Query Processing - Join Operation

Evaluate the condition join $R \bowtie_C S$.

1. 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
```

   - R is called the outer and S the inner relation of the join.
   - Requires no indexes and can be used with any kind of join condition
   - Worst case: db buffer can only hold one block of each relation
     - $B_R + N_R * B_S$ disk accesses
   - Best case: both relations fit into db buffer
     - $B_R + B_S$ disk accesses.
2. Block Nested-Loop Join

   An improvement of Nested-Loop Join.

```
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
```

   - Within each pair of blocks, every tuple in one block is paired with every tuple in the other block, to generate all pairs of tuples. All pairs of tuples that satisfy the join condition are added to the result.
   - The primary difference in cost between the block nested-loop join and the basic nested-loop join is that, in the worst case, each block in the inner relation s is read only once for each block in the outer relation, instead of once for each tuple in the outer relation.
   - Worst case: db buffer can only hold one block of each relation
     - $B_R + B_R * B_S$ disk accesses.
   - Best case: both relations fit into db buffer
     - $B_R + B_S$ disk accesses.
   - Some improvements of block nested-loop algorithm
     - If **equi-join** attribute is the key on inner relation, stop inner loop with first match
     - Use M − 2 disk blocks as blocking unit for outer relation, where M = db buffer size in blocks; use remaining two blocks to buffer inner relation and output. Reduces number of scans of inner relation greatly.
     - Scan inner loop forward and backward alternately, to make use of blocks remaining in buffer (with LRU replacement strategy)

        - Use index on inner relation, if available . . .

3. Indexed Nested-Loop Join
    - In a nested-loop join, if an index is available on the inner loop's join attribute, index lookups can replace file scans.
    - It is even possible (reasonable) to construct index just to compute a join.
    - For each tuple t_r in the outer relation r, the index is used to look up tuples in s that will satisfy the join condition with tuple t_r.
    - Worst case: db buffer has space for only one page of R and one page of the index associated with S:
        - BR disk accesses to read R, and for each tuple in R, perform index lookup on S.
        - Cost of the join: $B_R + N_R * c$, where c is the cost of a single selection on S using the join condition.
    - If indexes are available on both R and S, use the one with the fewer tuples as the outer relation.
    - Example:
        - Compute CUSTOMERS ⋈ ORDERS, with CUSTOMERS as the outer relation.
        - Let *ORDERS* have a primary B+-tree index on the join attribute *CName*, which contains 20 entries per index node
        - Since *ORDERS* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data records (based on tuple identifier).
        - Since *NCUSTOMERS* is 5,000, the total cost is 250 + 5000 ∗ 5 = 25,250 disk accesses.
        - This cost is lower than the 100,250 accesses needed for a block nested-loop join.