Chapter 12: Query Processing
Chapter 12: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Join Operation
- Evaluation of Expressions
Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

Diagram:
- Query
- Parser and translator
- Relational-algebra expression
- Optimizer
- Execution plan
- Evaluation engine
- Data
- Statistics about data
Basic Steps in Query Processing (Cont.)

- 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., $\sigma_{salary<75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary<75000}(instructor))$

- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, 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 salary to find instructors with salary < 75000,
  - or can perform complete relation scan and discard instructors with 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 database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.

- In this set of slides we study
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression

- In the following set of slides
  - 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 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
For simplicity we just use the number of block transfers from disk and the number 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
  \[ b \times t_T + S \times t_S \]

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.
Several algorithms can reduce disk I/O by using extra buffer space

- Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
Selection Operation

- **File scan**

- Algorithm A1 (**linear search**). Scan each file block and 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
    - cost = \( (b_r/2) \) block transfers + 1 seek
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

- Note: binary search generally does not make sense since data is not stored consecutively
  - except when there is an index available,
  - and binary search requires more seeks than index search
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 = \((h_i + 1) \times (t_T + t_S)\)
    - \(h_i\) = number of blocks needed to retrieve to consult an index entry

- **A3 (primary index, equality on nonkey)** 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 nonkey).
  - Retrieve a single record if the search-key is a candidate key
    - Cost = \((h_i + 1) * (t_T + t_S)\)
  - Retrieve multiple records if search-key is not a candidate key
    - each of \(n\) matching records may be on a different block
    - Cost = \((h_i + n) * (t_T + t_S)\)
      - 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
    - Linear file scan may be cheaper
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 combination of $\theta_i$ and algorithms A1 through A7 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 conditions do not have appropriate indices, apply test in memory.
Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta_1 \lor \theta_2 \lor \cdots \lor \theta_n}(r)$.

- **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
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of student: 5,000 takes: 10,000
  - Number of blocks of student: 100 takes: 400
Nested-Loop Join

- To compute the theta join \( r \bowtie_{\theta} s \) 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 and 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: $b_r \times b_s + b_r$ block transfers + 2 * $b_r$ 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
Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or 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 or $r$
  - $c$ 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.
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
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 \[ + \lceil \frac{b_r}{M} \rceil + \lceil \frac{b_s}{M} \rceil \] seeks
  where \( 2M \) is the available memory
  - + 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 $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1, \ldots, s_n$ denotes partitions of $s$ tuples
    - Each tuple $t_s \in s$ is put in partition $s_i$, where $i = h(t_s[JoinAttrs])$. 
Hash-Join (Cont.)

Hash

Hash

- Join (Cont.)

partitions of $r$

partitions of $s$

$0$

$1$

$2$

$3$

$4$

$r$

$s$
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 value for the join attributes.
  - If that value is hashed to some value $i$, the $r$ tuple has to be in $r_i$ and the $s$ tuple in $s_i$. 
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.
Hash-Join algorithm (Cont.)

- The value $n$ 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
Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - Materialization: generate results of an expression whose inputs are relations or are already computed, materialize (store) it on disk. Repeat.
  - Pipelining: pass on tuples to parent operations even as an operation is being executed