Query Processing: From Logical to Physical Plans

SQL query
⇒ logical plan (i.e., algebra expression)
⇒ preferred logical plan (using rewritings)
⇒ physical plan (???)

• choose order for associative and commutative operators
• choose algorithm for each operator
• add operators needed for the physical plan (scan, sort, ...)
• choose on operator interaction (materialize intermediate results on disk vs iterators (“getnext”) which allow a pipelined in memory evaluation)

⇒ based on estimated evaluation cost

Estimating Costs

• Estimate
  – the size of results
  – the number of disk I/Os
• For each relation R consider
  – \( T(R) \) : # tuples in R
  – \( S(R) \) : # bytes in each R tuple
  – \( B(R) \) : # blocks to hold all R tuples
  – \( V(R, A) \) : # distinct tuples over the attribute(s) \( A (=A_1, ..., A_n) \) of R
Physical Query Plan Operators: SCAN

- **scan(R)**: bring (all/part of) R from disk to memory, based on
  - disk blocks: **table-scan** (e.g., union, join R)
  - index: **index-scan** (e.g., selections on R)
  - ordered by attribute A: **sort-scan** (e.g., use B-tree index on A, or table/index-scan + in-memory sort, or external merge-sort)

- number of disk I/Os:
  - table-scan: $B(R)$ (assuming R is clustered)
  - index-scan: $|index| < |R|$
  - sort-scan:
    - $B(R)$ if R fits into memory then
    - $\sim 3 * B(R)$ if R is clustered but needs 2-phase multiway merge sort
      (read R into sublists, write out sublists, reread sublist -- ignores final write; pipelined execution)

Digression: 2-Phase Multiway Mergesort ...

- **merge-sort L**:
  - partition list into two halves L1, L2
  - merge-sort(L1); merge-sort(L2)
  - merge(L1,L2)

- 2-phase multiway merge-sort: variant suitable for dbs:
  - **phase 1**: sort largest possible chunks of R in-memory $\Rightarrow n$ sorted sublists are written to disk
  - **phase 2**: merge the $n$ sorted sublists (multiway)
    - read one block of each sublist into memory (assuming they all fit)
    - use an output buffer (one block)
  - note: this means two passes over the data (whereas table-scan, index-scan need just one pass)

...2-Phase Multiway Mergesort

- How many records can we sort?
  - $BS=$ block size, $MS=$ memory size for block buffers, $RS=$ record size
  - $\Rightarrow MS/BS$ memory buffers $\Rightarrow n = MS/BS - 1$ sublists
  - in phase 1: $MS/RS$ records are sorted for each sublist
  - $\Rightarrow (MS/BS - 1) * MS/RS = MS/(RS * BS)$ records can be sorted
  - E.g., $MS=100$MB, $BS=4$KB, $RS=100$ Bytes
    - $\Rightarrow \sim 2,000,000,000$ records ($\sim 100$ bytes) $\Rightarrow \sim 2.4$ terabyte
  - if that is not enough $\Rightarrow$ use more passes over the data
Physical Query Plan Operators: Categories

- Goal: minimize number of passes: one-pass algorithms
- unary, tuple-at-a-time: selections, projections
  - do not require seeing all/most of R
  - cost: table-scan \( \sim B(R) \) (clustered) or \( \sim T(R) \) (unclustered)
  - for selections with index: retrieve only blocks with potential answers
- unary, full-relation: duplicate elimination, aggregation
  - require seeing all/most of R
  - aggregation:
    - create one entry per group (=values of grouped attributes + current aggregated value)
    - scan R, one block at a time
    - first output deliverable only after all of R has been seen (does not fit nicely with iterator concept)

- binary, full-relation: join, union, difference, ...
- Nested-Loop Join:

```plaintext
for each tuple r of R do
  for each tuple s of S do
    if r and s join then output (r,s)
```

- improvement: instead of accessing R, S tuple-at-a-time, use read them in blocks:

```plaintext
for each block BR of R do
  for each block BS of S do
    if r of BR and s of BS join then output (r,s)
```

Physical Query Plan Operators: Nested-Loop Join

- binary, full-relation: join, union, difference, ...
- Nested-Loop Join:

```plaintext
for each tuple r of R do
  for each tuple s of S do
    if r and s join then output (r,s)
```

- improvement: instead of accessing R, S tuple-at-a-time, use read them in blocks:

```plaintext
for each block BR of R do
  for each block BS of S do
    if r of BR and s of BS join then output (r,s)
```

Merge Join

1. if R and S not sorted, sort them
2. \( i := 1 \); \( j := 1 \);
   while \( i \leq T(R) \) \&\& \( j \leq T(S) \) do
     if \( R[i].C = S[j].C \) then outputTuples
     else if \( R[i].C > S[j].C \) then \( j++ \)
     else if \( R[i].C < S[j].C \) then \( i++ \)

procedure outputTuples
while \( R[i].C = S[j].C \) \&\& \( i \leq T(R) \) do
  \{ \( j := j; \)
  while \( R[i].C = S[j].C \) \&\& \( j \leq T(S) \) do
    output pair \( R[i], S[j] \);
    \( j++ \)
\}
```

Example

<table>
<thead>
<tr>
<th>R[i].C</th>
<th>S[j].C</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>40</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>60</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>70</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

If R and S not sorted, sort them
\[ i := 1; \] \[ j := 1; \]
while \( i \leq T(R) \) \&\& \( j \leq T(S) \) do
  if \( R[i].C = S[j].C \) then outputTuples
  else if \( R[i].C > S[j].C \) then \( j++ \)
  else if \( R[i].C < S[j].C \) then \( i++ \)
**Index Join**

Let \( \text{index}(R, A, V) \) be an index access to \( R \) selecting all tuples where attribute \( A \) has value \( V \), i.e., \( \text{index}(R, A, V) = \sigma_{A=V}(R) \)

```plaintext
for each \( r \in R \) do
  \{ SI := index(S, C, r.C) \} // get all matching s's
  for each \( s \in SI \) do
    output \( r, s \) pair
```

**Hash Join**

- hash function \( h \), range 0...k
- buckets for \( R \): \( R_0, R_1, \ldots, R_k \)
- buckets for \( S \): \( S_0, S_1, \ldots, S_k \)

- hash \( R \) tuples into \( R_i \) buckets;
- hash \( S \) tuples into \( S_i \) buckets;
- for \( i = 0 \) to \( k \) do
  - match tuples in \( R_i, S_i \) buckets

**Hash Join: (Simplified) Example**

<table>
<thead>
<tr>
<th>R</th>
<th>S</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>11</td>
</tr>
</tbody>
</table>

- even
- odd

R even: \[ 2, 4, 8, 14 \]
R odd: \[ 3, 9 \]
S even: \[ 4, 12, 8, 14 \]
S odd: \[ 3, 13, 11 \]
Estimating Cost (#Disk I/Os): Nested-Loop Join

- Running example:
  - \( T(R) = 10,000 \) \( T(S) = 5,000 \)
  - \( S(R) = S(S) \) s.t. 10 tuples fit in a block
  - available memory: 101 blocks
- Nested-Loop Join (first: non-clustered relations)
  - for each \( R \) tuple: (read tuple + read \( S \))
  - \#IOs = 10,000 (1+5,000) = 50,010,000
- improvements:
  - read 100 blocks (* 10 tuples = 1,000 tuples) of \( R \) at a time:
    \[ \Rightarrow \text{#IOs} = \frac{10,000}{1,000} (1,000+5,000) = 60,000 \]
  - ... and reverse join order: \( S \leftarrow R \)
    \[ \Rightarrow \text{#IOs} = \frac{5,000}{1,000} (1,000+10,000) = 55,000 \]

Nested-Loop Join: Clustered Relations

- Nested-Loop Join \( S \Join R \) (now: clustered relations)
  ```
  for i = 1 to T(S)/1000 {
    read 100 blocks * 10 tuples from \( S \);
    for j = 1 to T(R)/10 {
      read 1 block * 10 tuples from \( R \);
      join
    }
  }
  ```
  - before: \#IOs = 5,000/1,000 (1,000+10,000) = 55,000
  - now: \#IOs = 5,000/1,000 (100+1,000) = 5,500

Merge Join

- assume \( R, S \) ordered by \( C \), relations clustered:
  \[ \Rightarrow \text{each relation block is read only once} \]
  \[ \Rightarrow \text{#IOs} = \text{read 1000 R blocks + read 500 S blocks} = 1,500 \]
- what if \( R, S \) are not ordered?
  \[ \Rightarrow \text{use 2-phase multiway merge-sort:} \]
  ```
  each tuple is read and written twice \[ \Rightarrow 4 \times (1000+500) = 6,000 \]
  \[ \Rightarrow \text{total #IOs} = 6,000 \text{ (sort)} + 1,500 \text{ (join)} = 7,500 \]
- Q: how to improve on the 7,500 I/Os? (hint: join earlier)
**Merge Join vs Nested-Loop: Scalability**

- Pump up R, S by a factor of 10: $T(R) = 100,000$, $T(S) = 50,000$

\[
\text{for } i = 1 \text{ to } T(S)/1000 \{
\text{read 100 blocks } \times 10 \text{ tuples from } S;
\text{for } j = 1 \text{ to } T(R)/10 \{
\text{read 1 block } \times 10 \text{ tuples from } R;
\text{join}
\}\}
\]

*nested loop* (clustered):

$\Rightarrow \#\text{IOs} = 50,000/1,000 (100+10,000) = 505,000$

*merge join* (including sort):

$\Rightarrow \#\text{IOs} = (4 \text{ (sort)} + 1 \text{ (join)}) \times (10,000+5000) = 75,000$

---

**Index Join**

- Assume: in-memory index on R.C and S clustered, unordered

\[
\text{for each } s \in S \text{ do }
\]

\[
\{ \text{RI } \Rightarrow \text{index (R, C, s.C)}
\text{for each } r \in \text{RI do }
\text{output s.r pair}
\} \]

- For each S tuple:
  - if (index(R,C,s.C) => match) then read matching R tuple(s)

- How many R tuples match?
  - (a) R.C key and S.C foreign key => 1 match
  - (b) $V(R,C)=5,000$, $T(R)=10,000$, uniform distribution => ~2 matches
  - (c) $DOM(R,C)=1,000,000$ => ~1/100 matches

\[\#\text{IOs} = B(S) + T(S) \times \text{matches}:
\]

- (a) $\#\text{IOs} = 500 + 500 \times 1 = 5,500$
- (b) $\#\text{IOs} = 500 + 500 \times 2 = 10,500$
- (c) $\#\text{IOs} = 500 + 500 \times 100 = 550$

- $Q$: estimate cost if probing the index is not free (e.g. index(R)
  needs 201 blocks)
Hash Join

- R, S are clustered (but unordered)
- use 100 buckets:
  - block-read R; hash into buckets; write buckets
  - similar for S
  - read R buckets, match with corresponding S buckets

- ideal distribution across buckets:
  \[ \text{IOs} = (2 \times \text{"bucketize"}) + 1 \times \text{"join"}) \times (1000+500) = 4,500 \]

Summary: Join Algorithms

- Summary (good practice/rules of thumb):
  - nested loop: only for small relations
  - for equi-joins (R.A=S.B) on unsorted, non-indexed relations => hash join
  - for non-equi joins (R.A>S.B): sort/merge join
  - sorted relations => merge join
  - indexed relations => consider index join (but estimate expected result size!)

Summary Query Processing

- use algebraic transformations to rewrite initial logical plan into better ones
- use estimates on the size of relations for reordering of associative and commutative operators (join!) and choosing good plans
- finding good plans is a search problem
  (common techniques based on dynamic programming, greedy approaches, branch and bound... are used)
- plan generator should also consider pipelining vs. materialization of intermediate results