# Query Optimization

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Optimization Algorithms
- Statistics Estimation

# Cost estimation

- Computing operator costs requires information like:
  - Primary key ?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks ?
  - RAID ?? Which one ?
    - Read/write costs are quite different
  - How many tuples match a predicate like “age > 40” ?
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
    - E.g. (R JOIN S) is input to another join operation – need to know if it fits in memory
  - And so on…

# Cost estimation

- Some info is static and maintained in the metadata
  - Primary key ?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks ?
  - RAID ?? Which one ?
    - Read/write costs are quite different
- Typically kept in some tables in the database
  - "all_tab_columns" in Oracle
  - Postgresql: `analyze` cmd updates `pg_statistic` and `pg_stats`
- Most systems have commands for updating them

# Cost estimation

- Others need to be estimated:
  - How many tuples match a predicate like "age > 40" ?
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
- The problem variously called:
  - "intermediate result size estimation"
  - "selectivity estimation"
- Very important to estimate reasonably well
  - e.g. consider "SELECT * FROM R WHERE zipcode = 20742"
  - We estimate that there are 10 matches, and choose to use a secondary index (remember: random I/Os)
  - If turns out there are 10000 matches
    - using a secondary index very bad idea…
  - Optimizer often choose block-nested-loop joins if one relation very small

  … underestimation can be very bad

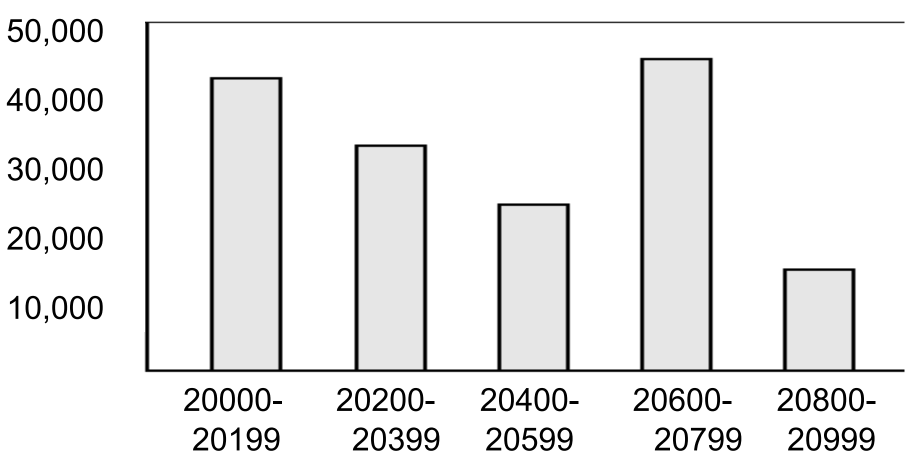
# Selectivity Estimation

- Basic idea:
  - Maintain some information about the tables
    - More information → more accurate estimation
    - More information → higher storage cost, higher update cost
  - Make uniformity and randomness assumptions to fill in the gaps

- Example:
  - For a relation "people", we keep:
    - Total number of tuples = 100,000
    - Distinct "zipcode" values that appear in it = 100
  - Given a query: "zipcode = 20742"
    - We estimated the number of matching tuples as: 100,000/100 = 1000
  - What if I wanted more accurate information ?
    - Keep histograms…

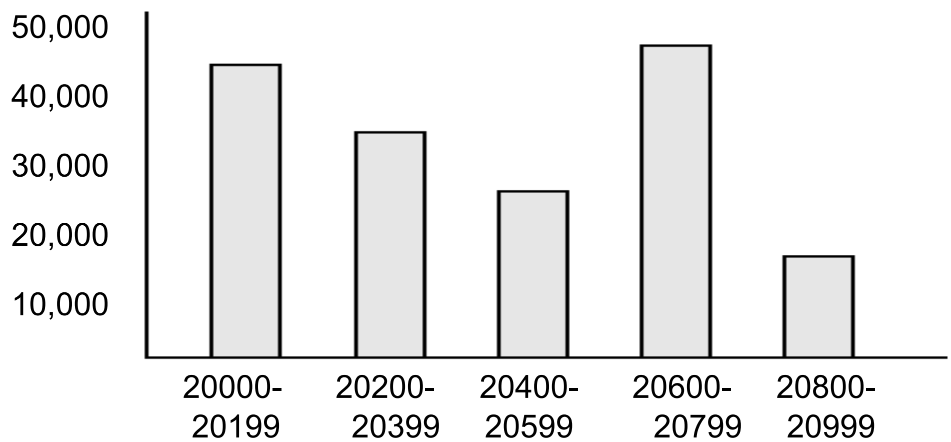
# Histograms

- A condensed, approximate version of the "frequency distribution"
  - Divide the range of the attribute value in "buckets"
  - For each bucket, keep the total count
  - Assume uniformity within a bucket



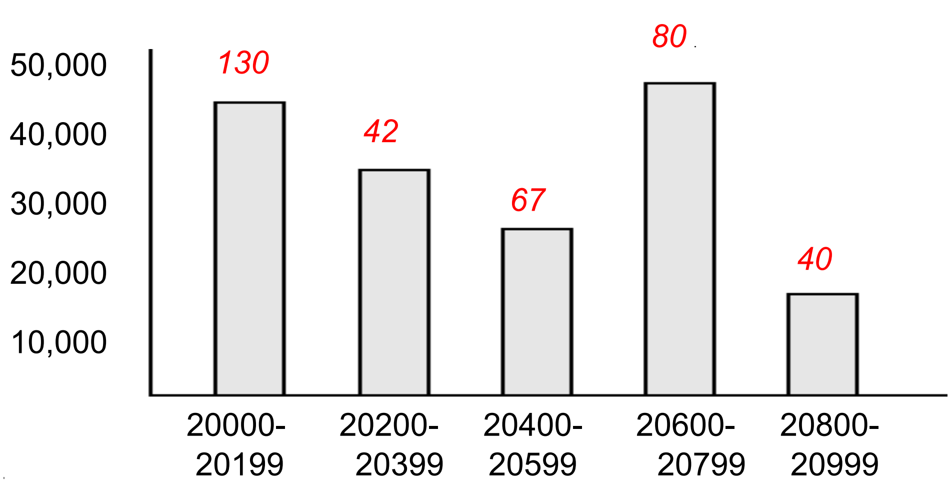
# Histograms

- Given a query: zipcode = " 20742"
  - Find the bucket (Number 3)
  - Say the associated count = 45000
  - Assume uniform distribution within the bucket: 45,000/200 = 225




# Histograms

- What if the ranges are typically not full ?
  - ie., only a few of the zipcodes are actually in use ?
- With each bucket, also keep the number of distinct values used for zipcodes
- Now the estimate would be: 45,000/80 = 562.50
- More Information → Better estimation



# Exam #2

- Functional dependences (extraneous attributes, covers)
- Storage manager
- RAID
- File organization (heap, sorted, hash)
- Indexes (primary / secondary, dense sparse, hash)
  - B+-trees: height, cost of access, including xtra leaves
  - insertions, deletions
- Query execution (including costs)
  - selections
  - joins (block nested, hash, merge, index nested..)
  - sorts (in-memory, external)
- Query estimation
  - histograms
  - uniformity
  - using attribute stats
- Query optimization
  - execution trees
  - materialization/pipelining

# Query Optimization

- Introduction
- Example of a Simple Type of Query
- Transformation of Relational Expressions
- Optimization Algorithms
- Statistics Estimation

# Histograms

- Very widely used in practice
  - One-dimensional histograms kept on almost all columns of interest
    - ie., the columns that are commonly referenced in queries
  - Sometimes: multi-dimensional histograms also make sense
    - Less commonly used as of now
- Two common types of histograms:
  - Equi-depth
    - The attribute value range partitioned such that each bucket contains about the same number of values
  - Equi-width
    - The attribute value range partitioned in equal-sized buckets
  - More dimensions, etc …

# Estimating Result Sizes…

- Estimating sizes of the results of various operations
- Guiding principle:
  - Use all the information available
  - Make uniformity and randomness assumptions otherwise
  - Many formulas, but not very complicated…
    - In most cases, the first thing you think of!

# Basic statistics

- Basic information stored for all relations
  - $n_r$: number of tuples in a relation *r.*
  - $b_r$: number of blocks containing tuples of *r.*
  - $f_r$: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
  - *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
  - *MAX(A, r):* maximum value of *A* that appears in *r*
  - *MIN(A, r)*
  - If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

# Selection Size Estimation

- $\sigma_{A=X}(r)$
  - $n_r / V(A,r)$ : number of records that will satisfy the selection
  - equality condition on a key attribute: *size estimate* = 1

- $\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ is symmetric)
  - Let *c* denote the estimated number of tuples satisfying the condition.
  - If *min(A,r)* and *max(A,r)* are available in catalog
    - $c = 0$ if $v < min(A,r)$
    - $c = n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$ if *min(A,r)* <= *v* <= *max(A,r)*
    - $c = n_r$ otherwise
  - If histograms available, can refine above estimate
  - In absence of *any* information *c* is assumed to be $n_r / 2$.

## Size Estimation of Complex Selections

- selectivity($\theta_i$) = the probability that a particular tuple in *r* satisfies $\theta_i$ .
  - If $s_i$ is the number of satisfying tuples in *r*, then selectivity ($\theta_i$) = $s_i / n_r$.

- conjunction: $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *Assuming independence*, estimate of tuples in the result is:

$$n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$$

- disjunction: $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

- negation: $\sigma_{\neg\theta}(r)$. Estimated number of tuples: $n_r - size(\sigma_\theta(r))$

## Estimating Output Sizes: Joins

- R JOIN S: *R.a = S.a*
  - |R| = 10,000; |S| = 5000

- CASE 1: *a* is key for S
  - *Worst case: each tuple of* R *joins with exactly one tuple of* S
  - So: |*R* JOIN *S*| = |*R*| = 10,000

- CASE 2: *a* is key for R
  - Each *S* tuple can match w/ only a single *R* tuple.
  - So: |*R* JOIN *S*| = |*S*| = 5,000

Equi-joins simplify things.

# Estimating Output Sizes: Joins

- R JOIN S: R.a = S.a
  - |R| = 10,000; |S| = 5000

- CASE 3: *a* is not a key for either
  - Reason with the distributions on *a*
  - Say: the domain of *a: V(a, R) = V(a, S)* = 100 (distinct values *a* can take)
  - THEN, *assuming uniformity*
    - For each value of *a*
      - We have 10,000/100 = 100 tuples of R with that value of *a*
      - We have 5000/100 = 50 tuples of S with that value of *a*
      - All of these will join with each other, and produce 100 *50 = 5000 for each *a*
    - So total number of results in the join:
      - 5000 * 100 (distinct values) = 500,000
  - We can improve the accuracy if we know the distributions on *a* better
    - Say using a histogram

# Estimating Output Sizes: Other Ops

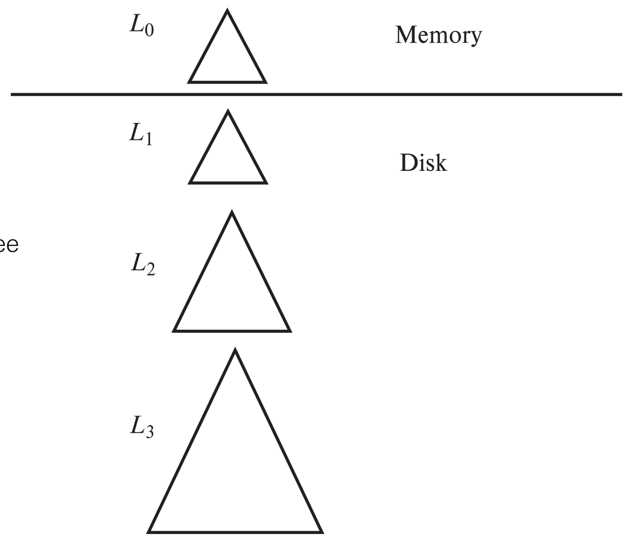
- Projection: $\prod_A(R)$
  - If no duplicate elimination, THEN $|\prod_A(R)| = |R|$
  - If *distinct* used (duplicate elimination performed): $|\prod_A(R)| = V(A, R)$

- Set operations: (heuristic upper bounds)
  - Union ALL: $|R \cup S| = |R| + |S|$
  - Intersect ALL: $|R \cap S| = \min\{|R|, |S|\}$
  - Except ALL: $|R - S| = |R|$
  - Union, Intersection, Except (with duplicate elimination)
    - Somewhat more complex reasoning based on the frequency distributions etc…

- And so on …

# Log Structured Merge (LSM) Tree *B+Tree Alternative*

- For write-heavy workloads
  - also SSDs
- Looking at just inserts/queries
  - Records inserted first into in-memory tree (L0 tree)
  - When in-memory tree is full, records moved to disk (L1 tree)
  - B+-tree constructed using bottom-up build by merging existing $L_1$ tree with records from $L_0$ tree
- When $L_1$ tree exceeds some threshold, merge into $L_2$ tree
  - And so on for more levels
  - Size threshold for $L_{i+1}$ tree is *k times* size threshold for $L_i$ tree
- A query is applied to all trees $L_0$ through $L_n$
  - but a match in $L_i$ means $L_j$ s.t. j>i ignored




# Log Structured Merge (LSM) Tree *B+Tree Alternative*

- Benefits of LSM approach
  - Inserts are done using only sequential I/O operations
  - Leaves are full, avoiding space wastage
  - Reduced number of I/O operations per record inserted as compared to normal B+-tree (each tree written in single write)
- Drawback of LSM approach
  - Queries have to search multiple trees
  - Entire content of each level copied multiple times
- Many variants, but especially:
  - Each query requires lookup on each tree.
  - But keys in a disk-only trees can be summarized w/ a *bloom filter*

# HW 5:4.3

*Please make it easier on the graders and use the algorithm in Figure 8.9*

- $F$ logically implies $F_c$
- $F_c$ logically implies $F$
- no extraneous attributes
- each left side of $F_c$ is unique

*Please make it easier on the graders and use the algorithm in Figure 8.9*

Candidate keys: *A*, *C*

$A \to CD, C \to ABE, BC \to A, A\underline{E} \to B$

$\underline{A} \to CD, C \to ABE, BC \to A, \underline{A} \to B$ (E extra)

$A \to BCD, C \to ABE, \underline{BC} \to A$ (union)

$A \to BCD, \underline{C} \to ABE, \underline{C} \to A$ (B extra)

$A \to \underline{B}CD, C \to ABE$ (union)

$A \to CD, C \to ABE$ (B extra)

$A^+$=ABCDE in F'

$C^+$=ABCDE in F'

C is candidate key in F

σ is *extraneous* in α iff:
F → F', *or*
**(α – σ)⁺ includes β under *F***

σ is *extraneous* in β iff:
F' → F, *or*
**α⁺ includes σ in *F'***

# HW 7: 5

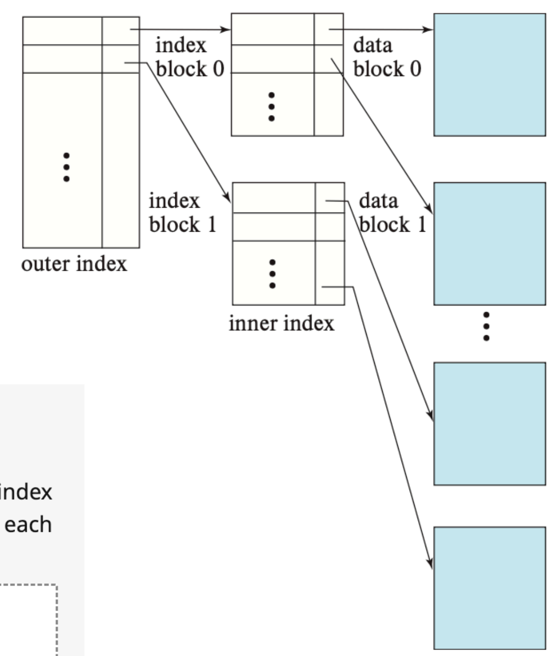



**Figure 14.5** Two-level sparse index.

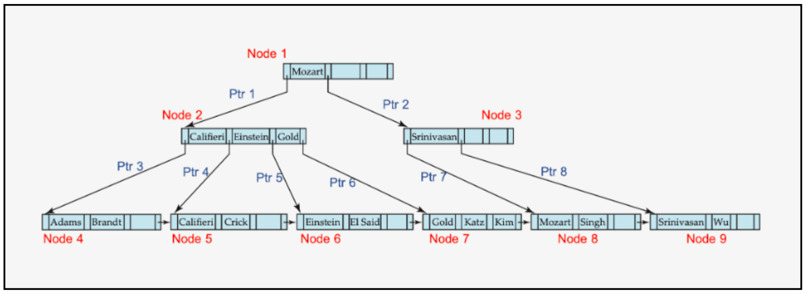
**Q5**
**2 Points**

Calculate the total minimum number of blocks in a multi-level index (Figure 14.5) if there are 2,000,000 blocks of sorted tuples, and each block in the inner or outer index can store 500 pointer entries.

**Explanation**

We would need a ptr to each block of tuples, which means 2,000,000 ptrs. This would require 2,000,000/500 = 4000 inner index pages, plus 8 more outer index pages = 4008.

# HW 7: 8.3



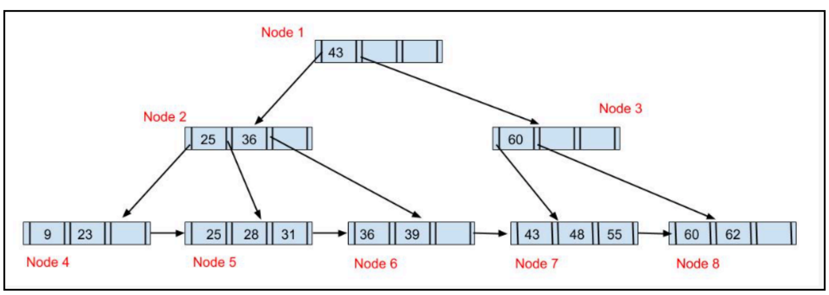

**Q8.3**

**1 Point**

Consider deleting the tuple with the key "Gold". After deleting the key from Node 7, how would Node 2 be affected (per the algorithm 14.21 in the book)?

- Delete "Gold" and the corresponding pointer from Node 2.
- Replace "Gold" with "Katz" in Node 2.
- No change -- leave "Gold" as is in Node 2.

# HW 7: 9.3




**Q9.3**

**1 Point**

Which of the following sequences of inserts will result in addition of a new entry to the root?

- 49, 50, 51
- 30, 20, 21
- 56, 57, 58, 59
- 24, 30, 26, 40