Recap

Cardinality (Size) Estimation

- **Most of the operators are straightforward**
  - π(R), τ(R) : |R|
  - R U S : |R| + |S|
  - R x S : |R| * |S|
  - R ⋈ S : Identical to σ(R x S)…
- **Some are hard**
  - σ(R)
  - ɣ(R) & δ(R)
- **Selection : Compute Selectivity (or % tuples passed through)**
  - Generic (Default) Heuristic:
    - Selectivity = 0.5
    - Works … mostly well 70% of the time. Very brittle and liable to break things
    - **Be wary**: DBMSes actually do this!
  - R.A = [Const]
    - Idea 1:
      - Compute COUNT(*) for every value value of A
      - Gives exact selectivity
    - Idea 2
      - Min/Max COUNT(*)
      - Gives lower/upper bound on selectivity
    - Idea 3
      - Avg COUNT(*) === Min/Max(A) (for a continuous domain) + Total Count == # distinct values of A + Total Count
      - Gives selectivity in average case, assuming a uniform distribution
      - Selectivity = Total Count / # distinct values of A
    - Can we do better?

Selectivity Estimation

Other types of queries

- **R.A < [Const] (also works for others)**
  - **Idea**: Collect stats: Min/Max, and assume a uniform distribution of values
    - Selectivity = ([Const] - Min) / (Max - Min)
    - Works for continuous data (Floats)
- **R.A = R.B**
  - (the Equijoin condition)
  - **Idea 1**: Assume no correlation
    - Becomes identical to either R.A = const or R.B = const
    - For each row, you're testing whether R.B = Some specific, somewhat arbitrary value
    - **Both R.A and R.B** are an upper bound on the selectivity, so take whichever reduction gives you the lower value
    - Interesting, this magically works for foreign key relationships
- **C1 AND C2**
  - Assuming no correlation between C1 and C2: Selectivity(C1) • Selectivity(C2)

More complex ideas…

- **Idea 4: Intermediate… Build a Histogram**

            - Store COUNT(*) for smaller ranges
            - e.g., For 1 from 1-100, store 10 buckets: 1-10, 11-20, etc…
            - Equality predicates are exactly the same as before.
            - Range predicates:
                - If the whole bucket is in the range, the entire count is in the range
                - If part of the bucket is in the range, make a uniform distribution assumption **for the bucket**.
        - **Idea 5: Wavelets**
            - Ever seen an image on a webpage load and it's all blocky at first and then it gets clearer?
                - That's a progressive image.
                - How could we make a progressive histogram?
            - Overview
                - Start with a completely uniform distribution
                - What information do you need in order to go from this to a 2-bucket histogram?
                - Idea 1: Split Bucket Ranges Evenly (e.g., 1-100 becomes 1-50, 51-100)
                    - Only need to communicate one integer *Difference = (Left.Count - Right.Count)*
                    - You have *Total.Count = (Left.Count + Right.Count)*
                        - Left.Count = (Total.Count + Difference) / 2
                        - Right.Count = (Total.Count - Difference) / 2
                - Idea 2: Communicate *Median* value (e.g., { 1, 45, 47, 48, 60, 72, 91, 99 } becomes 1-48, 49-100)
                    - Guaranteed to have an equal count on either side.

Columnar Layouts

Row-based layouts

- **Store rows together**

Columnar-Layouts

- **Store attributes together**
- **Option 1: Array of VALUE (Index = ROWID)**
    - Values with the same ROWID "join" together
    - Key advantage: Can avoid loading multiple columns.
        - Advertising datasets == 1000s of columns or more
        - Costly if you only care about 5ish
- **Option 2: <ROWID, VALUE>**
    - Key advantage: Can reorder. Effectively a big secondary index.
    - Often want both ROWID -> VALUE and VALUE -> ROWID
    - Can Compress w/ Run-length encoding
- **Other reasons to use Arrays of values**
    - Easier SIMD
    - ROWID Joins become intersections of bit vectors
- **Reasons not to use columnar layouts**
    - Updates are expensive
    - Inserts are prohibitive