Basic Steps in Query Processing

1. Parsing and translation: Translation of SQL queries into naive RA
2. Optimization: of RA expressions by
   1. Pushing selection and projection
   2. Estimating cost of different join orders, and selecting that of minimum cost
   3. Selecting best implementation for each operator.
3. Evaluation

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

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.
  ```sql
  for each block R, of R do begin
    for each block S, of S do begin
      for each tuple t in R, do begin
        for each tuple s in S, do begin
          Check if (t, s) satisfy the join condition
          if they do, add t * s to the result.
        end
      end
    end
  end
  ```

R is the outer relation and S is the inner relation
Index Join (conceptual)
1. If index supporting \( S.C \) does not exist, create one, and
2. For each \( r \in R \) do
   \[ X ? \text{index-lookup}(S.C, r.C) \]
   For each \( s \in X \), output \((r, s)\)

Selection and Indexing
- If we have a \( S.C \) condition supported by an existing index we use the index
- If we have a conjunction, such as \( S.A > 5 \) and \( S.B < 10 \) with indexes on both, then we can select the better of the two (optimization)
- If there is no index, do we create one?

Cost of Joins
- Nested Loop: \( b_R \times |R| \times b_S \) (\( b_R \) and \( b_S \) denote the number of blocks in \( R \) and \( S \), resp.) this is the worst case. What is the best case?
- Pre-existing Index on \( S \): \( b_R \times |R| \times k \) (\( k \): depth of index)
- Block-Nested-Loop Join: \( b_R \times b_S \times b_S \) (\( b_S \): number of blocks in \( S \))

Sort-Merge Join
- Sort the relations first and join

Merge-Join
1. Can be used only for equi-joins and natural joins
2. Sort both relations on their join attribute (if not already sorted on the join attributes)
3. Merge the sorted relations to join them

Sort-Merge Join (conceptual)
1. If \( R \) and \( S \) not sorted, sort them
2. \( i = 1; j = 1; \)
   While \((i \leq |R| \land j \leq |S|)\) do
      if \( R[i].C = S[j].C \) then outputTuples
      else if \( R[i].C > S[j].C \) then \( j \leftarrow j + 1 \)
      else if \( R[i].C < S[j].C \) then \( i \leftarrow i + 1 \)
**External Sorting Using Sort-Merge**

- a 24
- d 31
- c 13
- b 14
- e 16
- r 16
- m 3
- p 2
- a 14
- c 24
- b 14
- e 16
- d 31
- e 16
- a 14
- p 2
- r 16
- m 3
- c 13
- b 14
- e 16
- d 31
- e 16
- a 14
- p 2
- r 16
- m 3

---

**Hash Join**

- **Hash tuples into buckets**
- **Join between only matching buckets**

\[ H(k) = k \mod 3 \]

```
<table>
<thead>
<tr>
<th>R</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

---

**Hash Join (conceptual)**

- Hash function \( h \), range \( 1 \leq k \)

**Algorithm**

1. Hashing stage (bucketizing)
   - Hash \( R \) tuples into \( G_1, \ldots, G_k \) buckets
   - Hash \( S \) tuples into \( H_1, \ldots, H_k \) buckets
2. Join stage
   - For \( i = 1 \) to \( k \) do
     - match tuples in \( G_i, H_i \) buckets

```
<table>
<thead>
<tr>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>40</td>
<td>15</td>
</tr>
</tbody>
</table>
```

---

**Materialization vs Pipelining**

- **Materialized evaluation**: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- **Pipelined evaluation**: one tuple at a time.

---

**Summary of Join Algorithms**

- **Nested-loop join** ok for "small" relations (relative to memory size)
- **Hash join** usually best for equi-join if relations not sorted and no index
- **Merge join** for sorted relations
- **Sort merge join** good for non-equi-join
- Consider index join if index exists
- DBMS maintains statistics on data
Statistics collection commands

- **DBMS** has to collect statistics on tables/indexes
  - For optimal performance
  - Without stats, DBMS does stupid things...
- **DB2**
  - `RUNSTATS ON TABLE <userid>.<table> AND INDEXES ALL`
- **Oracle**
  - `ANALYZE TABLE <table> COMPUTE STATISTICS`
  - `ANALYZE TABLE <table> ESTIMATE STATISTICS` (cheaper than `COMPUTE`)
- Run the command after major update/index construction