15. Query Processing

15.1 Overview

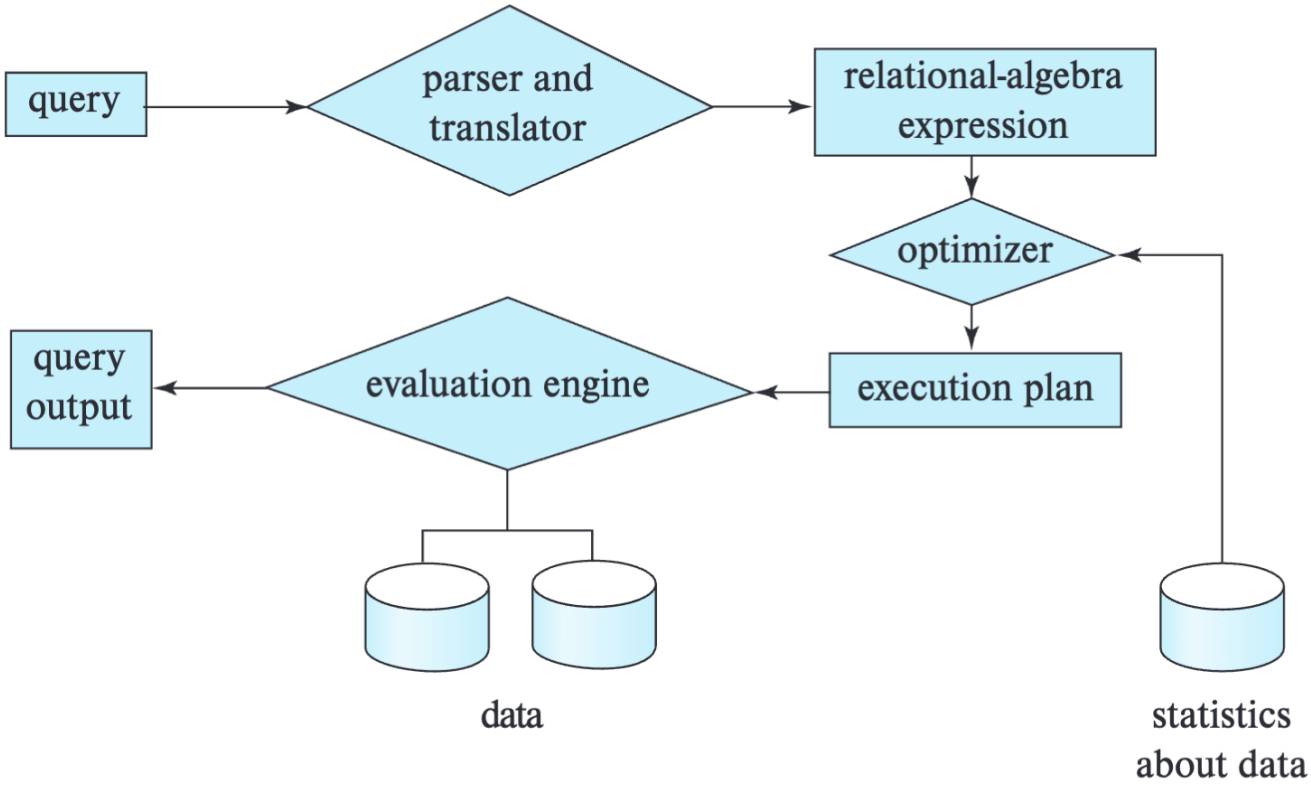

Figure 15.1 Steps in query processing.

> Figure 15.1 Steps in query processing.
> Abraham Silberschatz, Henry F. Korth, S. Sudarshan - Database System Concepts-McGraw-Hill Higher Education (2019), page 719

The steps involved in processing a query:

1. Parsing and translation
    1. query -> parse-tree -> extended relational-algebra expression.
    2. the relational-algebra representation specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions
    3. instead of using the relational-algebra representation, several databases, such as Postgres, use an annotated parse-tree representation based on the structure of the given SQL query.
2. Optimization

    1. query optimization: the task to construct a query-evaluation plan that minimizes the cost of query evaluation.
    2. **Evaluation Primitive**: A relational-algebra operation annotated with instructions on how to evaluate it.
        1. Annotations may state the algorithm to be used for a specific operation or the particular index or indices to use
    3. **query-execution plan (query-evaluation plan)**: A sequence of primitive operations that can be used to evaluate a query.
3. Evaluation
    1. The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

15.2 Measures of Query Cost

In general, we evaluate Query Cost with I/O cost.

- cost of query evaluation can be measured in terms of several different resources, including:
    - disk accesses
    - CPU time to execute a query
    - the cost of communication (in parallel and distributed database systems)

- For large databases resident on magnetic disk, the I/O cost to access data from disk usually dominates the other costs
    - *number of blocks transferred* from storage
        - t_T: average seconds to transfer a block (magnetic disks: 4 ms; SATA SSD: 10μs for a 4-kb block)
    - the *number of random I/O accesses*
        - t_S: average block-access time (disk seek time plus rotational latency) (magnetic disks: 0.1 ms; SATA SSD: 90 μs; main memory: <100 ns)
    - an operation that transfers b blocks and performs S random I/O accesses would take $b * t_T + S * t_S$ seconds.
- For data that are already present in main memory, reads happen at the unit of **cache lines**, instead of disk blocks.
- With data resident in memory or on SSDs, I/O cost does not dominate the overall cost, and we must include CPU costs when computing the cost of query evaluation. We do not include CPU costs in our model to simplify our presentation.
- The costs of all the algorithms that we consider depend on the **size of the buffer** in the main memory.

- **Response Time** for a query-evaluation plan is very hard to estimate without actually executing the plan
  - response time depends on the contents of the buffer when the query begins execution; this information is not available
  - In a system with multiple disks, the response time depends on how accesses are distributed among disks, which is hard to estimate without detailed knowledge of data layout on disk.
- Instead of trying to minimize the response time, optimizers generally try to **minimize the total resource consumption** (disk access time, including seek and data transfer) of a query plan.
  - resource consumption-based model of query cost

15.3 Selection Operation

- the **file scan** is the lowest-level operator to access data
  - File scans are search algorithms that locate and retrieve records that fulfill a selection condition

Selections Using File Scans and Indices

- Linear Search
- Index Scans: Search algorithms that use an index.
  - **clustering index (primary index)** is an index that allows the records of a file to be read in an order that corresponds to the physical order in the file.
  - An index that is not a clustering index is called a **secondary index (nonclustering index)**.
- > Figure 15.3 Cost estimates for selection algorithms. Abraham Silberschatz, Henry F. Korth, S. Sudarshan - Database System Concepts-McGraw-Hill Higher Education (2019), page 725
- Selections Involving Comparisons
  - clustering index, comparison: For comparisons of the form A <v or A ≤v, an index lookup is not required.
  - secondary index, comparison: If the number of retrieved records is large, using the secondary index may be even more expensive than using linear search. Therefore, the secondary index should be used only if **very few records are selected**.

- If the number of matching tuples is known ahead of time, query optimizer can choose between using a secondary index or using a linear scan based on the cost estimates.
    - PostgreSQL uses a hybrid algorithm called **bitmap index scan**.
        - Bitmap Index Scan algorithm uses a hybrid approach when a **secondary index is available**, but the number of **matching records is unknown**
        - When dealing with multi-condition queries, e.g., c1 and c2, the Bitmap Index Scan leverages the secondary index to create a “map” to only scan the blocks that likely contain the records we want (satisfying c1), saving time and resources.

15.4 Sorting

Importance of sorting:

1. SQL queries can specify that the output be sorted.
2. several of the relational operations, such as joins, can be implemented efficiently if the input relations are first sorted.
   Two cases:
3. relations that fit entirely in main memory
    - standard sorting techniques such as quick-sort
4. for relations that are bigger than memory: external sorting
    - External Sort-Merge Algorithm

15.5 Join Operation

15.5.1 Nested-Loop Join

Nested-Loop Join requires no index, and it **can be used regardless of what the join condition is**.

- theta join: $r \bowtie_\theta s$
    - outer relation: r, inner relation: s
    - # pairs of tuples to be considered is $n_r * n_s$
    - natural join: add an extra step of deleting repeated attributes from the tuple $t_r \cdot t_s$, before adding it to the result.
- nested-loop join

```
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 θ
      if they do, add t_r · t_s to the result;
   end
end
```

- Worst Case
  - the buffer can hold only one block of each relation
  - total block transfers = $n_r * b_s + b_r$
  - total seeks = $n_r + b_r$

- Best Case
  - there is enough space for both relations to fit simultaneously in memory
  - total block transfers = $b_r + b_s$
  - total seeks = 2
    - first to find and transfer t_s into main memory
    - second to find the starting position of t_r, and read blocks of the r table sequentially
- If one of the relations **fits entirely in main memory**, it is beneficial to use that relation as the **inner relation**, since the inner relation would then be read only once.
  - if s is small enough to fit in main memory, our strategy requires only a total $b_r + b_s$ block transfers and two seeks

15.5.2 Block Nested-Loop Join

- an improved version of the nested-loop join algorithm, processing the relations by blocks instead of by tuples, which can significantly reduce the number of block accesses.

```
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
            test pair (t_r, t_s) to see if they satisfy the join condition
            if they do, add t_r · t_s to the result;
         end
      end
   end
end
```

- worst case

    - It's more efficient to use the smaller relation as the outer relation.
    - block transfer: $b_r * b_s + b_r$
    - seeks: $2 * b_r$
- best case
    - inner relation fits in memory
    - block transfer: $b_r + b_s$
    - seeks: 2
- use the biggest blocking unit for the outer relation, that can fit in memory, while leaving enough space for the buffers of the inner relation and the output.

15.5.3 Indexed Nested-Loop Join

If an index is available on the inner loop's join attribute, index lookups can replace file scans.

- worst case
    - one block of r, one block of the index
    - read relation r: b_r I/O operations, each I/O requires a seek and a block transfer
    - cost: $b_r(t_T + t_S) + n_r * c$
        - r is the number of records in relation r
        - c is the cost of a single selection on s using the join condition.
- if indices are available on both relations r and s, it is generally most efficient to use the one with fewer tuples as the outer relation.
- Although the number of block transfers has been reduced, the seek cost has actually increased
- if we had a selection on the outer relation that reduces the number of rows significantly, indexed nested-loop join could be significantly faster than block nested-loop join

15.5.4 Merge Join

- used to compute **natural joins** and **equi-joins**
- Merge Join is an efficient algorithm that joins two sorted relations by sequentially scanning and merging tuples with matching join attributes, significantly reducing the number of comparisons and improving performance, especially when dealing with large datasets.

$pr :=$ address of first tuple of r;
$ps :=$ address of first tuple of s;
while ($ps \neq$ null and $pr \neq$ null) do
begin
  $t_s :=$ tuple to which ps points;
  $S_s := \{t_s\}$;
  set ps to point to next tuple of s;
  done := false;
  while (not done and $ps \neq$ null) do
  begin
    $t'_s :=$ tuple to which ps points;
    if ($t'_s[JoinAttrs] = t_s[JoinAttrs]$)
    then begin
      $S_s := S_s \cup \{t'_s\}$;
      set ps to point to next tuple of s;
    end
    else done := true;
  end
end
$t_r :=$ tuple to which pr points;
while ($pr \neq$ null and $t_r[JoinAttrs] < t_s[JoinAttrs]$) do
begin
  set pr to point to next tuple of r;
  $t_r :=$ tuple to which pr points;
end
while ($pr \neq$ null and $t_r[JoinAttrs] = t_s[JoinAttrs]$) do
begin
  for each t_s in S_s do
  begin
    add $t_s \bowtie t_r$ to result;
  end
  set pr to point to next tuple of r;
  $t_r :=$ tuple to which pr points;
end
end.

- If there are some join attribute values for which Ss is larger than available memory, a block nested-loop join can be performed for such sets Ss, matching them with corresponding blocks of tuples in r with the same values for the join attributes.
- number of block transfers: $b_r + b_s$
- Assuming that b_b buffer blocks are allocated to each relation, the number of disk seeks required would be $\lceil b_r \ / \ b_b \rceil + \lceil b_s \ / \ b_b \rceil$disk seeks.

15.5.4.3 Hybrid Merge Join

1. One relation is sorted.
2. The other relation is unsorted but has a secondary B+-tree index on the join attributes.
   Hybrid Merge-Join Steps
3. **Merge Sorted Relation with Index**:
   - The algorithm merges the sorted relation with the leaf entries of the B+-tree index from the unsorted relation.
   - The result file contains tuples from the sorted relation and addresses for tuples in the unsorted relation.
4. **Sort Result File**:
   - The result file is sorted based on the addresses of tuples in the unsorted relation.
5. **Efficient Retrieval**:
   - This sorting allows efficient retrieval of the corresponding tuples from the unsorted relation in physical storage order to complete the join.

15.5.5 Hash Join

- for natural joins and equi-joins
- partition the tuples of each of the relations into sets that have the same hash value on the join attributes.
- The hash join algorithm is used to join two relations (tables) r and s. If a tuple from r and a tuple from s satisfy the join condition, they have the same value for the join attributes. These values are hashed to a partition i, so tuples in r only need to be compared with tuples in s in the same partition i.

Concept

The hash join algorithm is used to join two relations (tables) `r` and `s`. If a tuple from `r` and a tuple from `s` satisfy the join condition, they have the same value for the join attributes. These values are hashed to a partition `i`, so tuples in `r` only need to be compared with tuples in `s` in the same partition `i`.

Algorithm Steps

1. **Partitioning**:
   - Partition relation `s`: Based on the hash value of the join attributes, distribute the tuples of `s` into different partitions `si`.

- Partition relation `r`: Similarly, distribute the tuples of `r` into different partitions `ri` based on the hash value of the join attributes.

2. **Join on each partition**:
   - For each partition pair `i` (`si` and `ri`), first build an in-memory hash index on `si` (s is called **build input**).
   - Use the tuples in `ri` to probe the hash index on `si` to find matching tuples and perform the join (r is called **probe input**).

Features

- **Reduce comparisons**: Only tuples in the same partition are compared, reducing unnecessary comparisons.
- **Single pass**: Each relation is scanned only once during the build and probe phases.

Optimization and Choice

- **Choosing the right number of partitions**: Select an appropriate number of partitions `nh` to ensure that each partition `si` and its hash index can fit into memory.
- **Prefer smaller relation as the build relation**: Using the smaller relation as the build input can reduce memory usage and improve efficiency.
  Pseudocode Explanation**

1. ****Partition relation `s`**:
   - For each tuple `ts` in `s`, compute its hash value `i` and place it into partition `Hsi`.

```
for each tuple ts in s do begin
    i := h(ts[JoinAttrs]);
    Hsi := Hsi ∪ {ts};
end
```

2. **Partition relation `r`**:
   - For each tuple `tr` in `r`, compute its hash value `i` and place it into partition `Hri`.

```
for each tuple tr in r do begin
    i := h(tr[JoinAttrs]);
    Hri := Hri ∪ {tr};
end
```

3. **Join on each partition**:

- For each partition pair `i`, build an in-memory hash index on `Hsi`.
- Use the tuples in `Hri` to probe the hash index on `Hsi` and join the matching tuples.

```
for i := 0 to nh do begin
    read Hsi and build an in-memory hash index on it;
    for each tuple tr in Hri do begin
        probe the hash index on Hsi to locate all tuples ts
        such that ts[JoinAttrs] = tr[JoinAttrs];
        for each matching tuple ts in Hsi do begin
            add tr ⋈ ts to the result;
        end
    end
end
```

Cost of Hash Join

- block transfers: $3(b_r + b_s) + 4n_h$, h denotes number of partition pairs
- seeks: $2(\lceil b_r \,/\, b_b \rceil + \lceil b_s \,/\, b_b \rceil) + 2n_h$

15.5.6 Complex Joins

- Conjunctive condition
  - $r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$
  - first compute the result of individual condition θ_1 to get intermediate result
  - sift the intermediate result with the remaining conditions
- Disjunctive condition
  - $r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s$
  - $(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \cdots \cup (r \bowtie_{\theta_n} s)$

15.6 Other Operations

15.6.1 Duplicate Elimination

1. Sorting for Duplicate Elimination
2. Hashing for Duplicate Elimination

15.6.3 Set Operations

1. **Set Operations Using Sorting**:
    - **Union (r ∪ s)**: Sort both relations, scan them, and retain only one copy of duplicate tuples.
    - **Intersection (r ∩ s)**: Sort both relations, scan them, and retain only tuples present in both relations.
    - **Set Difference (r − s)**: Sort both relations, scan them, and retain tuples in `r` that are not in `s`.
    - **Cost**: If relations are already sorted, cost is `br + bs` block transfers and disk seeks. If not, include sorting cost.
2. **Set Operations Using Hashing**:
    - **Partitioning**: Use the same hash function to partition both relations into `r0, r1, ..., rnh` and `s0, s1, ..., snh`.
    - **Union (r ∪ s)**:
        1. Build an in-memory hash index on `ri`.
        2. Add tuples from `si` to the hash index if not already present.
        3. Add tuples from the hash index to the result.
    - **Intersection (r ∩ s)**:
        1. Build an in-memory hash index on `ri`.
        2. For each tuple in `si`, probe the hash index and add it to the result if already present.
    - **Set Difference (r − s)**:
        1. Build an in-memory hash index on `ri`.
        2. For each tuple in `si`, probe the hash index and delete it if present.
        3. Add remaining tuples in the hash index to the result.

15.6.4 Outer Join

Strategies to Implement Outer Joins:

1. Strategy 1: Compute Join First
    1. Compute the join $r \bowtie_\theta s$ and save it as temporary relation q1.
    2. Compute $r - \Pi_R(q_1)$ to find tuples in r not in the join.
    3. Pad these tuples with nulls for s attributes and add them to q1 to get the left outer join.
    4. Right outer join r ⟕θ s can be implemented similarly by reversing r and s.
    5. Full outer join r ⟗θ s combines both left and right outer joins.
2. Strategy 2: Modify Join Algorithms
    1. **Nested-loop join**: left outer join, full outer join(possible but complex)

2. **Merge-join**: full outer join

15.7 Evaluation of Expressions

- How to evaluate a multiple-operation expression
  - evaluate on operation at a time. The result of each evaluation is **materialized** in a temporary relation.
    - cons: temporary relation must be written to disk
  - Evaluate several operations simultaneously in a **pipeline**, without the need to store a temporary relation

15.7.1 Materialization

Materialized evaluation
Each intermediate result is created (materialized) and used for the next operation.
Cost Consideration:
- The cost includes not only the operations but also writing intermediate results to disk.
- Estimate the number of blocks written as (\frac{nr}{fr}), where (nr) is the number of tuples and (fr) is the blocking factor.
- Disk seeks can be estimated as (\lceil \frac{br}{bb} \rceil), where (bb) is the size of the output buffer.
- Double buffering can reduce seeks by allowing parallel execution of CPU and I/O activities.

15.7.2 Pipelining

1. **Pipelined Evaluation**:
   - improve query-evaluation efficiency by reducing the number of temporary files that are produced.
   - Combine multiple relational operations into a pipeline where the output of one operation is immediately passed to the next without creating temporary files
   - Example: In the expression (\Pi_{a1, a2}(r \bowtie s)), instead of materializing the join result in a temporary file and then performing the projection, pass each tuple from the join directly to the projection operation.
2. **Benefits of Pipelined Evaluation**:
   - **Reduced Cost**:
     - Eliminates the need to read and write temporary relations, lowering query evaluation costs.

    - Modify cost formulas by removing the cost of reading inputs from disk if pipelining is used.
- **Quick Result Generation**:
    - Results can be generated and displayed to the user as soon as they are produced, reducing delay.

15.7.2.1 Implementation of Pipelining

```
Error parsing Mermaid diagram!

Parse error on line 11:
...tion]  ```1. **Pipeline Implement
----------------------^
Expecting 'SEMI', 'NEWLINE', 'EOF', 'AMP', 'START_LINK', 'LINK', got 'MULT'
```

Insights & resolution:

- The subexpression $(instructor \bowtie teaches \bowtie \Pi_{course\_id,title}(course))$ can create a very large intermediate result.
    - By reducing the number of tuples of the instructor relation that we need to access, we reduce the size of the intermediate result
    - $\Pi_{name,title}((\sigma_{dept\_name="Music"}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,title}(course))))$
- Use different join algorithms (hash join, merge join)
- Use Indices

Generation of query-evaluation plans involves:

1. generating expressions that are **logically equivalent** to the given expression
    - *Equivalence rules*
2. annotating the resultant expressions in alternative ways to generate alternative query-evaluation plans
3. estimating the cost of each evaluation plan, and choosing the one whose estimated cost is the least. (searching algorithm)

    - > VIEWING QUERY EVALUATION PLANS Abraham Silberschatz, Henry F. Korth, S. Sudarshan - Database System Concepts-McGraw-Hill Higher Education (2019), page 775

16.2 Transformation of Relational Expressions

Abraham Silberschatz, Henry F. Korth, S. Sudarshan - Database System Concepts-McGraw-Hill Higher Education (2019), page 776

- **Equivalent** relational-algebra expressions: on every legal database instance, the two expressions generate the same set of tuples.

16.2.1 Equivalence Rules

1. a cascade of σ

$$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. selections are commutative

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. a cascade of Π

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) \equiv \Pi_{L_1}(E)$$

\text{where } L_1 \subseteq L_2 \subseteq \ldots \subseteq L_n.

4. selections can be combined with Cartesian products and theta joins 1. $\sigma_{\theta}(E_1 \times E_2) \equ