A Comparison of Algorithms for Building ST-Histograms

José A. Muñiz Navarro

MIT

December 10, 2008

Histograms help database systems optimization

- Tables have fields
- One histogram per field.
- Selectivity helpful for optimizing queries

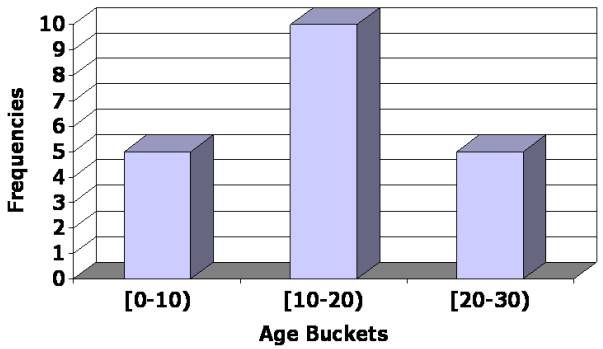
Need to sort by age

- Insertion sort?
- Merge sort?

ID	Age
1	17
2	21
3	18
4	19
5	18
6	21
7	17

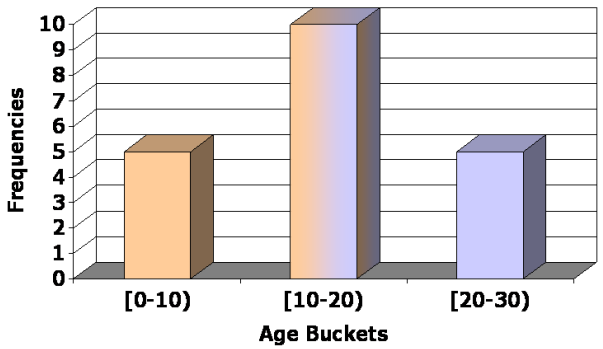
How to calculate selectivities from histograms

$$\texttt{FIND students WITH AGE age} \leq 14$$



How to calculate selectivities from histograms

$$\texttt{FIND students WITH AGE age} \leq 14$$

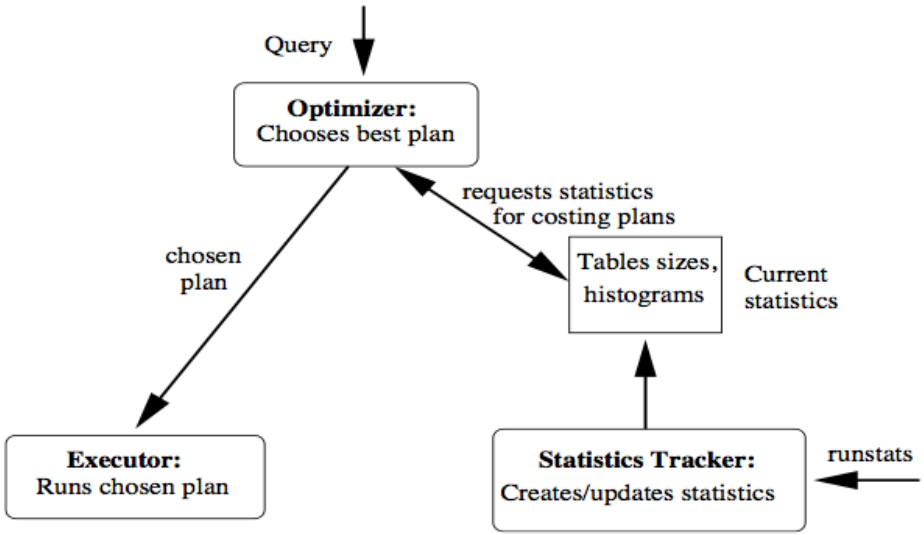


$$\sigma = 5 + \frac{10}{2} = 10$$

Building histograms: Traditional cost-based optimization

Four components:

- Optimizer
- Executor
- Histogram
- Statistics gatherer



No connection between executor and histogram.

Some problems with traditional approach

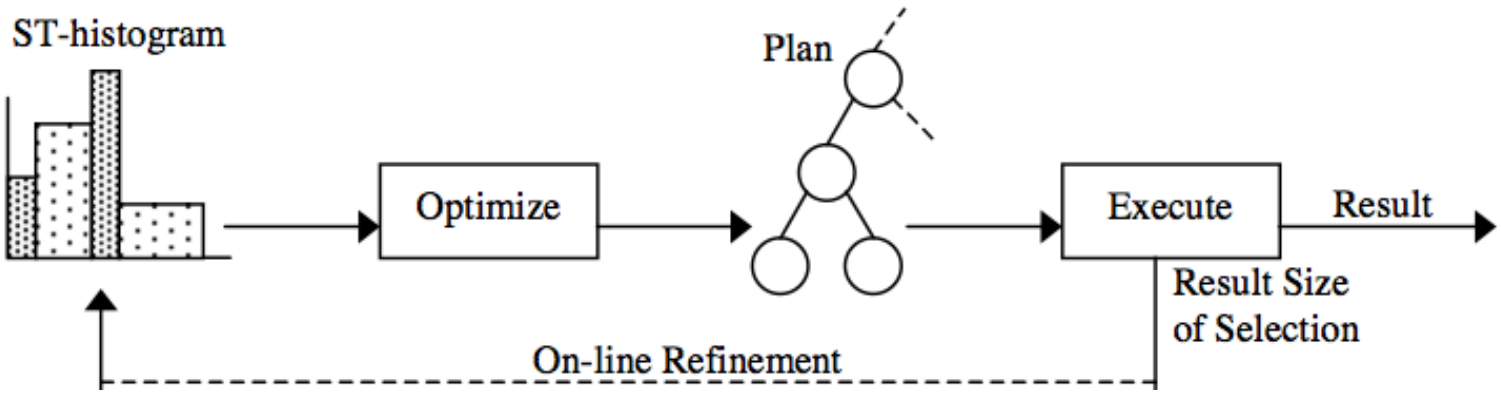
Problems:

- Tradeoffs
  - performance $\leftrightarrow$ adaptability
  - performance $\leftrightarrow$ precision
- Postgres suggests turning off statistics analyzer for large tables!

Idea: Incremental build via Self Tuning Histograms

Previous work by

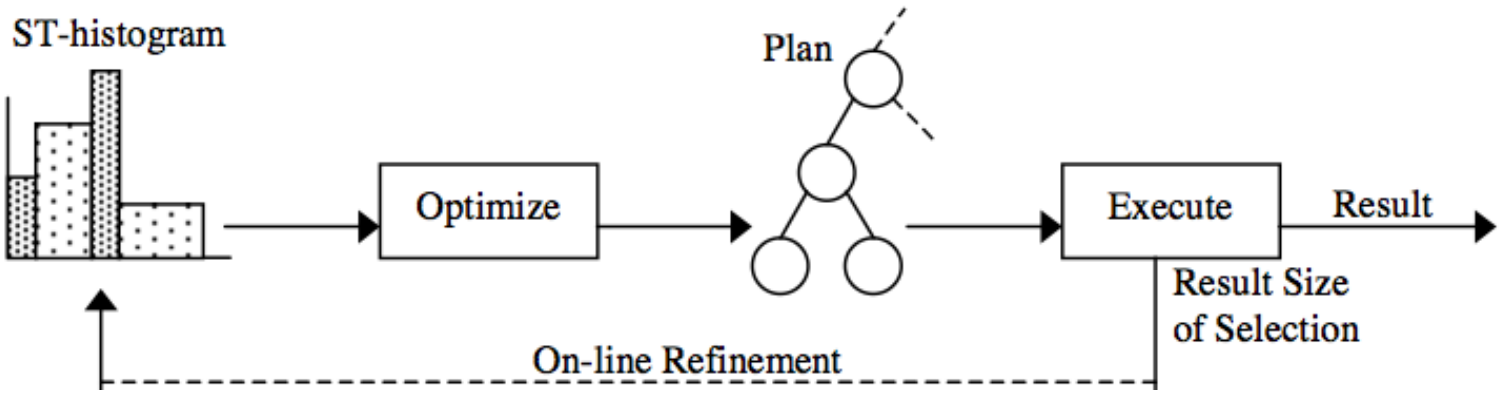
- AboulNaga and Chaudhuri - ST Histograms
- Babu, Bizarro - Adaptive Query Processing



Idea: Incremental build via Self Tuning Histograms

Previous work by

- Aboulnaga and Chaudhuri - ST Histograms
- Babu, Bizarro - Adaptive Query Processing



Interface

- Executor executes

  `emit([a...b], val)`

- Histogram builder provides

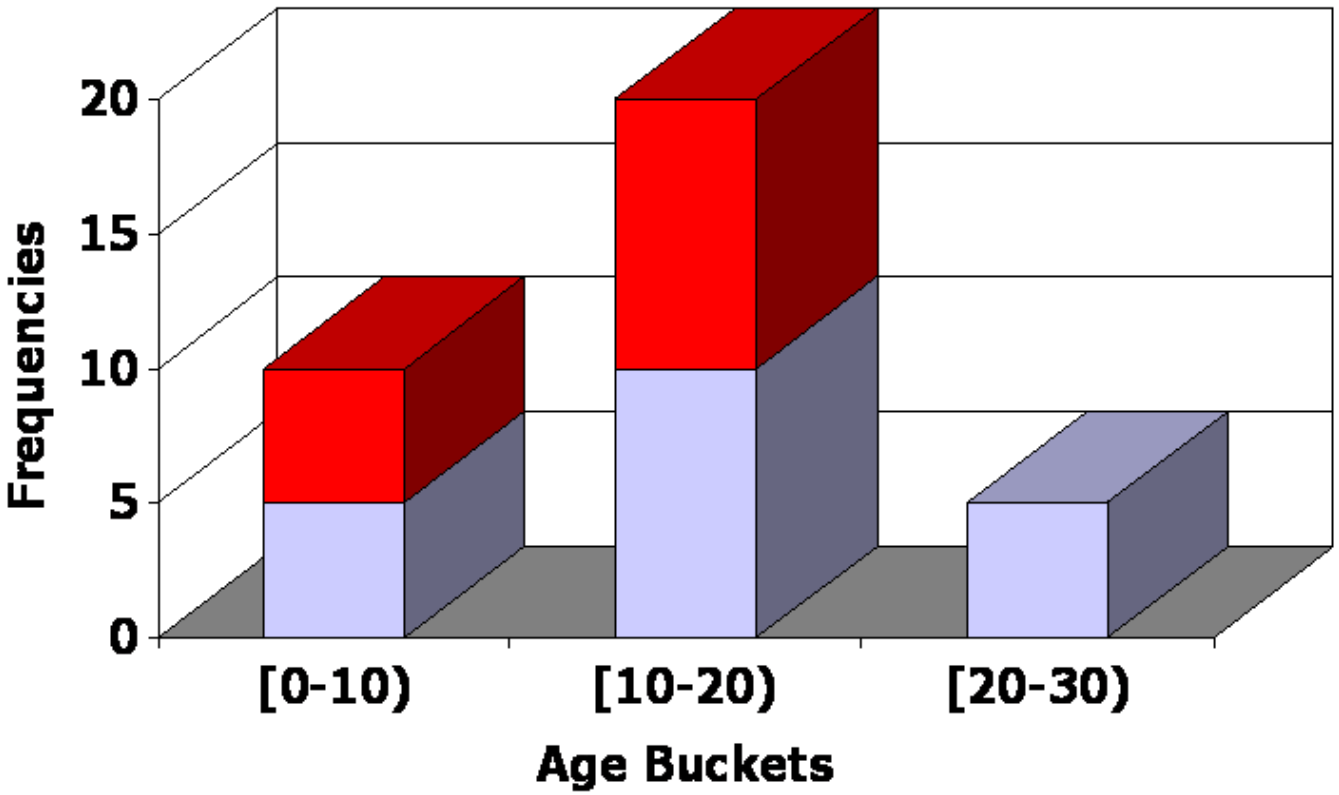
  `int estimate([a...b])`

Self Tuning Histograms: Interface and Motivation

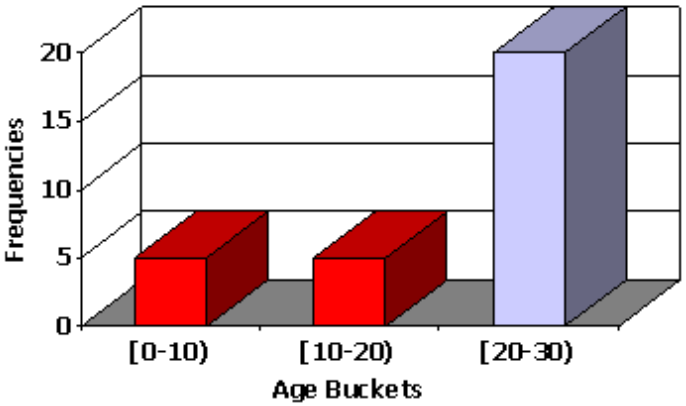
- How do we build the histogram without looking at data?
  - Update frequencies
  - Update buckets

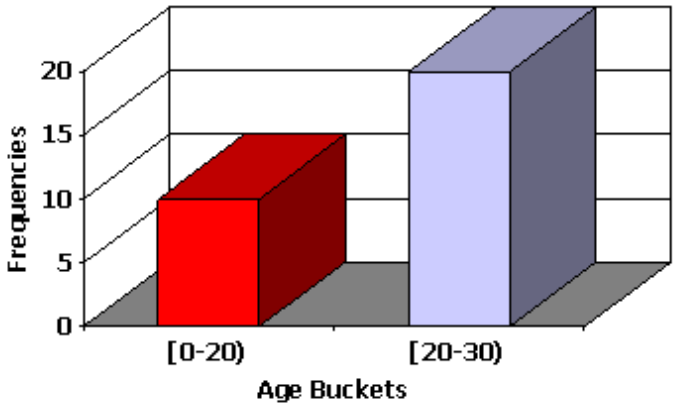
Updating frequencies

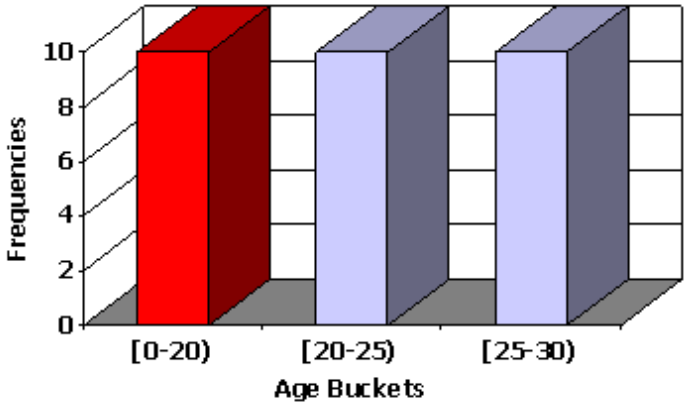
Result from `emit([0,19], 30)`



Merge and split







Proposed work

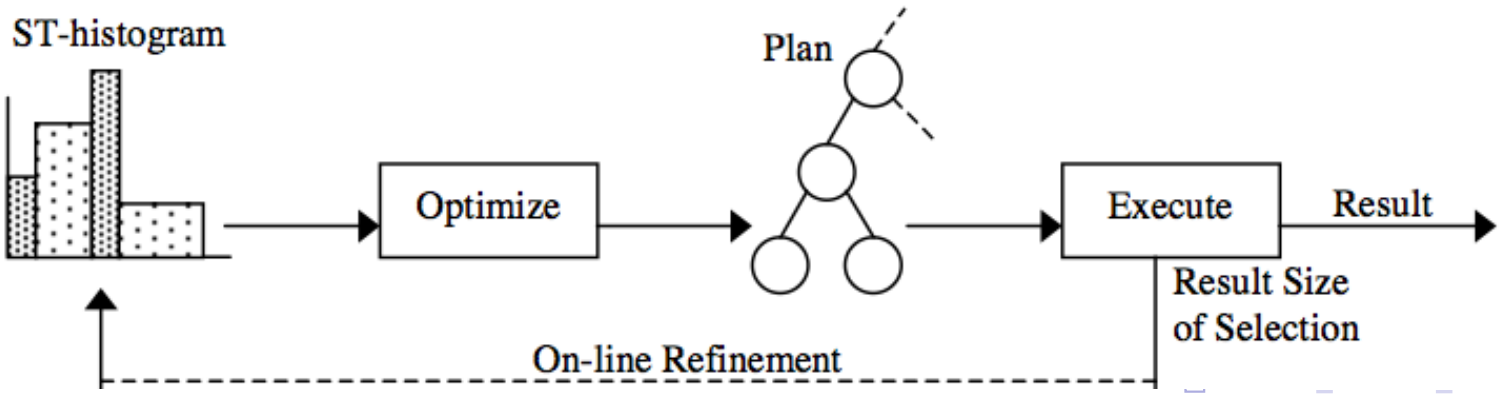
Determine feasibility of implementing ST Histograms in transactional databases, by benchmarking different ST algorithms against a common database system.

Algorithm modifications:

- 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

Proposed Setup

- Middle layer between database and end user
- Process:
  1. Relay query to database
  2. Obtain query plan from database
  3. Simulate query, using results for calling `emit`
  4. Calculate error rate in generated histograms
- Measure performance gains over common statistics gathering in transactional loads.



Expected 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.

- Week 1: Build parser
- Week 2: Build executor
- Week 3: Build histogram builder
- Week 4-5: Implement different histogram building algorithms
- Week 6-7: Implement TPC-C benchmark
- Week 8-10: Run statistics and gather data