A Comparison of Three Algorithms for Building ST-Histograms

Julián Hernández, Rodrigo Ipince, José Muñiz

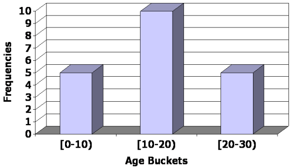
MIT

December 11, 2008

Histograms in Postgres

- One dimensional
- Independence assumption between fields in the same and different tables

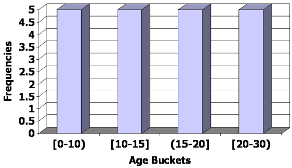
$$\texttt{SELECT * FROM students WHERE age} \leq 19$$



Histograms in Postgres

- One dimensional
- Independence assumption between fields in the same and different tables

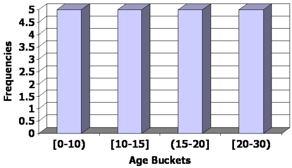
$$\texttt{SELECT * FROM students WHERE age} \leq 19$$

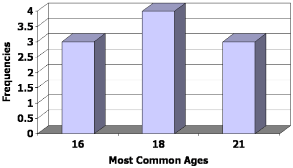


Histograms in Postgres

- One dimensional
- Independence assumption between fields in the same and different tables

`SELECT * FROM students WHERE age` ≤ 19

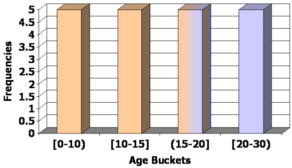


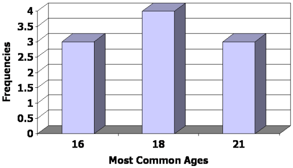


Histograms in Postgres

- One dimensional
- Independence assumption between fields in the same and different tables

SELECT * FROM students WHERE age ≤ 19

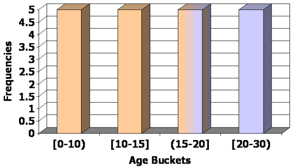


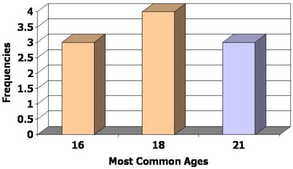


Histograms in Postgres

- One dimensional
- Independence assumption between fields in the same and different tables

SELECT * FROM students WHERE age ≤ 19



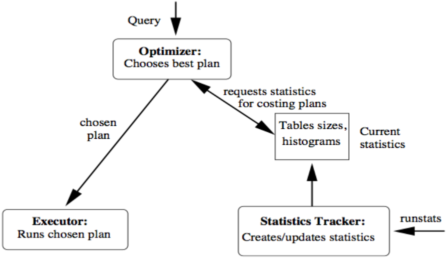


$$\sigma = \alpha \times \sigma_{hist} + \beta \times \sigma_{MCV}$$

Building histograms: Traditional cost-based optimization

Four components:

- Optimizer
- Executor
- Histogram
- Statistics gatherer



No connection between executor and histogram.

Some problems with traditional approach

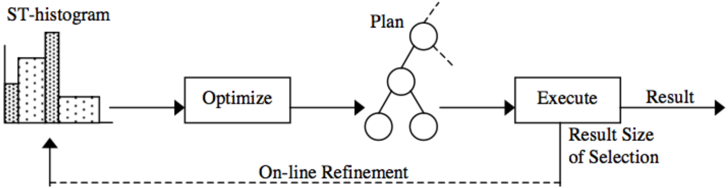
Key assumptions

- Uniformity of attribute values
- Uniformity of queries
- Constant number of records per block
- Random placement *or* full page scan

Problems:

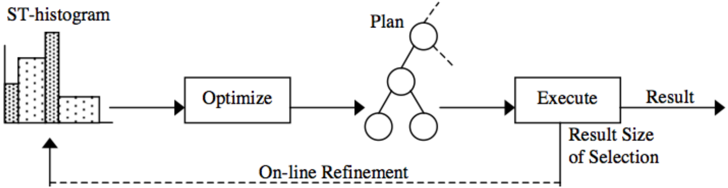
- Tradeoff between performance cost of statistics gatherer versus adaptability to change of conditions
- Tradeoff between number of pages read and adaptability to correlated data
- Postgres suggests turning off `autovacuum` for large tables and running end-of-day analysis.

Can we do something without needing to implement multidimensional histograms?



Self Tuning Histograms

Can we do something without needing to implement multidimensional histograms?



Interface

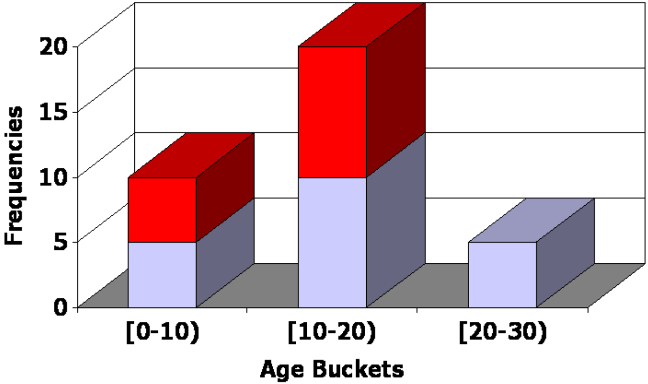
- Executor executes
  `emit([a...b], val)`
- Histogram builder provides
  `int estimate([a...b])`

Self Tuning Histograms: Interface and Motivation

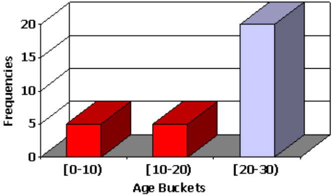
- How do we build the histogram without looking at data?
  - Update frequencies
  - Update buckets
- First idea: Uniform blame
  - Nothing known: assume uniform distribution
  - *Update frequencies:*
    On `emit`, calculate percentage error e, multiply all affected intervals by $\alpha \times p \times e$, where p is frequency proportion.
  - *Update buckets:*
    Split buckets with large frequencies.
    Merge buckets with small frequencies.

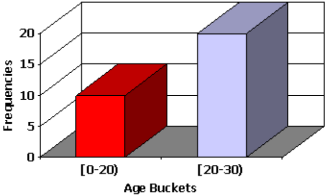
Updating frequencies

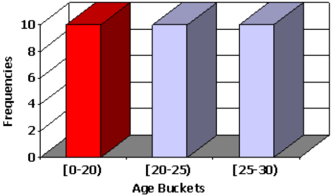
Result from `emit([0,19], 30)`



Merge and split







Improvements and alternatives

- Occasionally renormalize data to fit to total number of tuples. *Improves determination of error vs insertion*
- Adjust blame proportional to range and frequency
- Split most frequently updated ranges

Experimental Setup

- Middle layer between PostgreSQL and end user
- Process:
  1. Relay query to Postgres
  2. Obtain query plan via `EXPLAIN`
  3. Simulate query, using results for calling `emit`
  4. Calculate error rate in generated histograms

Results

- Fast to adapt
- Minimal insertion overhead (Small amount of tuples per query)
    - Ideal for large databases with constant insertions and varying ranges.

Caveats:

- Major optimizer misses due to independence assumptions, and not failures in row estimates.
- Not all cases lead to performance gains.
- No solution for `LIKE` queries.