CS222/CS122C: Principles of Data Management

UCI, Fall 2019

Notes #12

Set operations, Aggregation, Cost Estimation

Instructor: Chen Li

Relational Set Operations

- Intersection and cross-product special cases of join.
- Union (**distinct**) and Except similar; we'll do union.
- Sorting-based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
  - *Alternative*: Merge runs from Pass 0^+ for *both* relations (!).
- Hash-based approach to union (from Grace):
  - Partition both R and S using hash function *h1*.
  - For each S-partition, build in-memory hash table (using *h2*), then scan corresponding R-partition, adding truly new S tuples to hash table while discarding duplicates.

Sets versus Bags

- UNION, INTERSECT, and DIFFERENCE (EXCEPT or MINUS) use the set semantics.
- UNION ALL just combine two relations without considering any correlation.

Sets versus Bags

Example:

- R = {1, 1, 1, 2, 2, 2, 2, 3}
- S = {1, 2, 2}
- R UNION S = {1, 2, 3} ← eliminate duplicate
- R UNION ALL S = {1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3}
- R INTERSECT S = {1, 2}
- R EXCEPT S = {3}

Notice that some DBs such as MySQL don't support some of these operations.

Aggregate Operations (AVG, MIN, ...)

- Without grouping:
  - In general, requires scanning the full relation.
  - Given an index whose search key includes *all* attributes in the SELECT or WHERE clauses, can do an index-only scan.
- With *grouping*:
  - *Sort* on the *group-by attributes*, then scan sorted result and compute the aggregate for each group. (Can improve on this by combining sorting and aggregate computations.)
  - Or, similar approach via *hashing* on the group-by attributes.
  - Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do an index-only scan; if group-by attributes form prefix of search key, can retrieve data entries (tuples) in the group-by order.

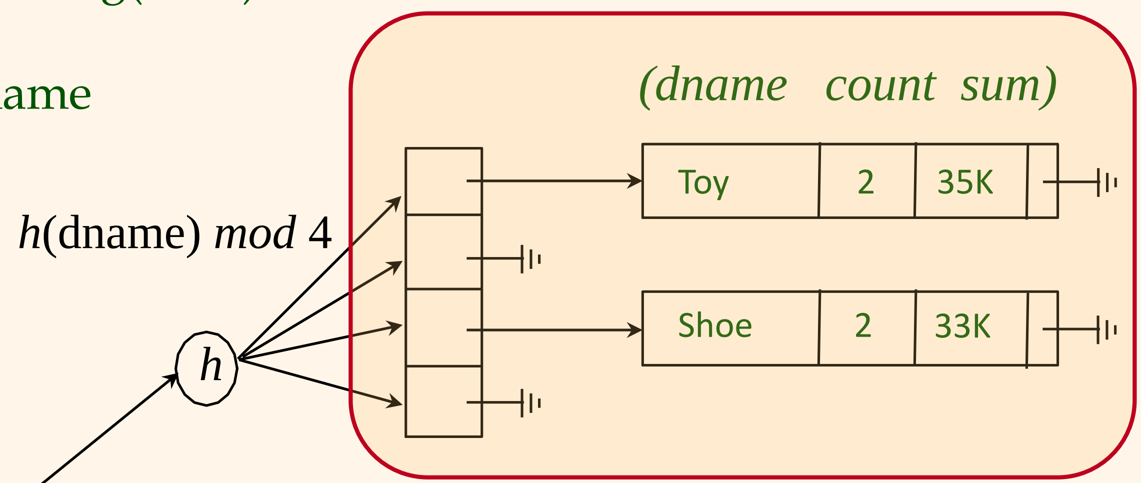
Aggregation State Handling

- State: init( ), next(value), finalize( ) → agg. value
- Consider the following query

```
SELECT   E.dname, avg(E.sal)
FROM   Emp E
GROUP BY  E.dname
```

Emp

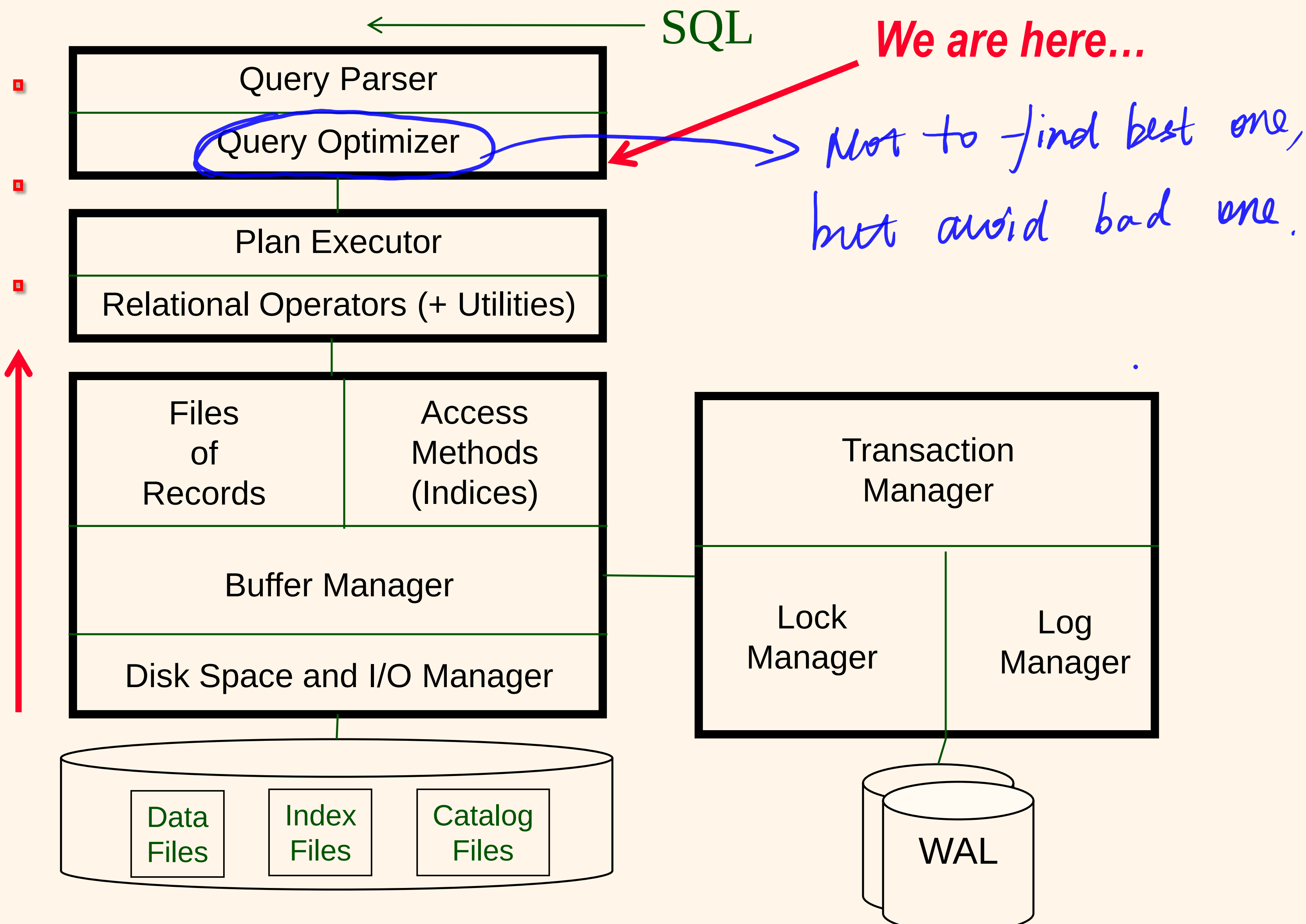
ename	sal	dname
Joe	10K	Toy
Sue	20K	Shoe
Mike	13K	Shoe
Chris	25K	Toy
Zoe	50K	Book
…	…	…



<u>Aggregate state (per unfinished group):</u>

- Count: # values
- Min: min value
- Max: max value
- Sum: sum of values
- Avg: count, sum of values

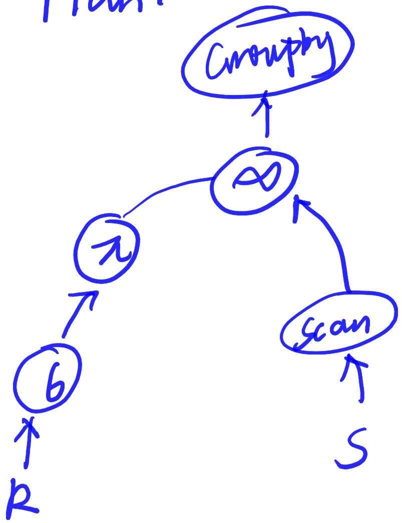
DBMS Structure (from Lecture 1)



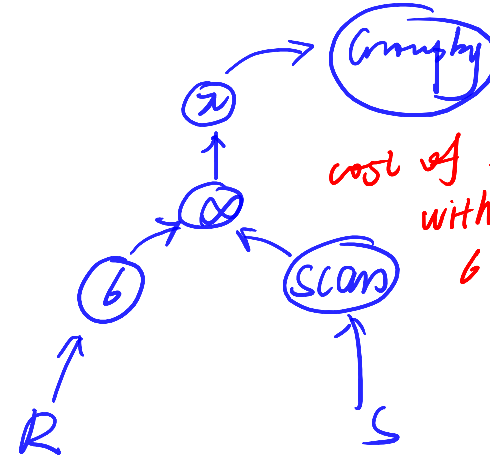
Cost Estimation

- For each plan considered, need to estimate its cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on their input cardinalities.
    - We've already discussed how to estimate the cost of one operation (sequential scan, index scan, join, etc.)
  - Must also estimate *size of result* for each operation in the query plan tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of query predicates.

Plan 1



Plan 2



cost of ⋈ is related with output size of σ and scan

Costs { Disk IOs; Memory requirements; CPU cycles; Network costs

"Cost function"

Q1: cost function

Q2: Estimate cost of query.

Q3: Estimate cost of operator.

(the cost of parent node is related to the cardinality of output of child node)

hard to estimate accurately.

Estimate the cardinality of an Operator.

1) G =

2) Range

3) Boolean

4) Sort → doesn't change cardinality

Estimating cost of a selection

- In our earlier lectures, we already discussed how to estimate the cost of a selection predicate (scan, or using B+ tree)
- We also analyzed the complexity of those operators (e.g., sort, different join methods)
- Next we will mainly focus on size estimation of intermediate results

Intermediate Result Size Estimation

- Optimizers use statistics in catalog to estimate the cardinalities of operators' inputs and outputs
  - Simplifying assumptions: uniform distribution of attribute values, independence of attributes, etc.
- For each relation R:
  - Cardinality of R (|R|), avg R-tuple width, and # of pages in R (||R||) – pick any 2 to know all 3 (given the page size)
- For each (indexed) attribute of R:
  - Number of distinct values $|\pi_A(R)|$
  - Range of values (i.e., low and high values)
  - Number of index leaf pages
  - Number of index levels (if B+ tree)

Simple Selection Queries (σ_p)

- Equality predicate (p is "A = val")
  - $|Q| \approx |R| / |\pi_A(R)|$ number of unique value → should be in catalog file.
    - *Translation*: R's cardinality divided by the number of distinct A values
    - Assumes all values equally likely in R.A (uniform distribution)
    - *Ex:* SELECT * FROM Emp WHERE age = 23;
  - RF (a.k.a. selectivity) is therefore $1 / |\pi_A(R)|$
- Range predicate (p is "$val_1 \le A \le val_2$")
  - $|Q| \approx |R| * ((val2 - val1) / ((high(R.A) - low(R.A))$
    - *Translation*: Selected range size divided by full range size
    - Again assumes uniform value distribution for R.A
    - (Simply replace val_i with high/low bound for a one-sided range predicate)
    - *Ex:* SELECT* FROM Emp WHERE age ≤ 21;
  - RF is $((val_2 - val_1) / ((high(R.A) - low(R.A))$

Boolean Selection Predicates

- Conjunctive predicate (p is "p1 and p2")
  - $RF_p \approx RF_{p1} * RF_{p2}$
    - *Ex 1:* SELECT * FROM Emp WHERE age = 21 AND gender = 'm'
    - Assumes independence of the two predicates p1 and p2 (uncorrelated)
    - *Ex 2:* SELECT * FROM Student WHERE major = 'EE' AND gender = 'm'
- Negative predicate (p is "not p_1")
  - $RF_p \approx 1 - RF_{p1}$
    - *Translation*: All tuples minus the fraction that satisfy p_1
    - *Ex:* SELECT * FROM Emp WHERE age ≠ 21
- Disjunctive predicate (p is "p1 or p2")
  - $RF_p \approx RF_{p1} + RF_{p2} - (RF_{p1} * RF_{p2})$ ← **Q:** Why?
    - *Translation*: All (unique) tuples satisfying either predicate
    - *Ex:* SELECT * FROM Student WHERE major = 'EE' OR gender = 'm'

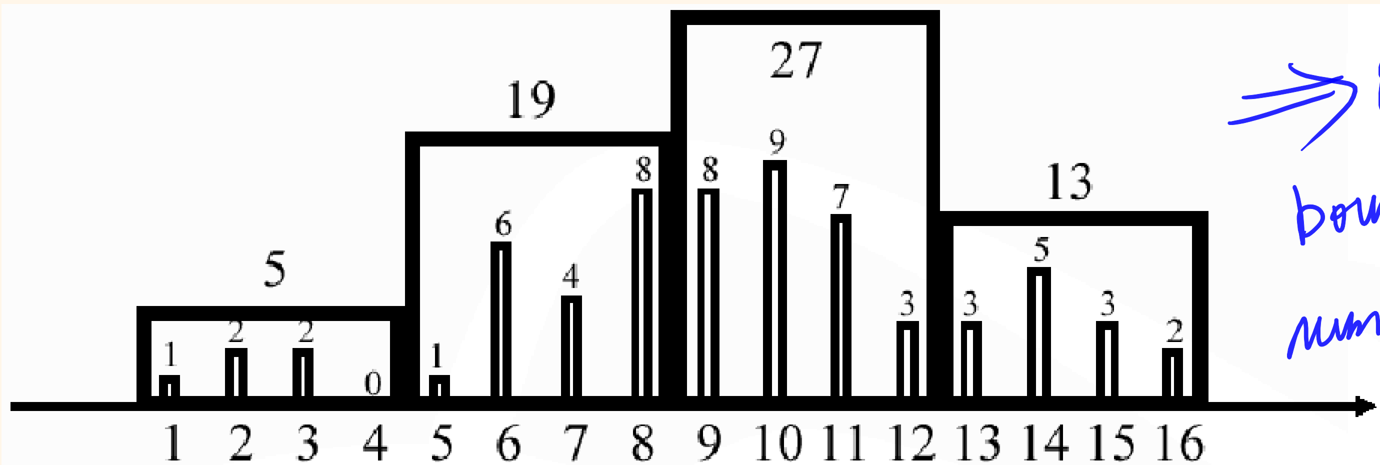
Two-way Equijoin Predicates Join.

- Query Q: R join S on R.A = S.B
- Assume *join value set containment* (FK/PK case)
  - $\pi_A(R)$ is a subset of $\pi_B(S)$ or vice versa
  - Translation: "Smaller" relation's join value set is subset of "larger" relation's join value set (where "size" is based on # unique values)
    - *Ex:* SELECT * FROM Student S, Dept D WHERE S.major = D.deptname
  - $|Q| \approx (|R| * |S|) / \max(|\pi_A(R)|, |\pi_B(S)|)$
    - *Ex:* 100 D.deptname values but 85 S.major values used by 10,000 students ← **Q: Why?** (Why not 1/85?)
    - Estimated size of result is ***1/100*** of the cartesian product of Student and Dept
    - Again making a uniformity assumption (i.e., about students' majors)
  - I.e., RF is $1 / \max(|\pi_A(R)|, |\pi_B(S)|)$
- Repeat same principle to deal with *N*-way joins

Improved Estimation: Histograms

- We have been making simplistic assumptions
  - *Specifically:* uniform distribution of values
  - This is definitely violated (all the time ☺) in reality
  - Violations can lead to huge estimation errors
  - Huge estimation errors can lead to Very Bad Choices
- By the way:
  - In the absence of info, System R assumed 1/3 and 1/10 for range queries and exact match queries, respectively
  - (Q: What might lead to the absence of info?)
- How can we do better in the OTHER direction:
  - Keep track of the most and/or least frequent values
  - Use histograms to better approximate value distributions

Equi-width Histograms



⇒ only store boundary and number of record inside bucket

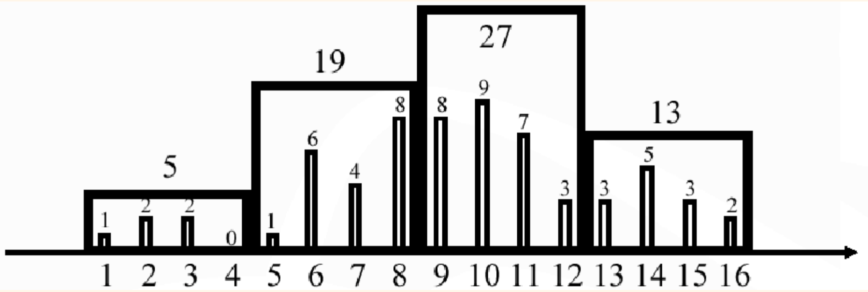
- Divide the domain into *B* buckets of equal width
  - E.g., partition Kid.age values into buckets
- Store the bucket boundaries and the sum of frequencies of the values with each bucket

Histogram Construction

- Initial Construction:
  - Make one full one pass over R to construct an accurate equi-width histogram
    - Keep a running count for each bucket
  - If scanning is not acceptable, use sampling
    - Construct a histogram on R_{sample}, and scale the frequencies by $|R|/|R_{sample}|$
- Maintenance Options:
  - Incremental: for each update or R, increment or decrement the corresponding bucket frequencies (*Q:* Cost?)
  - Periodically recompute: distribution changes slowly!

Using Equi-width Histograms

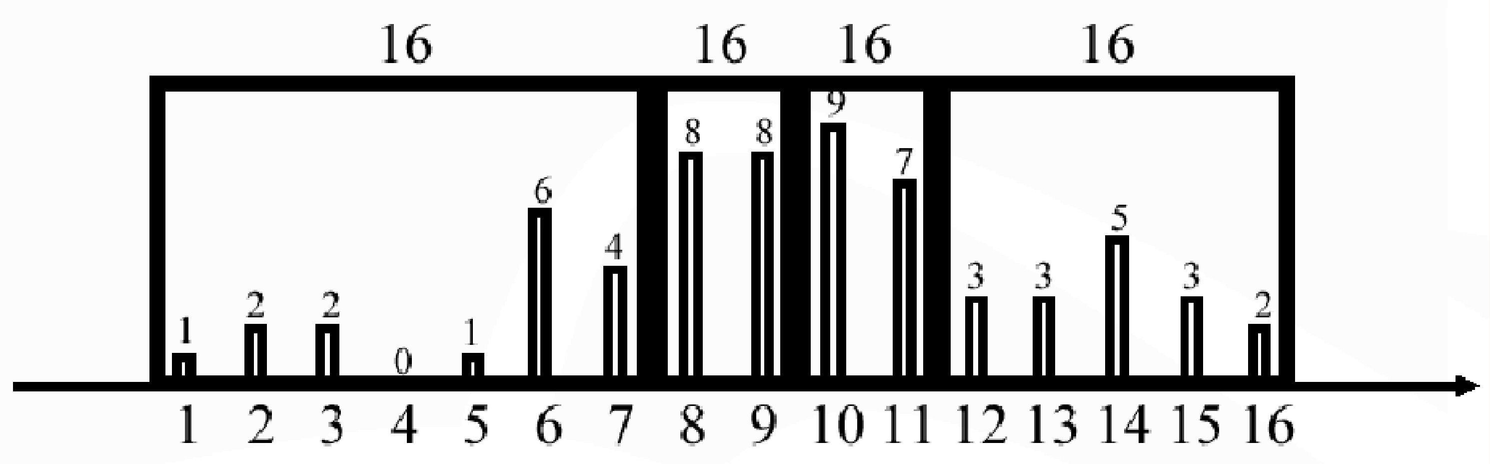
- Q: $\sigma_{A=5}(R)$
  - 5 is in bucket [5,8] (with 19 tuples)
  - Assume uniform distribution within the bucket
  - Thus $|Q| \approx 19/4 \approx 5$.
  - Actual value is 1



- Q: $\sigma_{A>=7\ \&\ A<=16}(R)$
  - [7,16] covers [9,12] (27 tuples) and [13,16] (13 tuples)
  - [7,16] partially covers [5,8] (19 tuples)
  - Thus $|Q| \approx 19/2 + 27 + 13 \approx 50$
  - Actual value = 52.

Equi-height Histogram

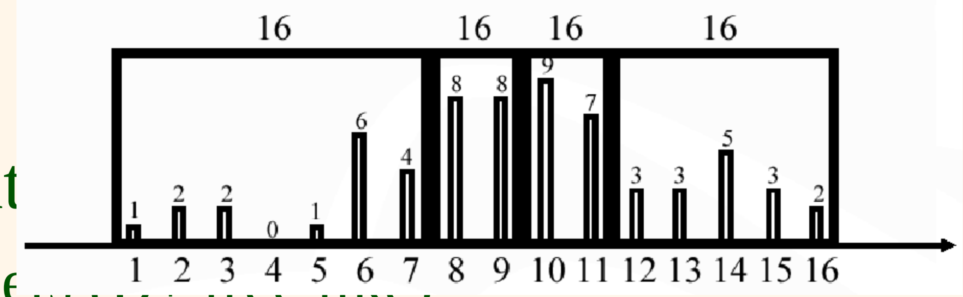
adaptive to skew



- Divide the domain into B buckets with (roughly) the same number of tuples in each bucket
- Store this number and the bucket boundaries
- Intuition: high frequencies are more important than low frequencies

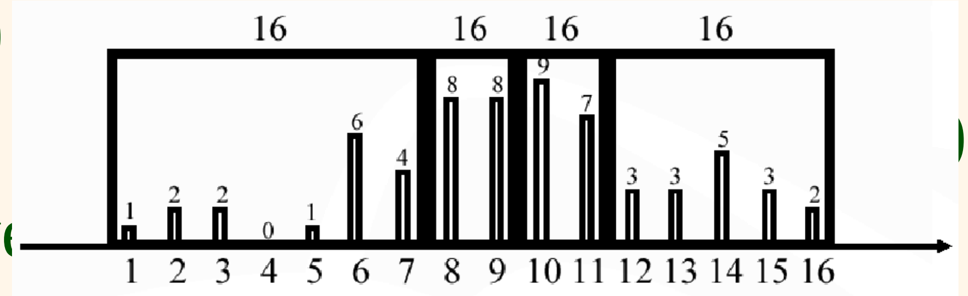
Histogram Construction

- Construction:
  - Sort all R.A values, and then take equally spaced slices
    - *Ex:* 1 2 2 3 3 5 6 6 6 6 6 6 7 7 7 7 8 8 8 8 8 …
  - Sampling also applicable here
- Maintenance:
  - Incremental maint
    - Split/merge bucke
  - Periodic recomputation



Using an equi-height histogram

- Q: $\sigma_{A=5}(R)$
  - 5 is in bucket [1,7] (with 16 tuples)
  - Assume uniform distribution within the bucket
  - Thus $|Q| \approx 16/7 \approx 2$. (actual value = 1)
- Q: $\sigma_{A>=7 \,\&\, A <= 16}(R)$
  - [7,16] covers [8,9],



  - [7,16] partially cove
  - Thus $|Q| \approx 16/7 + 16 + 16 + 16 \approx 50$
  - Actually $|Q| = 52$.

Can Combine Approaches

- If values are badly skewed
  - Keep high/low frequency value information in addition to histograms
  - Could even apply the idea recursively: keep this sort of information for each bucket
    - "Divide" by converting values to some # ranges
    - "Conquer" by keeping some statistics for each range
- Some "statistical glitches" to be aware of
  - Parameterized queries
  - Runtime situations

Histogram $\rightarrow$ distribution