# Datenbanksysteme II: Implementation of Database Systems

## Cost Estimation for Query Optimization

Material von

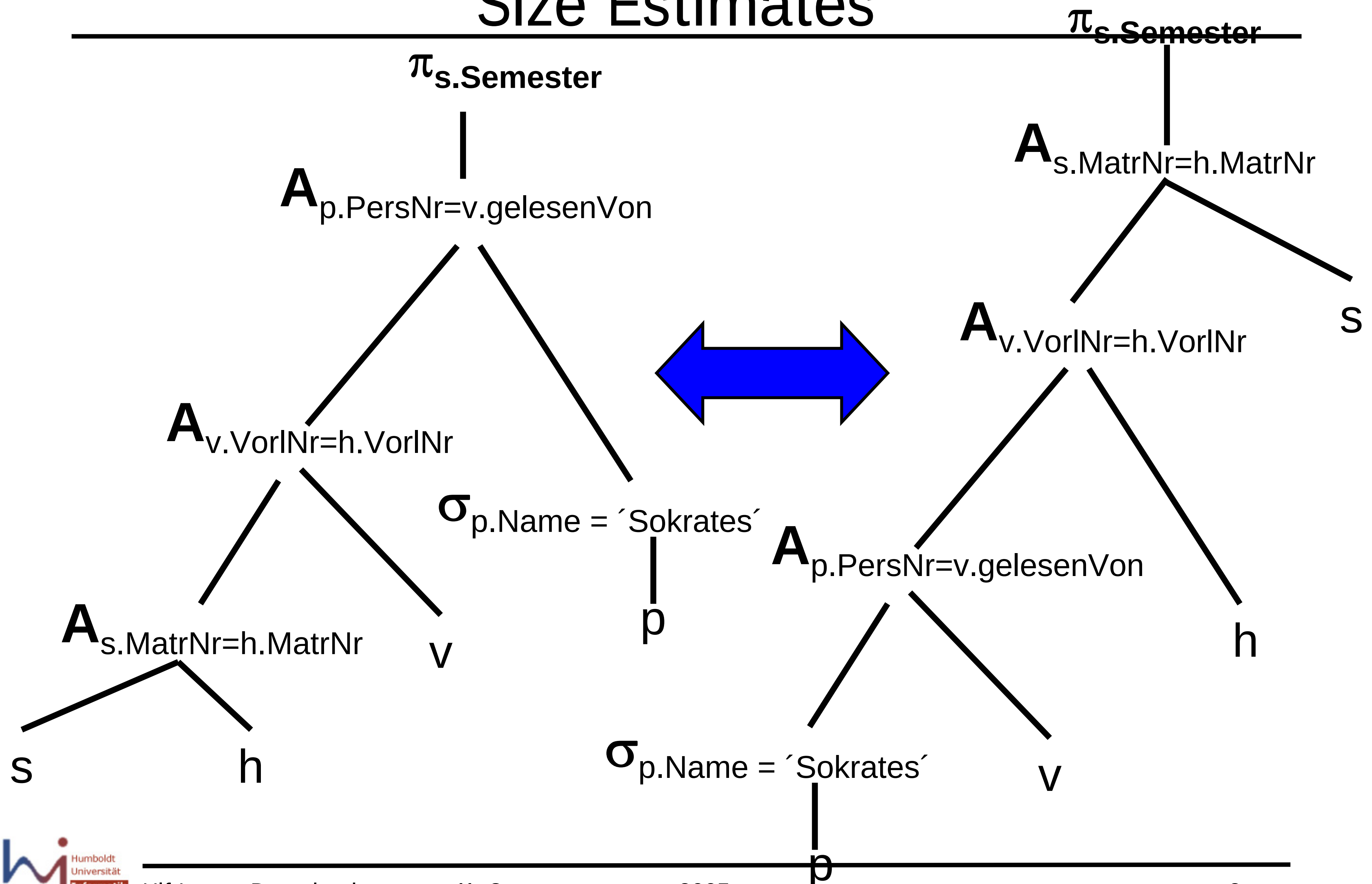
Prof. Johann Christoph Freytag

Prof. Kai-Uwe Sattler

Prof. Alfons Kemper, Dr. Eickler

Prof. Hector Garcia-Molina

# Order of Joins: Indistinguishable Without Size Estimates



# Choosing a Join Order

- Typical heuristic for pruning the search space
  - Consider only left-deep trees
  - Can be pipelined efficiently
  - Usually generates among the best plans
- Hence: Topology fixed, but still n! possible orders
- Find best using dynamic programming
  - Generate plans bottom up: Plans for pairs, triples, ...
  - For each concrete join group, keep only best plan
  - Use these to enumerate possibilities for larger join groups
  - Still very expensive problem
  - Use additional heuristics
    - Prune plans containing a cross product
    - Prune plans much worse than current best plan
    - ...

# Details

- Create a table containing for each join group
  - Estimated size of result (how: later)
  - Optimal cost for computing this group
    - For now, we simply take sum of sizes of intermediate results so far
  - Optimal plan for computing this group
- Induction over plan length = size of join group
  - i=1: Consider every relation in isolation
    - Size = Size of relation
    - Cost = 0 (assumption here)
  - i=2: Consider each relation pair
    - Size: Estimated size of “joining” both relations (might be product)
    - Cost = 0 (no int. res. so far due to previous assumption)
    - Fix an order (e.g.: smaller relation as inner relation)
      - This order will never change again
  - i=3: Consider each pair in triple and join with third relation
    - Consider only chosen order for pairs involved
    - ...

# Dynamic Programming

- DP is a heuristic
  - Assumption: Any subplan of an optimal plan is optimal
  - True for computing shortest paths, edit distance, knapsack???, ...
- But not true for join-order
  - Recall sort-merge join
  - Using a sort-merge join early in a plan might not be optimal for this particular join group
  - But result is sorted
  - Later joins can profit and also use sort-merge without sorting one intermediate relation again
- Solution
  - Keep different “optimal” plans for each join group
  - System R: One “optimal” plan per interesting sort order

# Ingredients

- We can evaluate different access paths for a single relation
- We can generate various equivalent relational algebra terms for computing a query
- We can optimize join order
  - Given selectivity estimates
- Query optimization =

  Search space (space of all possible plans)

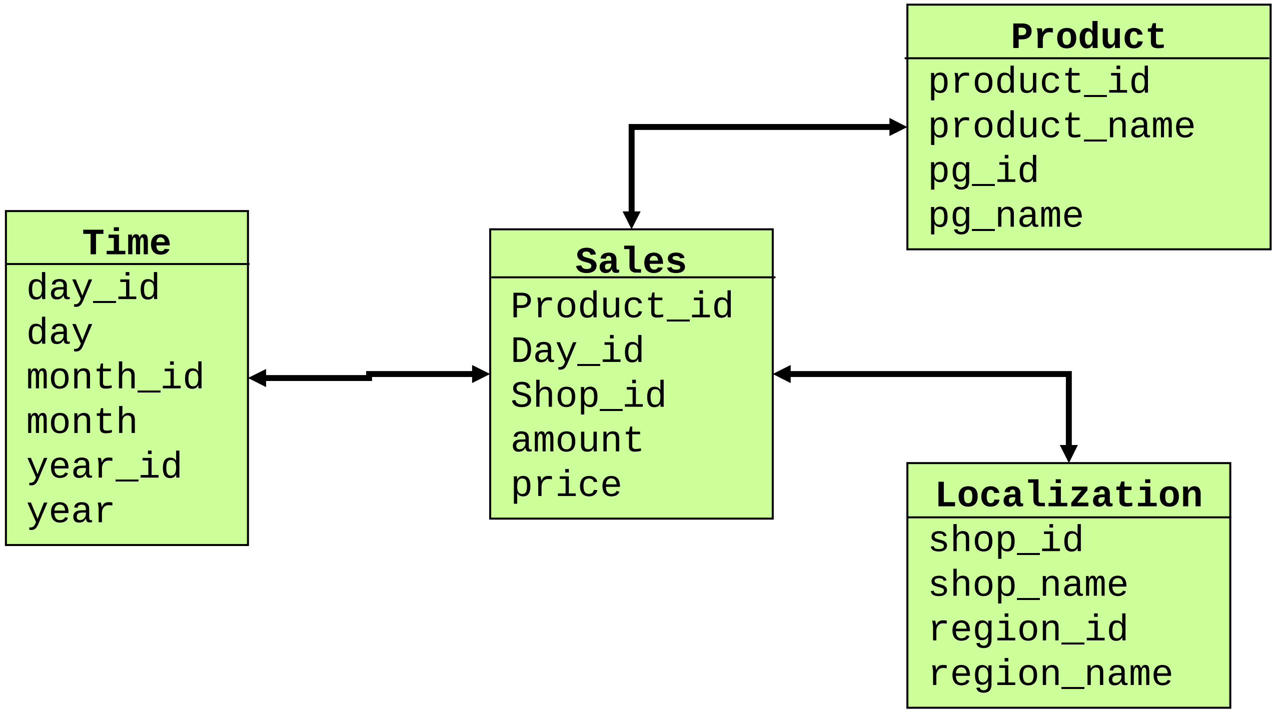
  +

  Search strategy (algorithm to enumerate all/some plans)

  +

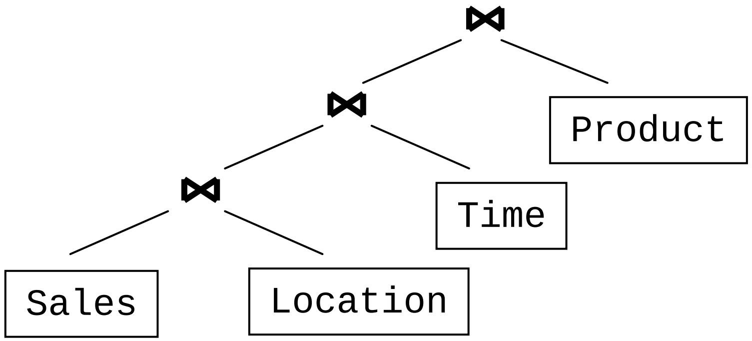
  Cost functions for evaluating and pruning plans (still missing)

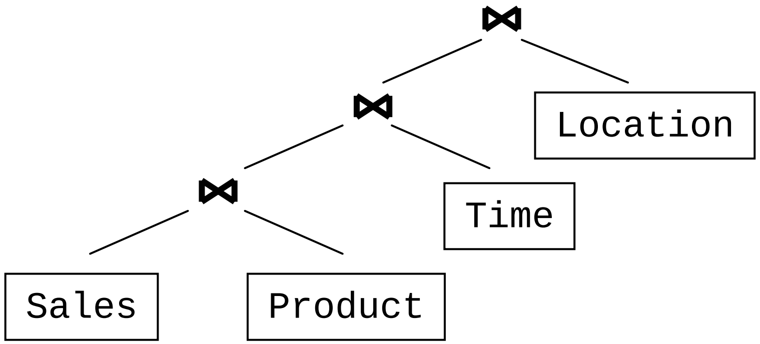
# Star Join




- Typische Anfrage gegen Star Schema
  - Aggregation und Gruppierung
  - Bedingungen auf den Werten der Dimensionstabellen
  - Joins zwischen Dimensions- und Faktentabelle

# Left-deep Pläne

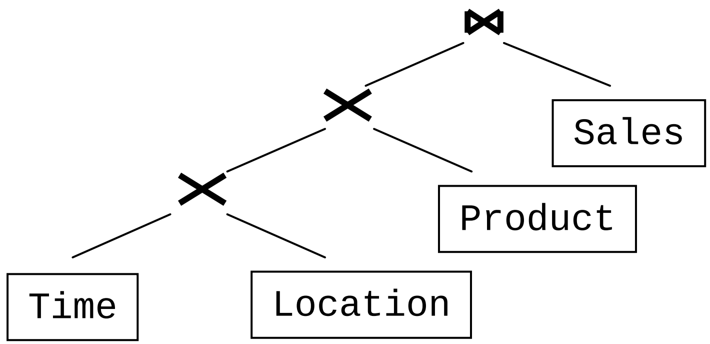



| | Zwischen-ergebnis |
|---|---|
| 1. Join (M / 15) | 6.666.666 |
| 2. Join ($\|J_1\|$*3/120) | 166.666 |
| 3. Join ($\|J_2\|$/50) | 3.333 |

| | Zwischen-ergebnis |
|---|---|
| 1. Join (M / 50) | 2.000.000 |
| 2. Join ($\|J_1\|$*3/120) | 50.000 |
| 3. Join ($\|J_2\|$/ 15) | 3.333 |

# Plan mit kartesischen Produkten




| | Zwischenergebnis |
|---|---|
| 1. Time x Location (3*20 * 100) | 6.000 |
| 2. … x Product ($\|P_1\|$*20) | 120.000 |
| 3. … ⋈ Sales | 3.333 |

- Es gibt mehr „Zellen“ als Verkäufe
- Nicht an jedem Tag wird jedes Produkt in jedem Shop verkauft

# STAR Join in Oracle 8i – 9i

- Neue STAR Join Strategie seit Oracle 8i
- Möglichkeit der (komprimierten) Bitmapindexe lässt kartesisches Produkt weniger vorteilhaft erscheinen
- Phasen
  1. Berechnung aller FKs in Faktentabelle gemäß Dimensionsbedingungen einzeln für jede Dimension
  2. Anlegen/laden von Join-Bitmapindexen auf allen FK Attributen der Faktentabelle
  3. Merge (AND) aller Bitmapindexe
  4. Direkter Zugriff auf Faktentabelle über TID
  5. Join nur der selektierten Fakten mit Dimensionstabellen zum Zugriff auf Dimensionswerte
- Zwischenergebnisse sind nur (komprimierte) Bitlisten

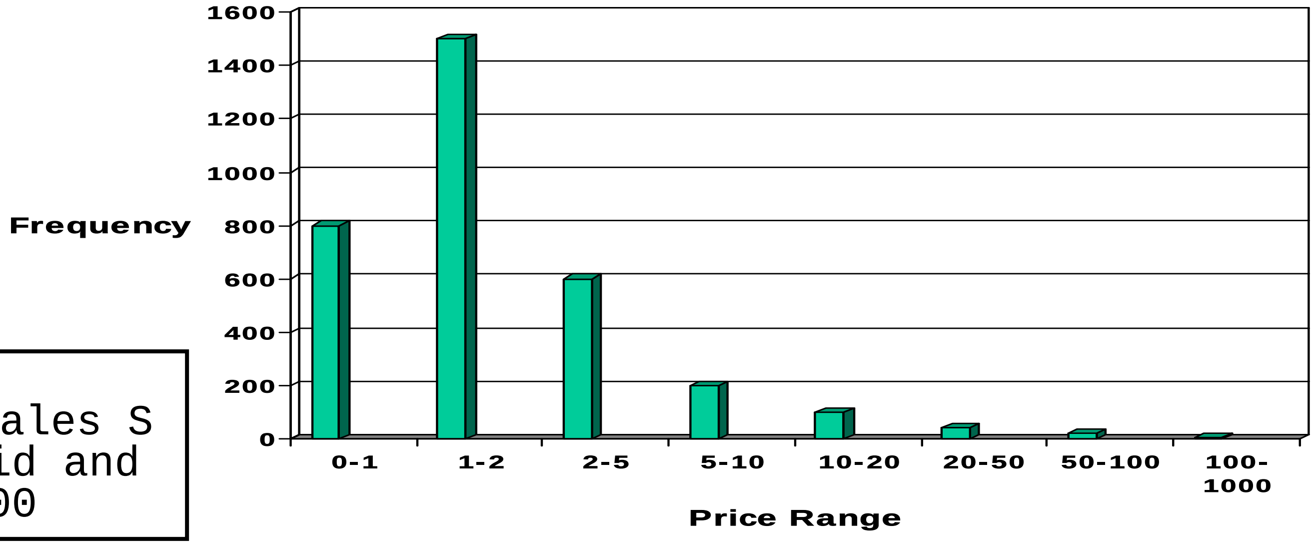
# Content of this Lecture

- Cost estimation
- Uniform distribution
- Histograms

# Cost Estimation

- Rule-based optimizer
  - Transformations depend only on schema elements, not on database instance (i.e., the actual data)
  - No notion of “plan cost”
    - Cannot differentiate join order
    - Cannot decide on access path selection / index usage
  - Needs heuristics for rule selection
- Cost-based optimizer
  - Estimate the cost of each operation
    - Estimated amount of IO
    - Estimated size of intermediate results
  - Requires: selectivity of conditions, space reduction of projections, size of joins and group-by’s, ...
- Cost estimation is important for
  - Choosing cheapest possibility for each single operations
  - Finding cheapest plan for entire query
    - Operations have non-local side-effects, especially order

# Example




```
SELECT *
FROM product p, sales S
WHERE   p.id=s.p_id and
        p.price>100
```

- Assume 3300 products, 1M sales, index on sales.p_id and product.id
- Uniform distribution
  - Price range is 0-1000 => selectivity of condition is 9/10
    - Expect 9/10*3300 ~ 3000 products
  - Choose BNL, hash, or sort-merge join (depending on buffer available)
- Using histograms
  - Assume 10 equi-width buckets (later)
  - Selectivity of condition is 5/3300 ~ 0,0015
  - Choose index-join: scan p, collect id of selected products,  use index to access sales
- Note: We are making another assumption – which??

# Cost Estimation

- Cost estimation is bottom-up
  - Start by building a model of the content of relations (attributes)
    - Model should be much smaller than relation (few values only)
    - Should allow for accurate predictions
    - Should be consistent – same size estimates for different plans of the same subquery
    - Model should be maintained when relation changes
    - Model should be generated quickly
    - Models need to be stored and accessed efficiently
  - Relational operations transform models of relations / intermediate results into models of results
- Example
  - Model=(count, min, max) for each attribute in each relation

Certainly wrong.
Consider PK/FK constraints

# Example

Acc#( 0, 1, 123456)

**Independence assumption: 1/123456 * 1/123456 = 6,5E-11%**

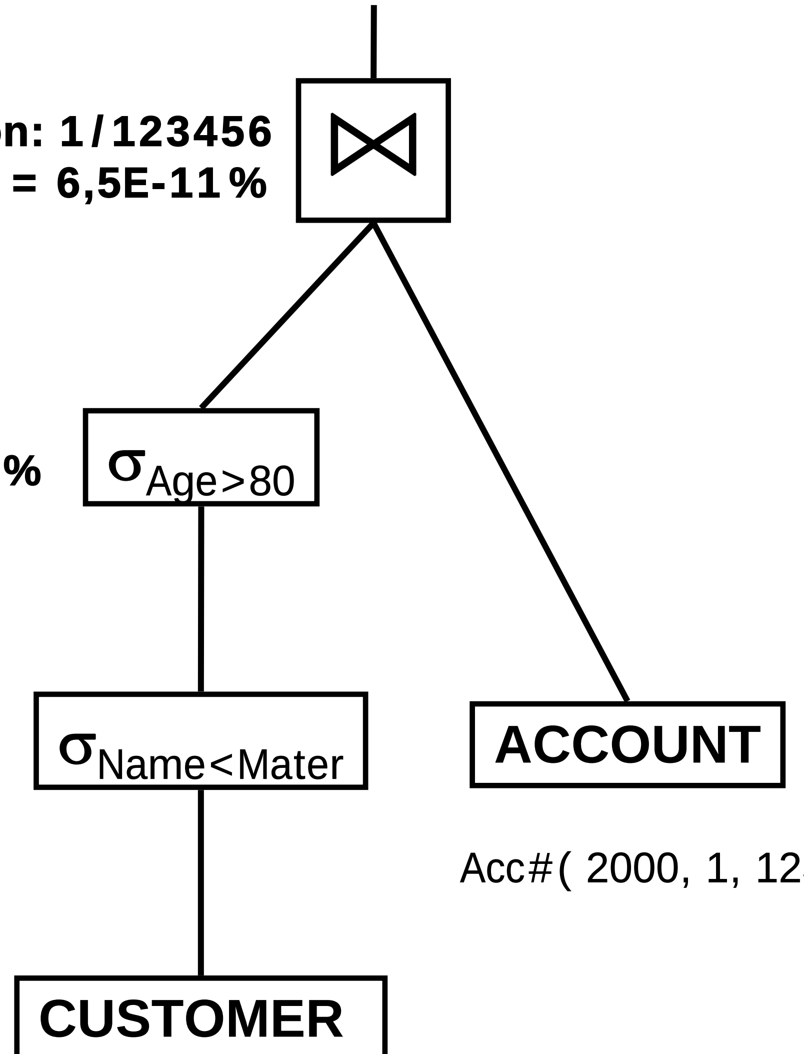
Name(112, Aare, Mater)
Age(112, 80, 98)
Acc#( 112, 1, 123456)

**Sel: 1/(89-18)*18= 22,5%**

Name(500, Aare, Mater)
Age(500, 18, 98)
Acc#( 500, 1, 123456)

**Sel: 50%**

Name(1000, Aare, Zyte)
Age(1000, 18, 98)
Acc#( 1000, 1, 123456)




Acc#( 2000, 1, 123456)

# Types of Models

- Uniform distribution of values
  - Very small model (count, max, min), simple to build
  - Bad predictions if assumption violated
- Histograms
  - Parameterized size, quite simple to build
  - Accuracy depends on type and size (and timeliness)
- Sampling
  - Use a representative subset of relation
  - Parameterized size, not so easy to chose a truly representative samples
  - Accuracy depends on selection method and size (and timeliness)

# Types of Models II

- Know the concrete distribution
  - Could be Gauss, Zipf, Poissoin, ...
  - Can be characterized by few parameters (mean, stddev, ...)
  - Very small model, but very expensive to check whether real distribution follows assumed model
  - Very accurate
- Describe values by function
  - Might be a very small model, very costly to build in general
  - Very accurate if good fitting function is found
  - Rarely used, since few value distributions can be described by simple functions (names? addresses?)
  - Used for approximations, e.g. wavelet transformations

# Content of this Lecture

- Cost estimation
- Uniform distribution
- Histograms

# Rules of Thumb

- Assume uniform distribution
- For relation R and attribute A, let
  - V(R, A) be number of distinct values of A
  - max(R, A), min(R, A) be the maximal/minimal value of A
    - Value that does exist, not maximal / minimal possible value
  - |R| be the number of tuples in R
  - Note: R may be an intermediate result
- Definition
  - The selectivity of a relational operation is the fraction of tuples of the input that will be in the output
- In the following, we often abbreviate with min=min(R, A), max=max(R, A)

# Size of a Selection

- Selection of the form "A=const"
  - |S| = |R| / v(R,A)
  - v(S,A) = 1; max(S,A)=min(S,A)=const
- Selection of the form "A<const"
  - If min<const<max
    - |S| = |R| / (max-min) * (const-min)
    - v(S,A) = v(R,A) / (max-min) * (const-min); min(S,A) = min; max(S,A)=const
  - Otherwise: |S|=v(S,A)=0; max(S,A)=min(S,A)=undef
  - Alternative: |S| = |R| / 3
    - Idea: With such queries, one usually searches for outliers
    - Very rough estimate, but requires no knowledge of R at all
- Selection of the form "A≠const"
  - |S| = |R| * (v(R,A)-1)/v(R,A)
  - Alternative: |S| = |R|
  - v(S,A)=v(R,A), min(S,A)=min, max(S,A)=max

# More than One Selection

- Selection of the form "$A\theta c_1 \wedge B\theta c_2 \wedge ...$"
  - Assumption: independence of values
    - That is not the same as uniform distribution assumption
  - Total selectivity is $sel(c_1) * sel(c_2) * ...$
  - v, min, max are adapted iteratively for each single condition
- Selection of the form "$A\theta c_1 \vee B\theta c_2 \vee ...$"
  - Rephrase into $\neg (\neg(A\theta c_1) \wedge \neg(B\theta c_2) \wedge ... )$
  - Selectivity is $1- (1-sel(c_1))*(1-sel(c_2))*...)$
- Selectivity of $A=10 \wedge A>10$ ??

# Projection and Distinct

- Selectivity of DISTINCT
  - |S| = v(R,A)
  - v(S,A)=v(R,A), min(S,A)=min, max(S,A)=max
- Selectivity of projection
  - Is 1 under BAG semantics
  - Is as selectivity of DISTINCT under SET semantics
  - Caution
    - In real life, we need to estimate the size of the intermediate relation
    - This requires number of tuples and size of tuples
    - We ignore(d) this issue
- Selectivity of grouping
  - Same as selectivity of distinct on group attributes
- Selectivity of `SELECT DISTINCT A, B, C FROM` …
  - Any ideas??

# Projection and Distinct

- Selectivity of `SELECT DISTINCT A, B, C FROM` …
  - Not easy
  - Clearly, $0 < |S| < v(R,A) * v(R,B) * v(R,C)$
  - Suggestion: $|S| = \min( ½*|R|, v(R,A)*v(R,B)*v(R,C))$
- Alternative
  - Multi-dimensional histograms (later)

# Selectivity of Joins

- Consider join $R \bowtie_A T$
  - Think of it as $\sigma_{R.A=T.A}$ (R x T)
  - Size if product is |R| * |T|, but what is selectivity of the condition?
- Same problem as for `DISTINCT A, B, C`
  - We do not know how values in R.A and T.A coincide
  - Would require conditional probabilities
- Suggestion
  - We assume that joins always are over PK / FK constraints
    - We cannot do this for GROUP BY ...
  - Thus, if v(R,A)<v(T,A), T.A is PK in T and R.A is FK
  - Each tuple from R will have |T|/v(T,A) joining tuples in T
- Together
  - |S|= |R| * |T| / max( v(R,A), v(T,A))
  - |R|<|T|: v(S,A)=v(R,A), min(S,A)=min(R,A), max(S,A) = max(R,A)
- What about $R \bowtie_{R.A<T.B} T$ ?

# Content of this Lecture

- Cost estimation
- Uniform distribution
- Histograms

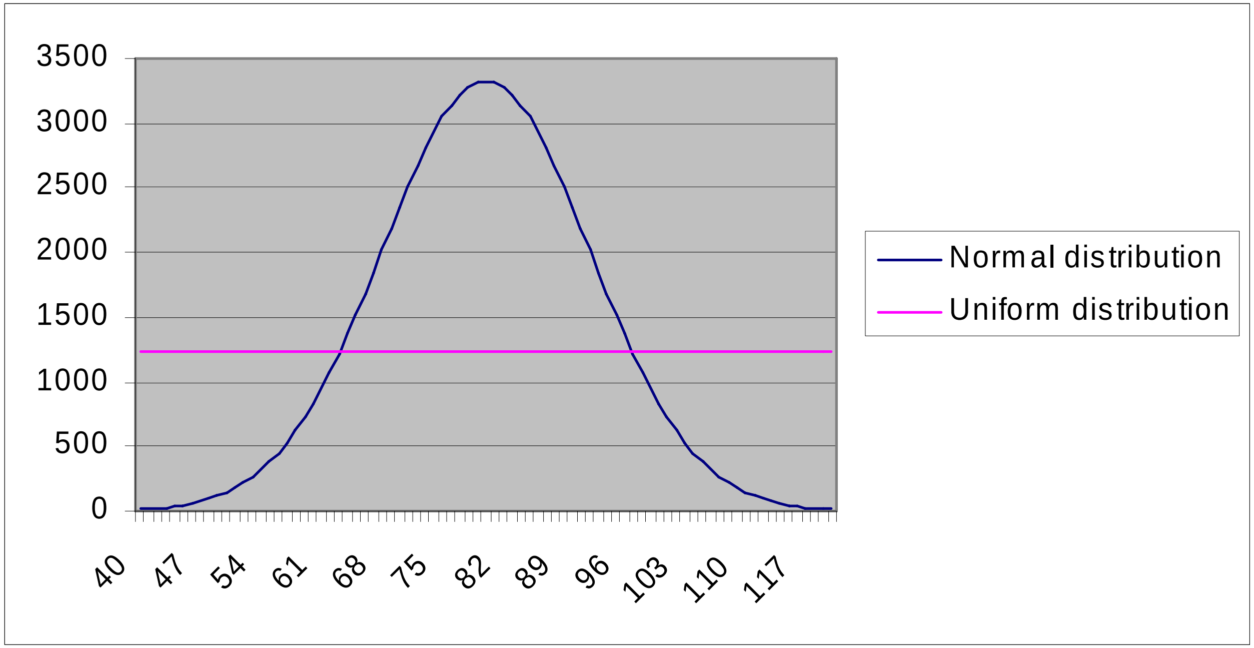
# Histograms

- Real data is rarely uniformly distributed
  - Neither Poissoin, Gauss, Extreme-Value, Zipf, ...
- Solution: Histograms
  - Partition the existing value range of an attribute into buckets
  - Count frequency of tuples in each bucket (i.e. range)
  - During cost estimation, approximate frequency of a single value in a bucket by the average over all values in this bucket
    - I.e., make uniform distribution assumption inside each bucket
    - Or use other assumptions, but always inside buckets
- Advantage
  - Lower errors due to consideration of smaller ranges
  - Hope: frequencies vary less inside smaller ranges
  - I.e., histograms do not help against completely erratic distributions
    - Erratic does not mean random – so what??

# Issues

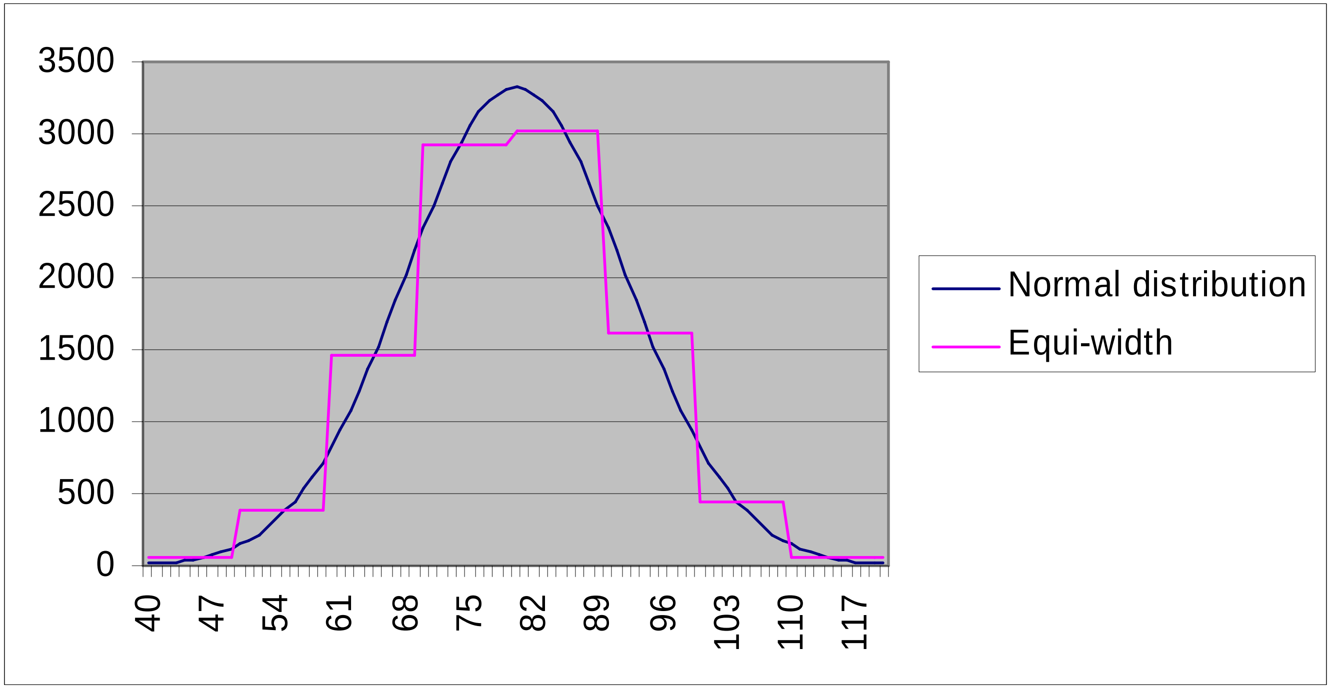
- We must think about
  - How should we chose the borders of buckets?
  - What do we store for each bucket (could be more than count)?
  - How do we keep buckets up-to-date?

# Distribution



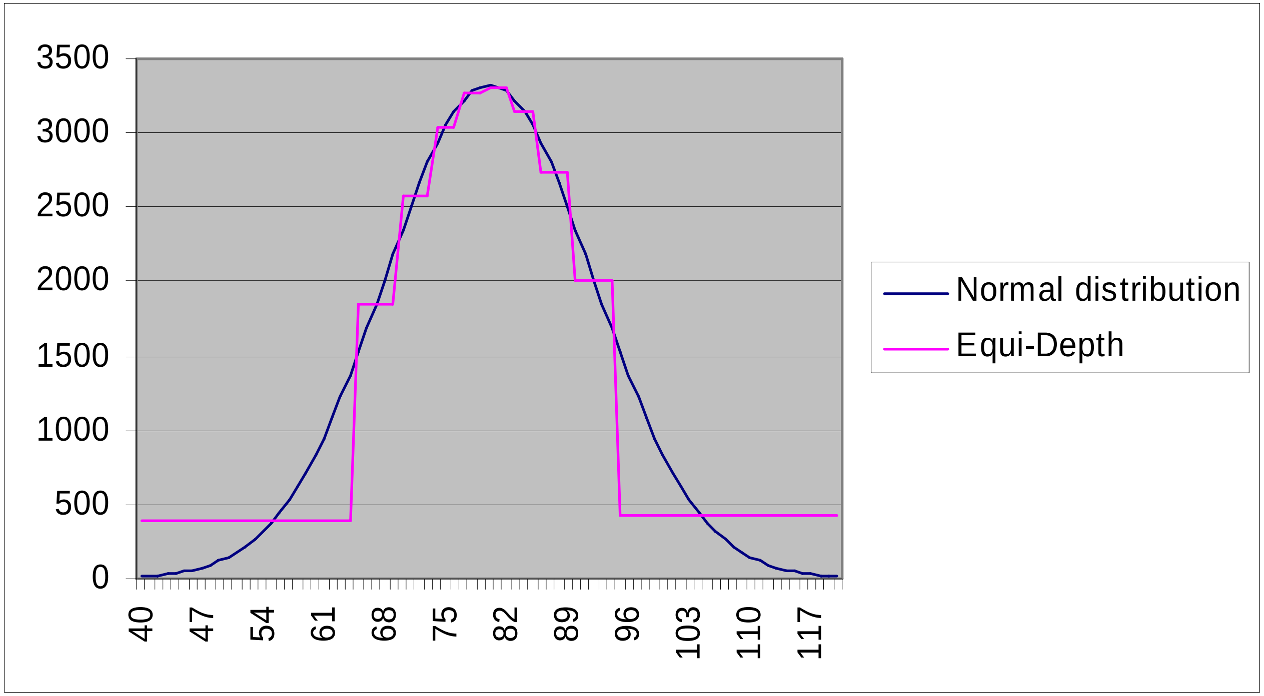

- Assume normal distribution of weights
  - Spread: 120-40=80, mean: 80, stddev: 12; 100000 people
- Uniform distribution: 100000/80=1250 for each possible weight
- Leads to large errors in almost all possible query ranges

# Equi-Width Histograms




- Fix n# of buckets; Borders are equi-distant (need not be stored)
- In each bucket, assume average frequency inside bucket
- Can be computed by scanning all values, keeping one count one bucket
- Remaining error depends on
  - Number of buckets (trade-off: more buckets, less errors, but more work)
  - Distribution of values in each bucket

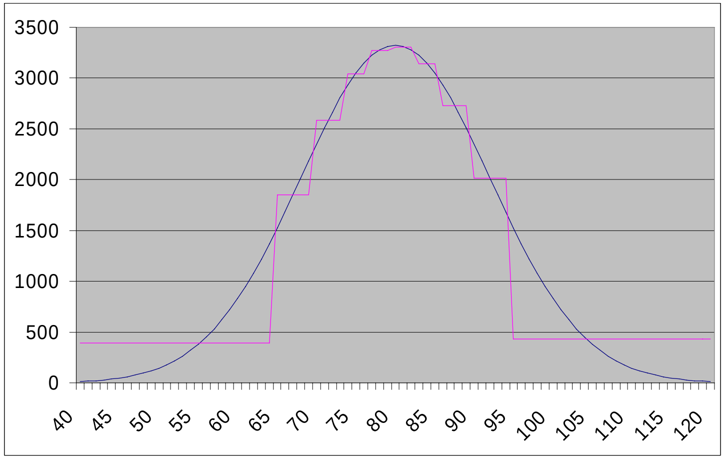
# Equi-Depth




- Fix n# of buckets
- Chose borders such that total frequency in each bucket is app. equal
  - Here: Roughly 10.000 persons in each bucket
  - Leads to buckets of varying size - borders need to be stored
  - Can be computed by sorting all values, then jump in equal steps
- Better fit to data

# Example

- Query: Number of people with weight between 65-70 (incl)
  - Real value: 11603
  - Uniform distribution: (70-65+1)*1250 = 7500
    - Error: 4103 ~ 35%
  - Equi-width histogram
    - Range 60-69 has average 1469
    - Range 70-79 has average 2926
    - Estimation: 5*1469 + 1*2926 = 10271
      - Error: 1332 ~ 11%
  - Equi-depth histograms
    - Range 65-69 has average 1850
    - Range 70-73 has average 2581
    - Estimation: 5*1850 + 1*2581 = 11831
    - Error: 228 ~ 2%




- Error depends on concrete value or range
  - On average, equi-depth histograms are better than equi-width histograms

# Other Types of Histograms

- There is a surprising wealth of different histograms
- Serial histograms
  - Sort values by frequency and build buckets as ranges of frequencies
  - Frequency ranges of different buckets do not overlap
  - Better fit, but values in bucket must be stored explicitly
    - No consecutive ranges
- V-optimal histograms
  - Sort values by frequency and build buckets such that weighted variance is minimized in each bucket
  - Provably best class of histograms for many queries
    - But costly generation and maintenance
    - Best known algorithm is $O(b*n^2)$ (n# values, n# buckets)
- End-biased histograms
  - Sort values by frequency and build singleton buckets for largest / smallest frequencies plus one bucket for all other values
  - Simple form of serial histograms, quite effective on many real-world data distributions (e.g. Zipf-like distributions)
- Commercial systems use equi-depth and compressed histograms (mixture of equi-depth and end-biased histograms)

# Histograms for Join Estimation

- Suppose histogram for both join attributes
  - Assume three tables: Product, Sales, and Supplier
    - Assume 1M sales, product IDs 1-1000, prices correlate with ID, and people mostly buy cheap products
    - Assume 2000 supplier, product IDs 1-1000, and few supplier for cheap products (A&P), yet many for costly products (exclusive brands)
  - Query: Whose products are sold most often = sales $\bowtie_{p\_id}$ supplier
  - Estimation without histograms
    - |R|= |Sales| * |Supplier| / max( v(Sales,p_id), v(Supplier,p_id)) = 1.000.000 * 2000 / max( 1000,1000) = 2.000.000
  - Assume 5 equi-width histograms with ranges 1-200, 200-400, ...
    - Sales: 850.000, 100.000, 45.000, 4.500, 500
    - Supplier: 10, 80, 210, 400, 1.300
    - Estimation of join result for each bucket pair: (850.000*10+100.000*80+45.000*210+4.500*400+500*1.300)/200~ 524.000
      - Only 1/200 of each combination from two buckets will quality for the join
- More complicated, if bucket borders do not coincide

# Histograms and Complex Conditions

- We only considered histograms for a single attribute
- Does not help to estimate selectivity of complex conditions
  - People with weight<30 and age<25 ?
  - People with income>1M and tax depth<500K ?
  - We need to know conditional distributions
  - Clearly, there is a combinatorial explosion in the number of combinations to consider
    - Also: Could be connected by AND, OR, AND NOT, ...
- Multidimensional histograms
  - Active research area
  - Need sophisticated storage structures – multidimensional indexes

# Maintaining Histograms

- Usually, computing histograms requires scanning a table
  - Potentially for each attribute
  - Cannot be done before each query
  - Indexes can help a lot
    - Statistics such as min, max are directly obtainable
    - Inner nodes of B+ trees ~ equi-depth histograms
    - But we rarely have indexes on all attributes of a relation
- Option 1: Compute once and maintain
  - Equi-width histograms
    - Simple; increase/ decrease total frequency in bucket upon insert/delete/update
  - Equi-depth histograms
    - Changes may influence borders of buckets
    - No simple and accurate method
    - Option 1: Proceed as for equi-width and regularly re-compute entire histogram
    - Option 2: Implement complex bucket merging/ splitting procedures

# Maintaining Histograms in Real Life

- Compute only on user request and leave unchanged in the meantime
  - Administrator need to trigger re-computation of all statistics from time to time
  - Otherwise, query performance may degrade
  - Both cases (new or outdated statistics) may lead to unpredictable changes in query behavior
    - To prevent, Oracle provides “query outlines”
  - Automatically maintaining statistics is a very active research topic
    - Self-optimizing, self-maintaining, self-healing, zero-administration, ...

# Sampling

- Scanning a table for computing a histogram is expensive
- But: We don't really need all values – we only want to estimate the value distribution
- Solution: Use a sample of the data
  - If chosen randomly, sample should have the same distribution as full data set
  - Very effective: Usually, a 10% sample suffices
- Also useful for approximately computing COUNT, AVG, SUM, etc.
  - Approximate query processing: Much faster answers in much less time with minimal error
  - Requires estimation of maximal error (confidence values)
  - Again: Very active research area ("Taming the terabyte")

# Problems with Sampling

- Problem
  - How we get a random, 10% sample?
  - Reading first 10% of rows is a very bad idea
  - Reading a row from 10% of the blocks is about as slow as reading the entire table (sequential reads!)
  - Option: Reservoir sampling: Explicitly store and maintain a sample
  - Sampling is a build-in database operator; impossible to emulate efficiently