R)) \Join_C (\pi_{B_1, ..., B_m} \ (S))$
8. Commutativity of set operations: The set operations $\cup$ and $\cap$ are commutative but “–” is not.

9. Associativity of $\times, \cdot, \cup,$ and $\cap$: These four operations are individually associative; that is, if $\theta$ stands for any one of these four operations (throughout the expression), we have
   - $(R \theta S) \theta T = R \theta (S \theta T)$

10. Commuting $\sigma$ with set operations: The $\sigma$ operation commutes with $\cup, \cap,$ and “–”. If $\theta$ stands for any one of these three operations, we have
   - $\sigma_c (R \theta S) = (\sigma_c (R)) \theta (\sigma_c (S))$

- Others on the book
Using Heuristics in Query Optimization

- Using transformation rules to apply Heuristic rules

1. break up any select operations with conjunctive conditions into a cascade of select operations.
2. move each select operation as far down the query tree as is permitted
3. rearrange the leaf nodes so that the leaf node relations with the most restrictive select operations are executed first
4. combine a Cartesian product operation with a subsequent select operation into a join operation.
5. break down and move lists of projection attributes down the tree as far as possible
6. Identify subtrees that represent groups of operations that can be executed by a single algorithm.
Heuristic Optimization of Query Trees

Example:

Q: SELECT LNAME
   FROM EMPLOYEE, WORKS_ON, PROJECT
   WHERE PNAME = ‘AQUARIUS’ AND
   PNMUBER = PNO AND ESSN = SSN
   AND BDATE > ‘1957-12-31’;
Figure 19.5
Steps in converting a query tree during heuristic optimization.
(a) Initial (canonical) query tree for SQL query Q.
(b) Moving SELECT operations down the query tree.
(c) Applying the more restrictive SELECT operation first.
(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.
(e) Moving PROJECT operations down the query tree.

(a)
\[ \pi_{\text{Lname}} \]
\[ \sigma_{\text{Pname}='\text{Aquarius'} \text{ AND Pnumber}=\text{Pno} \text{ AND Essn}=\text{Ssn} \text{ AND Bdate}>'1957-12-31'} \]


(b)
\[ \pi_{\text{Lname}} \]
\[ \sigma_{\text{Pnumber}=\text{Pno}} \]
\[ \sigma_{\text{Essn}=\text{Ssn}} \]
\[ \sigma_{\text{Bdate}>'1957-12-31'} \]

\[ \sigma_{\text{Pname}='\text{Aquarius'}} \]
\[ \text{PROJECT} \]
\[ \text{WORKS_ON} \]
\[ \text{EMPLOYEE} \]
(e)

\[ \pi_{Lname} \times \text{Essn=\text{Ssn}} \]

\[ \pi_{\text{Essn}} \times \text{Pnumber=Pno} \]

\[ \sigma_{\text{Pname='Aquarius'}} \]

\[ \pi_{\text{Pnumber}} \]

\[ \pi_{\text{Essn,Pno}} \]

\[ \sigma_{\text{Bdate>'1957-12-31'}} \]

\[ \text{PROJECT} \]

\[ \text{WORKS_ON} \]

\[ \text{EMPLOYEE} \]
Query Optimization

- **Logical level - Heuristics based optimization**
  - SQL query -> initial logical query tree -> optimized query tree

- **Physical level – cost based optimization**
  - Optimized query tree -> query execution plans -> cost estimation -> “best” query plan
Query Execution Plan

- An execution plan
  - A query tree
  - Access methods to be used for each relation
  - Methods to be used for each operator
- Materialized evaluation: result of an operation is stored as a temporary relation.
- Pipelined evaluation: result of an operation is forwarded to the next operator in sequence.
Cost based Query Optimization

- Estimate and compare the costs of executing a query using different execution strategies and choose the strategy with the lowest cost estimate

Issues

- Cost function
- Number of execution strategies to be considered
Cost-based Query Optimization

Cost Components for Query Execution
1. Disk I/O cost (select, join, etc)
2. Storage cost
3. Computation cost
4. Memory usage cost
5. Communication cost (distributed databases)

Note: Different database systems may focus on different cost components.
Cost-based optimization

- Catalog Information Used in Cost Functions
  - Information about the size of a file
    - number of records (tuples) \( (r) \),
    - record size \( (R) \),
    - number of blocks \( (b) \)
    - blocking factor \( (bfr) \)
  - Information about indexes and indexing attributes of a file
    - Number of levels \( (x) \) of each multilevel index
    - Number of first-level index blocks \( (bi1) \)
    - Number of distinct values \( (d) \) of an attribute
    - Selectivity \( (sl) \) of an attribute
    - Selection cardinality \( (s) \) of an attribute. \( (s = sl \times r) \)
Summary

- Query execution and optimization
- Implementation of query operators
  - Disk I/O cost analysis
- Heuristics based query optimization
- Cost based query optimization