Statistical Information for Cost Estimation

- \(n_r\): number of tuples in a relation \(r\).
- \(b_r\): number of blocks containing tuples of \(r\).
  - \(l_r\): size of a tuple of \(r\).
  - \(f_r\): blocking factor of \(r\) — i.e., the number of tuples of \(r\) that fit into one block.
- \(V(A, r)\): number of distinct values that appear in \(r\) for attribute \(A\); same as the size of \(\Pi_A(r)\).
- If tuples of \(r\) are stored together physically in a file, then:
  \[ b_r = \left\lfloor \frac{n_r}{f_r} \right\rfloor \]
More detailed $V(A,r)$ statistics: Histograms

- Histogram on attribute *age* of relation *person*

- **Equi-width** histograms
- **Equi-depth** histograms break up range such that each range has (approximately) the same number of tuples
  - E.g. (4, 8, 14, 19)
- Many databases also store *n* most-frequent values and their counts
  - Histogram is built on remaining values only

![Histogram on age of person](image)

**What do we do with this?**

- We have tree with costs
  - Compute number of IOs for each operation based on estimates
  - Choose the best plan

![Tree with costs](image)
Cost-Based Optimization

- Consider finding the best join-order for \( r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n \).
- There are \( (2(n - 1))!/(n - 1)! \) different join orders for above expression. With \( n = 7 \), the number is 665280, with \( n = 10 \), the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of \( \{ r_1, r_2, \ldots r_n \} \) is computed only once and stored for future use.

Dynamic Programming in Optimization

- To find best join tree for a set of \( n \) relations:
  - To find best plan for a set \( S \) of \( n \) relations, consider all possible plans of the form: \( S_1 \bowtie (S - S_1) \) where \( S_1 \) is any non-empty subset of \( S \).
  - Recursively compute costs for joining subsets of \( S \) to find the cost of each plan. Choose the cheapest of the \( 2^n - 2 \) alternatives.
  - Base case for recursion: single relation access plan
    - Apply all selections on \( R_i \) using best choice of indices on \( R_i \)
    - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
  - Dynamic programming
Join Order Optimization Algorithm

Procedure findbestplan(S)
    if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
    // else bestplan[S] has not been computed earlier, compute it now
    if (S contains only 1 relation)
        set bestplan[S].plan and bestplan[S].cost based on the best way
        of accessing S using selections on S and indices (if any) on S
    else for each non-empty subset S1 of S such that S1 ≠ S
        P1 = findbestplan(S1)
        P2 = findbestplan(S - S1)
        for each algorithm A for joining results of P1 and P2
            ... compute plan and cost of using A (see next page) ...
            if cost < bestplan[S].cost
                bestplan[S].cost = cost
                bestplan[S].plan = plan;
        return bestplan[S]
Left Deep Join Trees

- In **left-deep join trees**, the right-hand-side input for each join is a relation, not the result of an intermediate join.

![Diagram of left-deep and non-left-deep join trees](image)

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of $n$ relations:
  - Consider $n$ alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Modify optimization algorithm:
    - Replace “for each non-empty subset $S_1$ of $S$ such that $S_1 \neq S$”
    - By: **for each** relation $r$ in $S$
      - let $S_1 = S - r$.
  - If only left-deep trees are considered, time complexity of finding best join order is $O(n \cdot 2^n)$
    - Space complexity remains at $O(2^n)$
  - Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small $n$, generally < 10)
Structure of Query Optimizers

- Many optimizers consider only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick “best” relation to join next
    - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimization
  - E.g., nested subqueries

Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - Heuristic rewriting of nested block structure and aggregation
    - Followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
  - Optimization cost budget to stop optimization early (if cost of plan is less than cost of optimization)
  - Plan caching to reuse previously computed plan if query is resubmitted
    - Even with different constants in query
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries