Query Processing

- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions
Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation
Basic Steps in Query Processing

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations

- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.
Basic Steps in Query Processing: Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., \( \sigma_{\text{salary} < 75000} (\Pi_{\text{salary}} (\text{instructor})) \) is equivalent to \( \Pi_{\text{salary}} (\sigma_{\text{salary} < 75000} (\text{instructor})) \)

- Each relational algebra operation can be evaluated using one of several different algorithms
  - a relational-algebra expression can be evaluated in many ways

- Annotated expression specifying detailed evaluation strategy is called an evaluation-plan.
  - e.g., can use an index on \( \text{salary} \) to find instructors with \( \text{salary} < 75000 \),
  - or can perform complete relation scan and discard instructors with \( \text{salary} \geq 75000 \)
Basic Steps: Optimization (Cont.)

- **Query Optimization**: Amongst all equivalent evaluation plans, choose the one with *lowest cost*.
  - Cost is estimated using statistical information from the DB catalog
    - e.g., number of tuples in each relation, size of tuples, etc.

- In this chapter, we study
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression

- In Chapter 13
  - We study how to optimize queries, that is, how to find an evaluation plan with *lowest estimated cost*
Measures of Query Cost

- Cost is generally measured as total elapsed time for answering a query
  - Many factors contribute to time cost
    - disk accesses, CPU, or even network communication
  
- Typically, disk access is the predominant cost and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
  
  - Cost to write a block is greater than cost to read a block
    - data is read back after being written to ensure that the write was successful
Measures of Query Cost (Cont.)

- For simplicity, we just use the **no. of block transfers from disk** & the **no. of seeks** as the cost measures
  - \( t_T \) – time to transfer one block
  - \( t_S \) – time for one seek
  - Cost for \( b \) block transfers plus \( S \) seeks, i.e., \( b \times t_T + S \times t_S \)

- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account

- We do not include cost to writing output to disk in our cost formulae
Selection Operation

- **File scan**

  - Algorithm **A1 (linear search)**. Scan each file block & test all records to see whether they satisfy the selection condition
    - Cost estimate = $b_r$ block transfers + 1 seek
      - $b_r$ denotes number of blocks containing records from relation $r$
    - If selection is on a key attribute, can stop on finding record
      - (average case) cost = $(b_r/2)$ block transfers + 1 seek
    - Linear search can be applied regardless of
      - Selection condition,
      - Ordering of records in the file, or
      - Availability of indices

- **Note**: binary search generally does not make sense, since data is not necessarily stored consecutively
  - Except when there is an index available and in this case index scan can be considered
Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index

- **A2 (primary index, equality on key)**. Retrieve a single record that satisfies the corresponding equality condition
  - Cost = \((h_i + 1) \times (t_T + t_S)\)

- **A3 (primary index, equality on non-key)** Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let \(b\) = number of blocks containing matching records
  - Cost = \(h_i \times (t_T + t_S) + t_S + t_T \times b\)
Selections Using Indices

- **A4 (secondary index, equality on non-key).**
  
  - Retrieve a single record if the search-key is a candidate key
    
    - Cost = \((h_i + 1) \times (t_T + t_S)\)
  
  - Retrieve multiple records if search-key is not a candidate key
    
    - each of \(n\) matching records may be on a different block
    
    - Cost = \((h_i + n) \times (t_T + t_S)\)
      
      - Can be very expensive!
Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:

- **A5 (primary index, comparison).** (Relation is sorted on $A$)
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index

- **A6 (secondary index, comparison)**
  - For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
Implementation of Complex Selections

- **Conjunction:** \( \sigma_{\theta_1} \land \theta_2 \land \ldots \land \theta_n(r) \)

- **A7** *(conjunctive selection using one index)*
  - Select a particular \( \theta_i \) and algorithms A1 through A6 that results in the least cost for \( \sigma_{\theta_i}(r) \)
  - Test other conditions on tuple after fetching it into memory buffer

- **A8** *(conjunctive selection using composite index)*
  - Use appropriate composite (multiple-key) index if available

- **A9** *(conjunctive selection by intersection of identifiers)*
  - Requires indices with record pointers
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers
  - Then fetch records from file
  - If some do not have appropriate indices, apply test in memory
Algorithms for Complex Selections

- **Disjunction:** \( \sigma_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n}(r) \)
  
- **A10 (disjunctive selection by union of identifiers)**
  - Applicable if *all* conditions have available indices
    - Otherwise use linear scan
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers
  - Then fetch records from file

- **Negation:** \( \sigma_{\neg \theta}(r) \)
  - Use linear scan on file
  - If very few records satisfy \( \neg \theta \), and an index is applicable to \( \theta \)
    - Find satisfying records using index and fetch from file
Join Operation

- Several different algorithms to implement joins
  - (Simple) Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge join
  - Hash join

- Choice based on cost estimate

- Assume the following information on Student & Takes:
  - Number of records of student: 5,000  takes: 10,000
  - Number of blocks of student: 100  takes: 400
(Simple) Nested-Loop Join

- To compute the theta join \( r \Join_\theta s \)

  for each tuple \( t_r \) in \( r \) do begin
    for each tuple \( t_s \) in \( s \) do begin
      test pair \( (t_r, t_s) \) to see if they satisfy the join condition \( \theta \)
      if they do, add \( t_r \cdot t_s \) to the result.
    end
  end

- \( r \) is called the **outer relation** and \( s \) the **inner relation** of the join.

- Requires no indices and can be used with any kind of join condition.

- Expensive, since it examines every pair of tuples in the two relations.
In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

\[ n_r \times b_s + b_r \] block transfers, plus \( n_r + b_r \) seeks

If the smaller relation fits entirely in memory, use that as the inner relation.

- Reduces cost to \( b_r + b_s \) block transfers and 2 seeks

Assuming worst-case memory availability cost estimate is

- with student as outer relation:
  \[ 5000 \times 400 + 100 = 2,000,100 \] block transfers, \( 5000 + 100 = 5100 \) seeks
- with takes as the outer relation
  \[ 10000 \times 100 + 400 = 1,000,400 \] block transfers & \( 10,400 \) seeks

If smaller relation (student) fits entirely in memory, the cost estimate will be 500 block transfers.

Block nested-loops algorithm (next slide) is preferable.
Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation

```plaintext
for each block $B_r$ of $r$ do begin
    for each block $B_s$ of $s$ do begin
        for each tuple $t_r$ in $B_r$ do begin
            for each tuple $t_s$ in $B_s$ do begin
                Check if $(t_r, t_s)$ satisfy the join condition
                if they do, add $t_r \cdot t_s$ to the result.
            end
        end
    end
end
```
Block Nested-Loop Join (Cont.)

- Worst case estimate: \( b_r \cdot b_s + b_r \) block transfers + \( 2 \cdot b_r \) seeks
  - Each block in the inner relation \( s \) is read once for each block in the outer relation
- Best case: \( b_r + b_s \) block transfers + 2 seeks.
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use \( M - 2 \) disk blocks as blocking unit for outer relations, where \( M \) = memory size in blocks; use remaining two blocks to buffer inner relation and output
    \[
    \text{Cost} = \left\lceil \frac{b_r}{(M-2)} \right\rceil \cdot b_s + b_r \text{ block transfers} + 2 \left\lceil \frac{b_r}{(M-2)} \right\rceil \text{ seeks}
    \]
  - If equi-join attribute forms a key of inner relation, stop inner loop on first match
  - Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available (next slide)
Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - Join is an equi-join, i.e., natural join, and
  - An index is available on the inner relation’s join attribute
    - Can construct an index just to compute a join
- For each tuple $t_r$ in the outer relation $r$, use the index to look up tuples in $s$ that satisfy the join condition with tuple $t_r$
- Worst case: buffer has space for only one page of $r$, and for each tuple in $r$, we perform an index lookup on $s$.
- Cost of the join: $b_r (t_T + t_S) + n_r \times c$
  - where $c$ is the cost of traversing index and fetching all matching $s$ tuples for one tuple of $r$
  - $c$ can be estimated as cost of a single selection on $s$ using the join condition.
- If indices are available on join attributes of both $r$ and $s$, use the relation with fewer tuples as the outer relation
Example of Indexed Nested-Loop Join Costs

- Compute student ⋈ takes, with student as the outer relation.
- Let takes have a primary B⁺-tree index on the attribute ID, which contains 20 entries in each index node.
- Since takes has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data.
- Student has 5,000 tuples.
- Cost of block nested loops join is
  - 100 * 400 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
    - assuming worst-case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join
  - 100 + 5,000 * 5 = 25,100 block transfers and seeks.
  - Seek cost likely to be less than that for block nested-loop join.
**Merge-Join**

1. Sort both relations on their join attribute (if not already sorted on the join attributes).

2. Merge the sorted relations to join them
   1. Join step is similar to the merge stage of the sort-merge algorithm
   2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
   3. Detailed algorithm in book

```
<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>8</td>
</tr>
<tr>
<td>d</td>
<td>13</td>
</tr>
<tr>
<td>f</td>
<td>7</td>
</tr>
<tr>
<td>m</td>
<td>5</td>
</tr>
<tr>
<td>q</td>
<td>6</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a1</th>
<th>a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>A</td>
</tr>
<tr>
<td>b</td>
<td>G</td>
</tr>
<tr>
<td>c</td>
<td>L</td>
</tr>
<tr>
<td>d</td>
<td>N</td>
</tr>
<tr>
<td>m</td>
<td>B</td>
</tr>
</tbody>
</table>
```

```
Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus the cost of merge join is: $b_r + b_s$ block transfers + $\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks
  - Plus the cost of sorting if relations are unsorted.
Hash-Join

- Applicable for equi-joins and natural joins

- A hash function $h$ is used to partition tuples of both relations

- $h$ maps $JoinAttrs$ values to $\{0, 1, \ldots, n\}$, where $JoinAttrs$ denotes the common attributes of $r$ and $s$ used in the natural join

  - $r_0, r_1, \ldots, r_n$ denote partitions of $r$ tuples
    - Each tuple $t_r \in r$ is put in partition $r_i$, where $i = h(t_r[JoinAttrs])$

  - $r_0, r_1, \ldots, r_n$ denotes partitions of $s$ tuples
    - Each tuple $t_s \in s$ is put in partition $s_i$, where $i = h(t_s[JoinAttrs])$

- Note: In book, $r_i$ is denoted as $H_{r_i}$, $s_i$ is denoted as $H_{s_i}$ and $n$ is denoted as $n_h$. 
Hash-Join (Cont.)

\[ r \quad \rightarrow \quad \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\ldots & \ldots & \ldots & \ldots \\
4 & & & \\
\end{array} \quad \rightarrow \quad \begin{array}{cccc}
0 & 1 & 2 & 3 \\
\ldots & \ldots & \ldots & \ldots \\
4 & & & \\
\end{array} \quad \rightarrow \quad s \\
\]

partitions of \( r \)  
partitions of \( s \)
Hash-Join (Cont.)

- $r$ tuples in $r_i$ need only to be compared with $s$ tuples in $s_i$. Need not be compared with $s$ tuples in any other partition, since
  - An $r$ tuple and an $s$ tuple that satisfy the join condition will have the same hash value for the join attributes.
  - If that value is hashed to some value $i$, then $r$ tuple has to be in $r_i$ and the $s$ tuple in $s_i$. 
Sorting

- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.

- For relations that fit in memory, techniques like quicksort can be used. For relations that don’t fit in memory, **external sort-merge** is a good choice.
External Sort-Merge

Let $M$ denote memory size (in pages/blocks)

1. **Create sorted runs.** Let $i$ be 0 initially.

   Repeatedly do the following till the end of the relation:
   
   (a) Read $M$ blocks of relation into memory
   
   (b) Sort the in-memory blocks
   
   (c) Write sorted data to run file $R_i$; increment $i$

   Let the final value of $i$ be $N$
External Sort-Merge (Cont.)

2. **Merge the runs (N-way merge).** We assume that $N < M$.

   1. Use $N$ blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page.

2. **repeat**

   i. Select the first record (in sort order) among all buffer pages.

   ii. Write the record to the output buffer. If the output buffer is full, write it to disk.

   iii. Delete the record from its input buffer page.

   **If** the buffer page becomes empty **then**

       read the next block (if any) of the run into the buffer.

3. **until** all input buffer pages are empty.
If $N \geq M$, several merge passes are required

- In each pass, contiguous groups of $M - 1$ runs are merged
- A pass reduces the number of runs by a factor of $M - 1$, and creates runs longer by the same factor
  - e.g., if $M = 11$, and there are 90 runs, one pass reduces the number of runs by 9, each 10 times the size of the initial runs
- Repeated passes are performed till all runs have been merged into one
Assumption:
(i) one tuple fits into a block
(ii) Memory holds 3 blocks
Hash-Join Algorithm

The hash-join of $r$ and $s$ is computed as follows:

1. Partition the relation $s$ using hashing function $h$. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.

2. Partition $r$ similarly.

3. For each $i$
   
   (a) Load $s_i$ into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one $h$.

   (b) Read the tuples in $r_i$ from the disk one by one. For each tuple $t_r$ locate each matching tuple $t_s$ in $s_i$ using the in-memory hash index. Output the concatenation of their attributes.

Relation $s$ is called the build input and $r$ is called the probe input.
The value $n$, i.e., number of partitions, and the hash function $h$ is chosen such that each $s_i$ should fit in memory.

- Typically, $n$ is chosen as $\left\lceil \frac{b_s}{M} \right\rceil * f$, where $f$ is a “fudge factor”, typically around 1.2.

- The probe relation partitions $r_i$ need not fit in memory.

**Recursive partitioning** required if number of partitions $n$ is greater than number of pages $M$ of memory.

- Instead of partitioning $n$ ways, use $M - 1$ partitions for $s$.

- Further partition the $M - 1$ partitions using a different hash function.

- Use same partitioning method on $r$.
Partitioning is said to be **skewed** if some partitions have significantly more tuples than some others.

**Hash-table overflow** occurs in partition $s_i$ if $s_i$ does not fit in memory. Reasons could be:
- Many tuples in $s$ with same value for join attributes
- Bad hash function

**Overflow resolution** can be done in build phase:
- Partition $s_i$ is further partitioned using different hash function.
- Partition $r_i$ must be similarly partitioned.

**Overflow avoidance** performs partitioning carefully to avoid overflows during build phase, e.g., partition build relation into many partitions, then combine them.

Both approaches fail with large numbers of duplicates:
- Fallback option: use block nested loops join on overflowed partitions
Complex Joins

- Join with a conjunctive condition:
  \[ r \bowtie_{\theta_1 \land \theta_2 \land \ldots \land \theta_n} s \]
  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins \( r \bowtie_{\theta_i} s \)
    - Final result comprises those tuples in the intermediate result that satisfy the remaining conditions
    \[ \theta_1 \land \ldots \land \theta_{i-1} \land \theta_{i+1} \land \ldots \land \theta_n \]

- Join with a disjunctive condition \( r \bowtie_{\theta_1 \lor \theta_2 \lor \ldots \lor \theta_n} s \)
  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in \( r \bowtie_{\theta_i} s \):
    \[ (r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \ldots \cup (r \bowtie_{\theta_n} s) \]