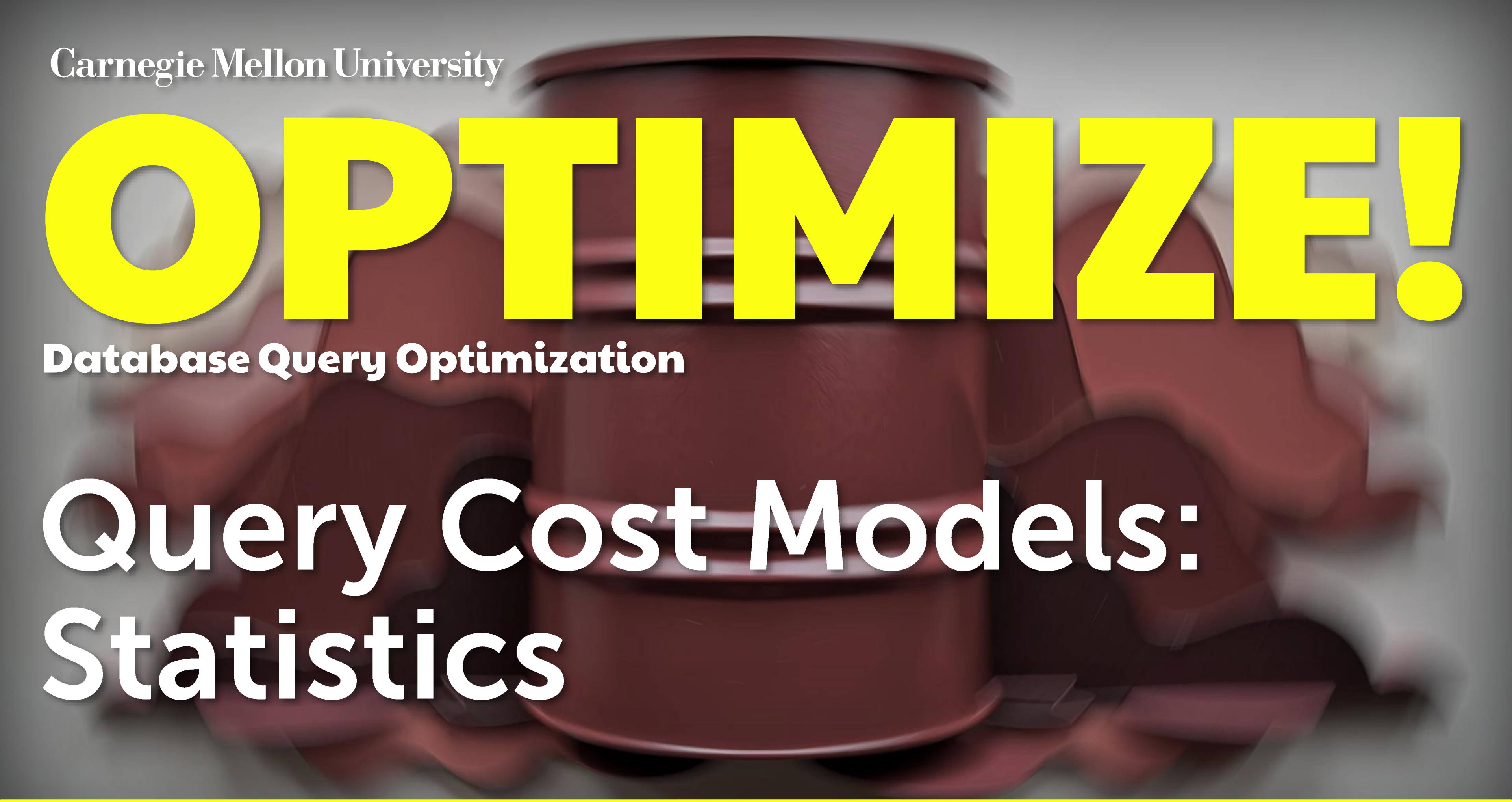
Carnegie Mellon University

OPTIMIZE!

Database Query Optimization

Query Cost Models: Statistics



SPRING 2025 » SPECIAL TOPICS IN DATABASES » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #1 is due Friday Feb 28th.

We will assign **Project #2** topics later today.

→ I am available during Spring Break to discuss your project plans and proposal presentation.

UPCOMING DATABASE TALKS

Pinot (DB Seminar)
→ Monday Feb 24th @ 4:30pm ET
→ Zoom

GoogleSQL Pipes (DB Seminar)
→ Monday Mar 10th @ 4:30pm ET
→ Zoom

Malloy (DB Seminar)
→ Monday Mar 17th @ 4:30pm ET
→ Zoom

LAST CLASS

How to use transformation rules to rewrite dependent joins in correlated subqueries to become inner joins.

This is one of the important advancements in query optimization from the last decade.

COST ESTIMATION

An optimizer uses a **cost model** to estimate the physical execution cost of a query plan given a database state.

→ This is an internal cost that allows the DBMS to compare one plan with another.
→ Estimating the expected runtime (or completion time) of a query is a similarly difficult problem.

The cost of a plan is estimated by combining the costs of individual operators in a plan.

COST MODEL COMPONENTS

Choice #1: Physical Costs

→ CPU cycles, disk I/O, network I/O, buffer pool misses, memory consumption…
→ Specific to physical implementation of an operator.
→ Depends heavily on hardware characteristics and requires calibration to ensure accurate estimates.

Choice #2: Logical Costs

→ Input/output size of intermediate results per operator.
→ Independent of the operator algorithm.
→ Can include width estimations of resulting tuples.

COST MODEL COMPONENTS

Choice #1: Physical Costs

→ CPU cycles, disk I/O, network I/O, buffer pool misses, memory consumption…
→ Specific to physical implementation of an operator.
→ Depends heavily on hardware characteristics and requires calibration to ensure accurate estimates.

Choice #2: Logical Costs

→ Input/output size of intermediate results per operator.
→ Independent of the operator algorithm.
→ Can include width estimations of resulting tuples.

Cardinality Estimation

OBSERVATION

We already saw the use of logical costs to guide top-down optimization searches.
→ Predicted-cost Bounding
→ Promises

Computing an operator's logical cost is not magically faster than computing its physical cost.

TODAY'S AGENDA

Background
Histograms
Sketches
Sampling
Implementations

STATISTICAL SUMMARIES

Auxiliary data structures that the DBMS populates from scanning the database to allow the optimizer to approximate data contents for different scenarios.

Trade-offs to consider:
→ Accuracy
→ Efficiency
→ Memory Consumption
→ Coverage / Applicability
→ Creation + Maintenance Costs

Source: EQOP Book

STATISTICS STORAGE

Most DBMSs store a database's statistics in its internal catalog.

The DBMS will periodically update statistics according to one or more triggering mechanisms:

→ Periodic Background Tasks (e.g., Postgres Autovacuum)
→ Maintenance Schedules (e.g., Oracle)
→ Modification Thresholds
→ Manual Invocation (e.g., **ANALYZE**, **UPDATE STATISTICS**)

COLUMN STATISTICS

Most DBMSs create single-column statistics for each column in a table.

The DBMS can also track statistics for groups of attributes together rather than just treating them all as independent variables.

→ Some systems automatically build multi-column statistics if they are already used in an index together (MSSQL).
→ Otherwise, a human manually specifies target columns.
→ Also called Column Group Statistics (Db2) or Extended Statistics (Oracle).

SUMMARIZATION APPROACHES

Choice #1: Histograms ⬅ ***Most Common***

→ Maintain an occurrence count per value (or range of values) in a column.

Choice #2: Sketches ⬅ ***Increasing Usage***

→ Probabilistic data structure that gives an approximate count for a given value.

Choice #3: Sampling ⬅ ***Rare***

→ DBMS maintains a small subset of each table that it then uses to evaluate expressions to compute selectivity.

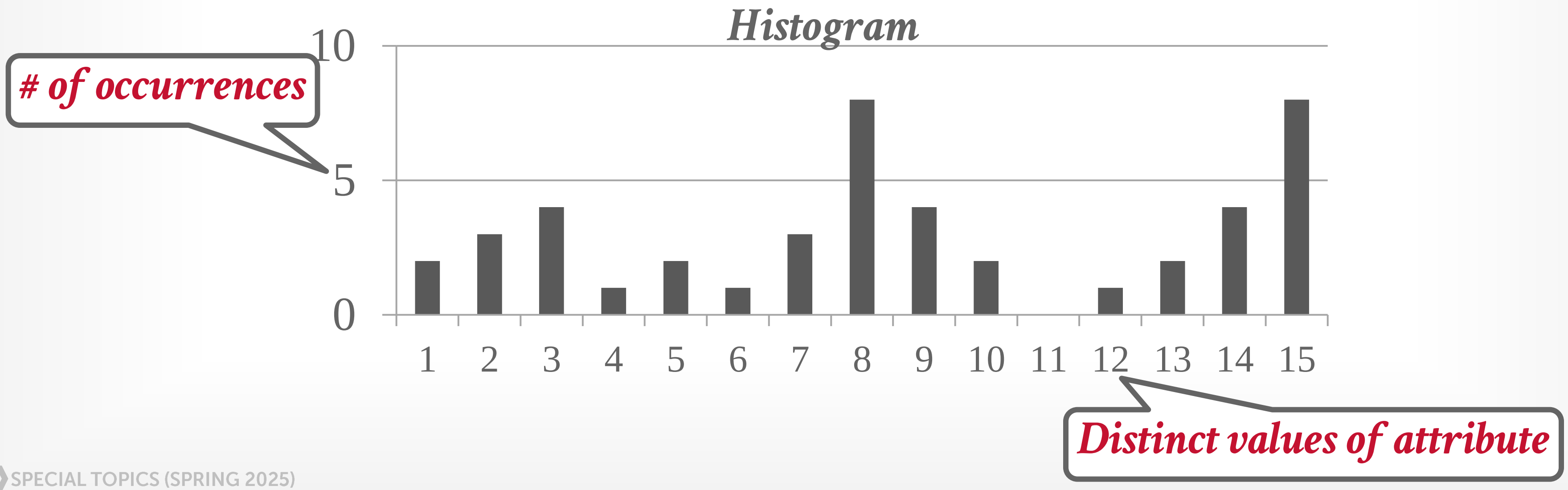
Choice #4: ML Model ⬅ ***Experimental / Very Rare***

→ Train an ML model that learns the selectivity of predicates and correlations between multiple tables.

HISTOGRAMS

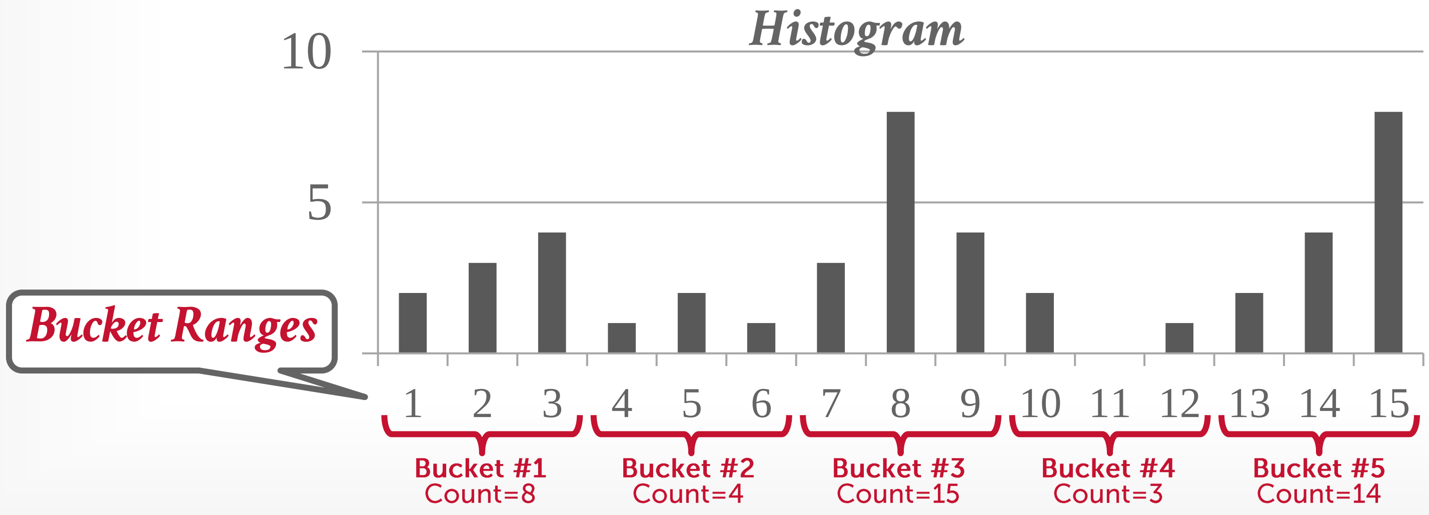
Approximate the distribution of values in a column for cardinality estimation.

→ Maintain an occurrence count per value (or range of values) in a column.



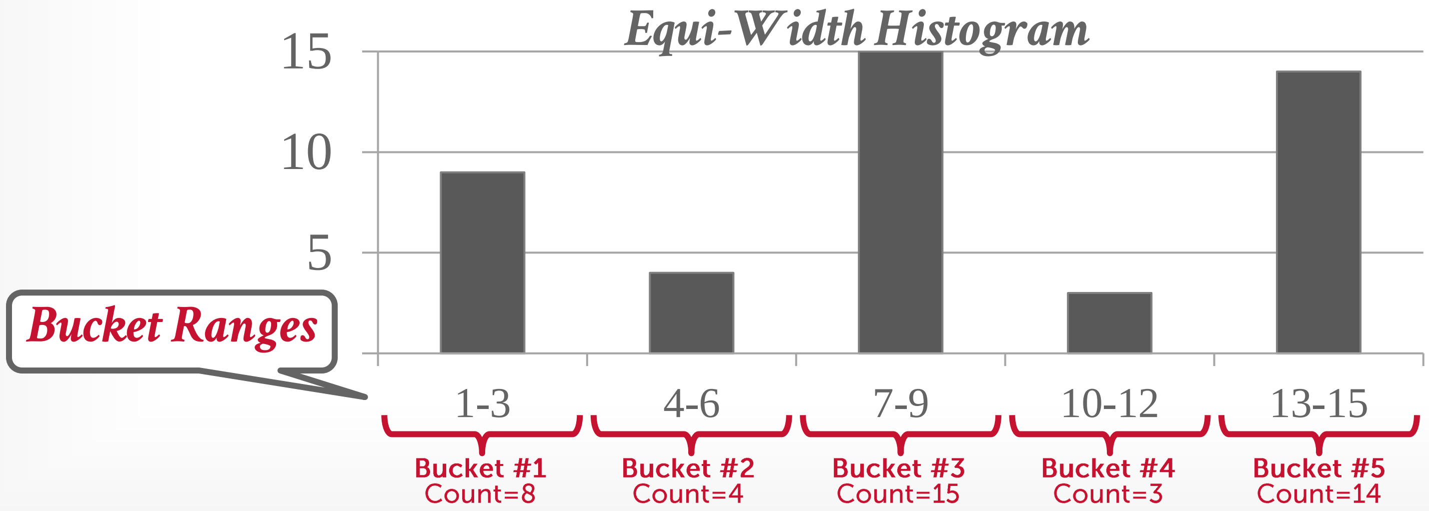
EQUI-WIDTH HISTOGRAM

Maintain counts for a group of values instead of each unique key. All buckets have the same width (i.e., same # of value).



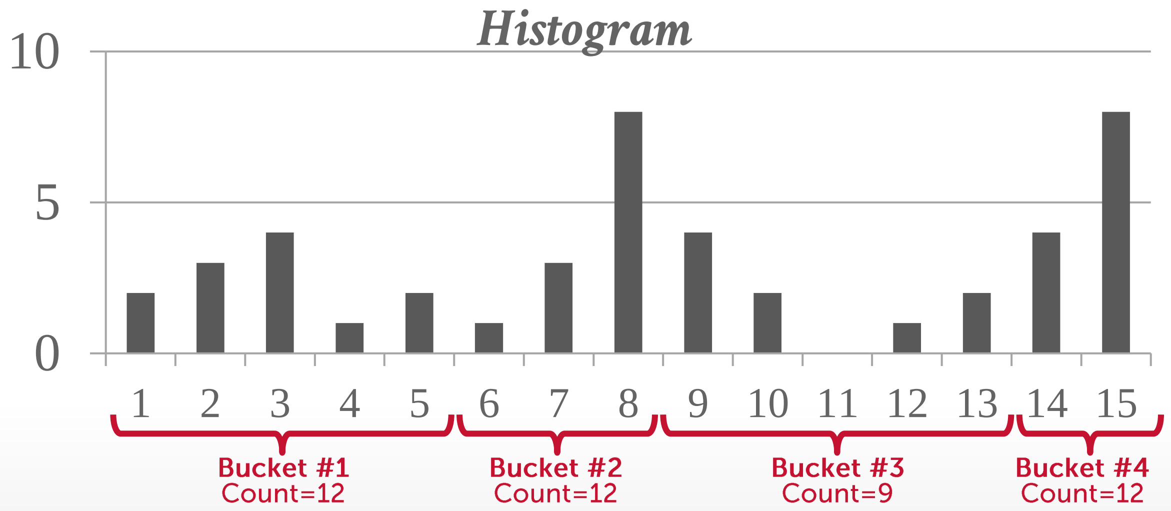
EQUI-WIDTH HISTOGRAM

Maintain counts for a group of values instead of each unique key. All buckets have the same width (i.e., same # of value).



EQUI-DEPTH HISTOGRAMS

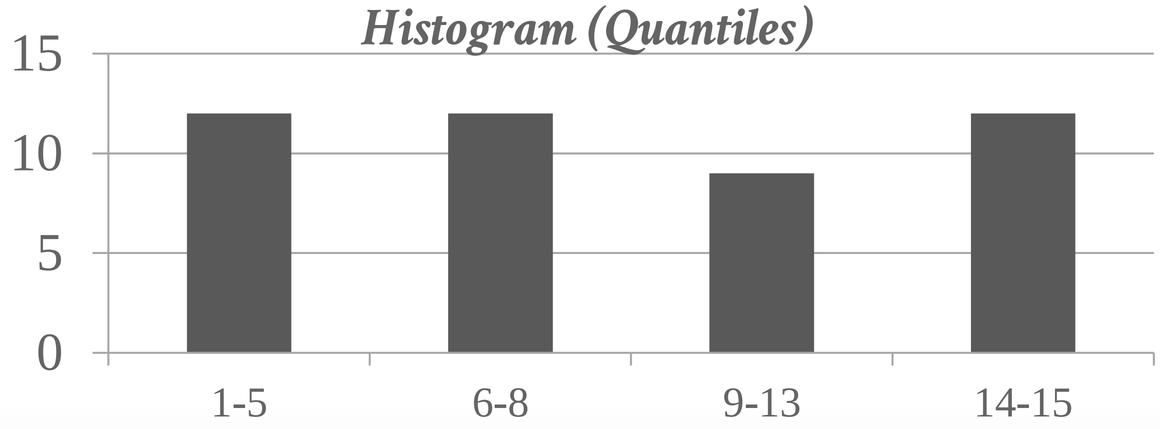
Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.
→ Equi-depth histograms are shown to have better worst-case and average error than equi-width histograms.



EQUI-DEPTH HISTOGRAMS

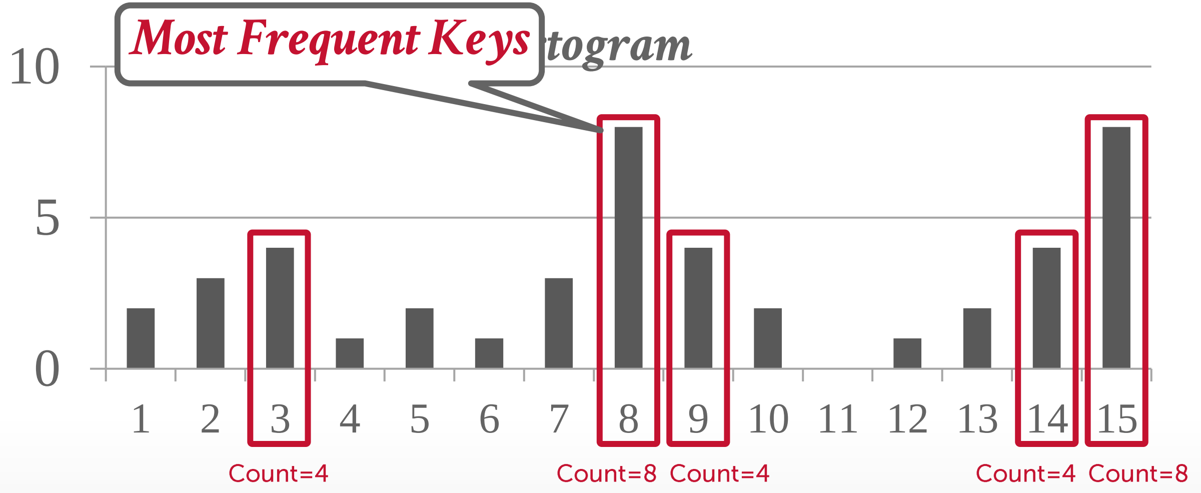
Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.

→ Equi-depth histograms are shown to have better worst-case and average error than equi-width histograms.



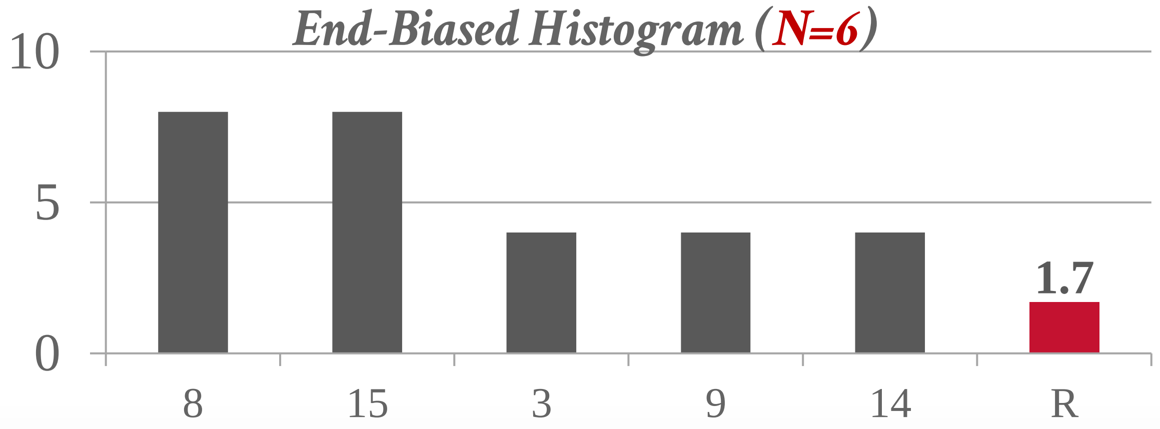
END-BIASED HISTOGRAMS

Use ***N-1*** buckets to store the exact count for the most frequent keys. The last bucket (**R**) stores the average frequency of all remaining values.



END-BIASED HISTOGRAMS

Use ***N-1*** buckets to store the exact count for the most frequent keys. The last bucket (**R**) stores the average frequency of all remaining values.



SKETCHES

Maintaining exact statistics about the database is expensive and slow.

Use probabilistic data structures called **sketches** to generate error-bounded estimates.

→ Frequent Items (Count-min Sketch)
→ Count Distinct (HyperLogLog)
→ Quantiles (t-digest)

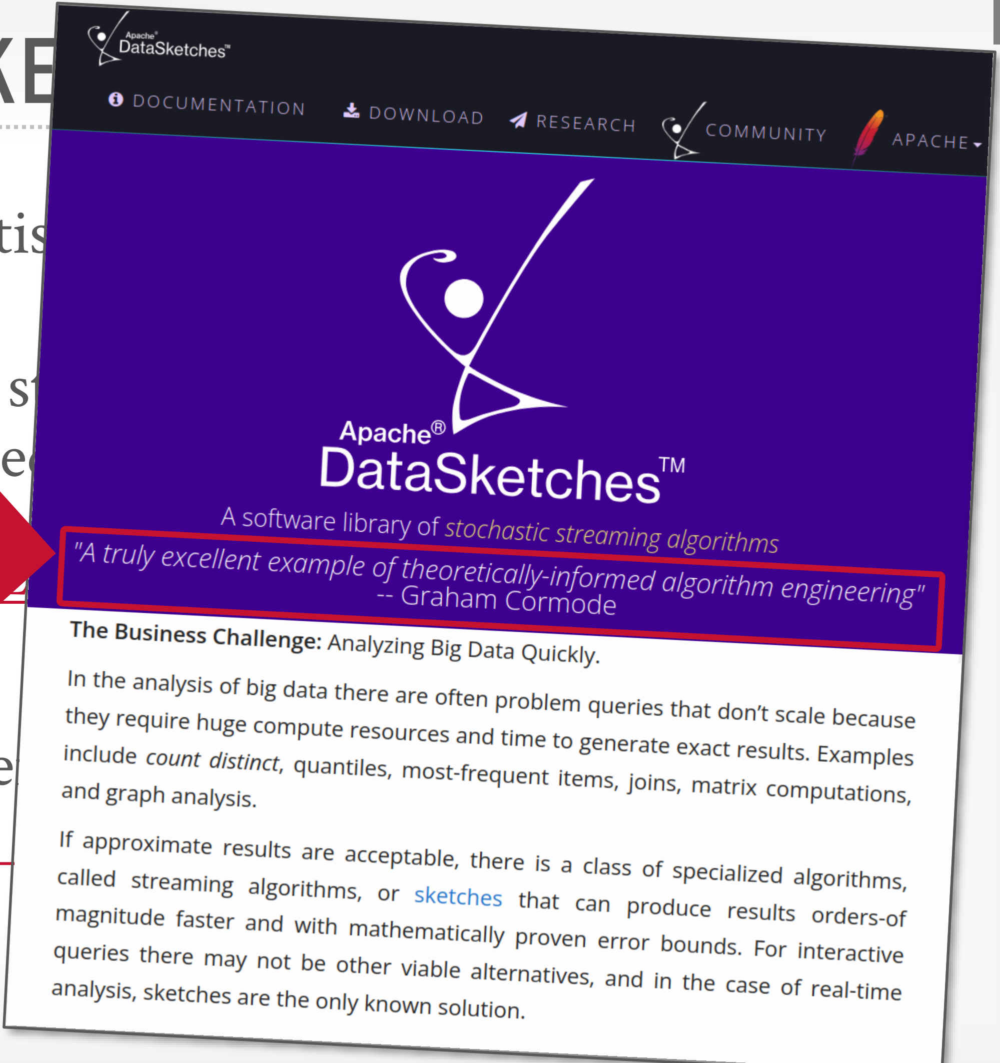
Open-source implementations are available (Apache DataSketches, Google ZetaSketch)

SKE

Maintaining exact statis
expensive and slow.

Use probabilistic data st
generate error-bounde
→ Frequent Items (Co
→ Count Distinct (Hype
→ Quantiles (t-digest)

Open-source impleme
DataSketches, Google



A software library of *stochastic streaming algorithms*

"A truly excellent example of theoretically-informed algorithm engineering"
-- Graham Cormode

The Business Challenge: Analyzing Big Data Quickly.

In the analysis of big data there are often problem queries that don't scale because they require huge compute resources and time to generate exact results. Examples include *count distinct*, quantiles, most-frequent items, joins, matrix computations, and graph analysis.

If approximate results are acceptable, there is a class of specialized algorithms, called streaming algorithms, or sketches that can produce results orders-of magnitude faster and with mathematically proven error bounds. For interactive queries there may not be other viable alternatives, and in the case of real-time analysis, sketches are the only known solution.

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch

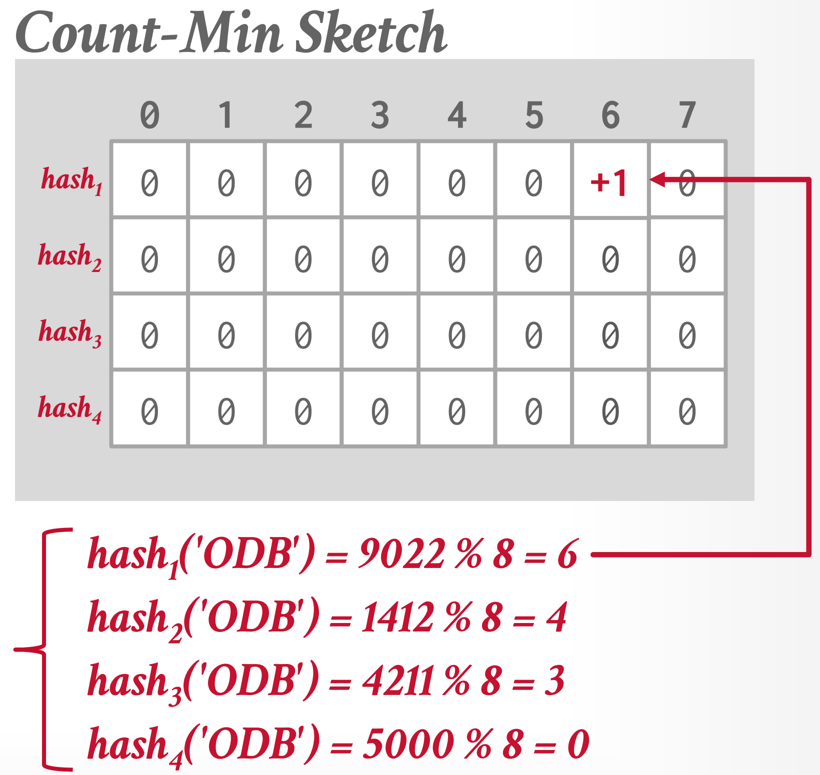
	0	1	2	3	4	5	6	7
$hash_1$	0	0	0	0	0	0	0	0
$hash_2$	0	0	0	0	0	0	0	0
$hash_3$	0	0	0	0	0	0	0	0
$hash_4$	0	0	0	0	0	0	0	0

$$\texttt{INSERT 'ODB'} \begin{cases} hash_1(\text{'ODB'}) = 9022 \,\%\, 8 = 6 \\ hash_2(\text{'ODB'}) = 1412 \,\%\, 8 = 4 \\ hash_3(\text{'ODB'}) = 4211 \,\%\, 8 = 3 \\ hash_4(\text{'ODB'}) = 5000 \,\%\, 8 = 0 \end{cases}$$

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.



	0	1	2	3	4	5	6	7
$hash_1$	0	0	0	0	0	0	+1	0
$hash_2$	0	0	0	0	0	0	0	0
$hash_3$	0	0	0	0	0	0	0	0
$hash_4$	0	0	0	0	0	0	0	0

$hash_1('ODB') = 9022 \% 8 = 6$

$hash_2('ODB') = 1412 \% 8 = 4$

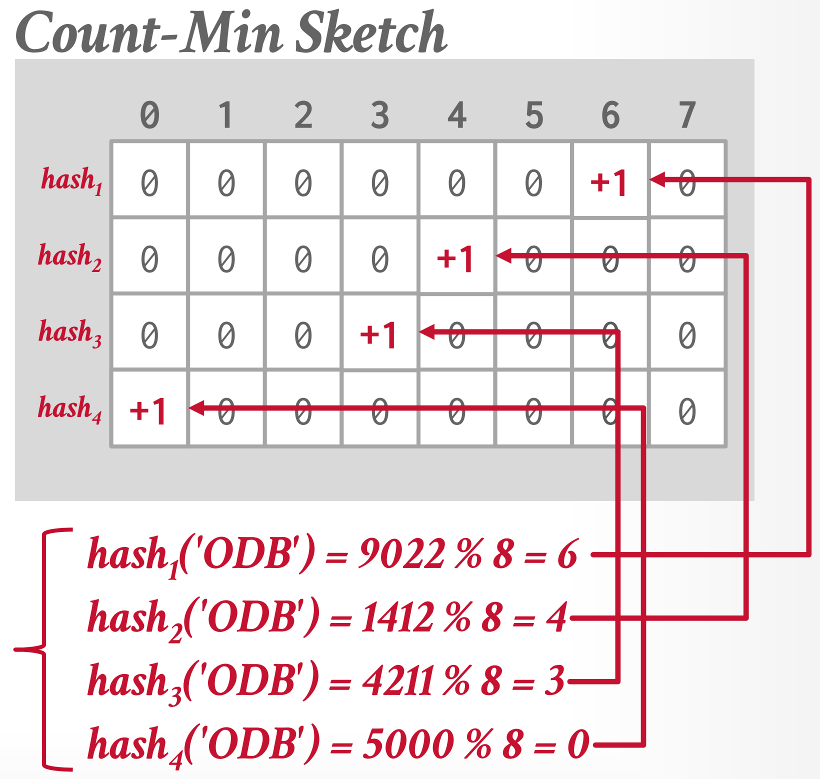
$hash_3('ODB') = 4211 \% 8 = 3$

$hash_4('ODB') = 5000 \% 8 = 0$

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.



INSERT 'ODB'

$hash_1('ODB') = 9022 \% 8 = 6$

$hash_2('ODB') = 1412 \% 8 = 4$

$hash_3('ODB') = 4211 \% 8 = 3$

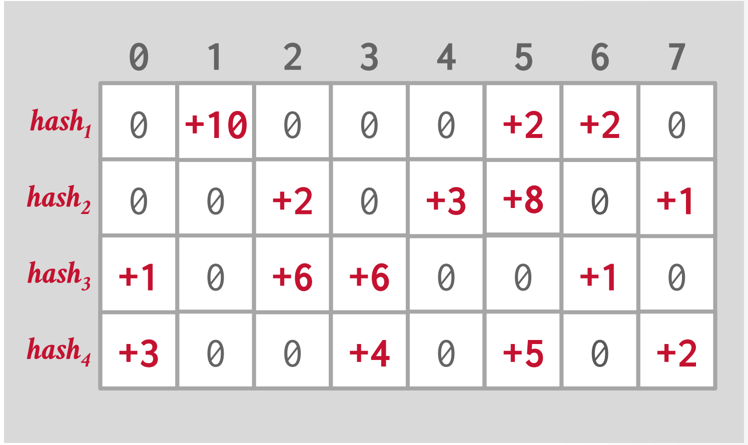
$hash_4('ODB') = 5000 \% 8 = 0$

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch



	0	1	2	3	4	5	6	7
$hash_1$	0	+10	0	0	0	+2	+2	0
$hash_2$	0	0	+2	0	+3	+8	0	+1
$hash_3$	+1	0	+6	+6	0	0	+1	0
$hash_4$	+3	0	0	+4	0	+5	0	+2

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch

	0	1	2	3	4	5	6	7
$hash_1$	0	+10	0	0	0	+2	+2	0
$hash_2$	0	0	+2	0	+3	+8	0	+1
$hash_3$	+1	0	+6	+6	0	0	+1	0
$hash_4$	+3	0	0	+4	0	+5	0	+2

`GET 'ODB'`

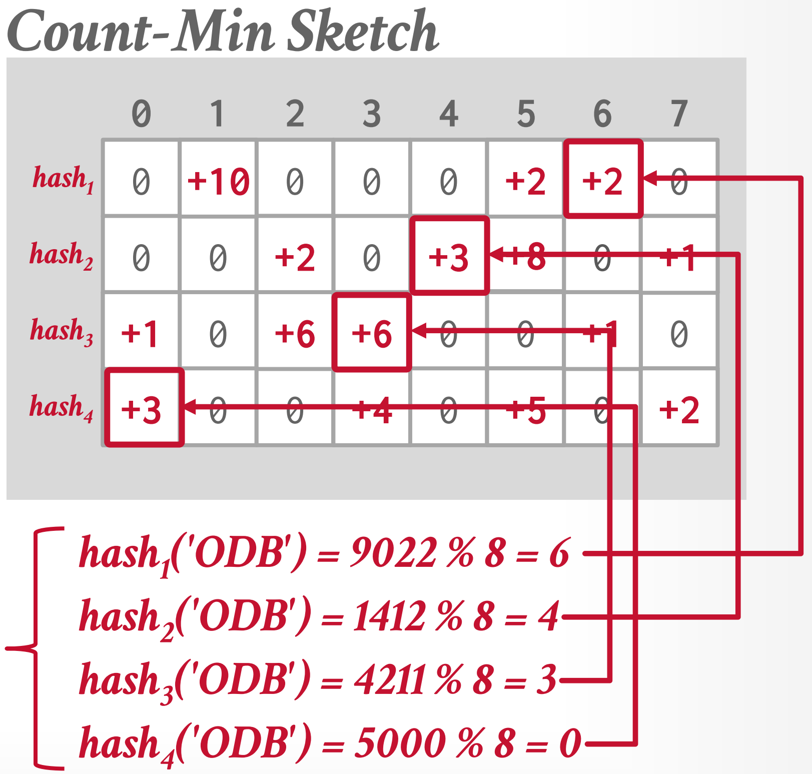
$$\left\{\begin{array}{l} hash_1(\text{'ODB'}) = 9022 \;\%\; 8 = 6 \\ hash_2(\text{'ODB'}) = 1412 \;\%\; 8 = 4 \\ hash_3(\text{'ODB'}) = 4211 \;\%\; 8 = 3 \\ hash_4(\text{'ODB'}) = 5000 \;\%\; 8 = 0 \end{array}\right.$$

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

GET 'ODB'



$$
\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
hash_1 & 0 & +10 & 0 & 0 & 0 & +2 & \boxed{+2} & 0 \\
hash_2 & 0 & 0 & +2 & 0 & \boxed{+3} & +8 & 0 & +1 \\
hash_3 & +1 & 0 & +6 & \boxed{+6} & 0 & 0 & +1 & 0 \\
hash_4 & \boxed{+3} & 0 & 0 & +4 & 0 & +5 & 0 & +2
\end{array}
$$

$hash_1('ODB') = 9022 \% 8 = 6$

$hash_2('ODB') = 1412 \% 8 = 4$

$hash_3('ODB') = 4211 \% 8 = 3$

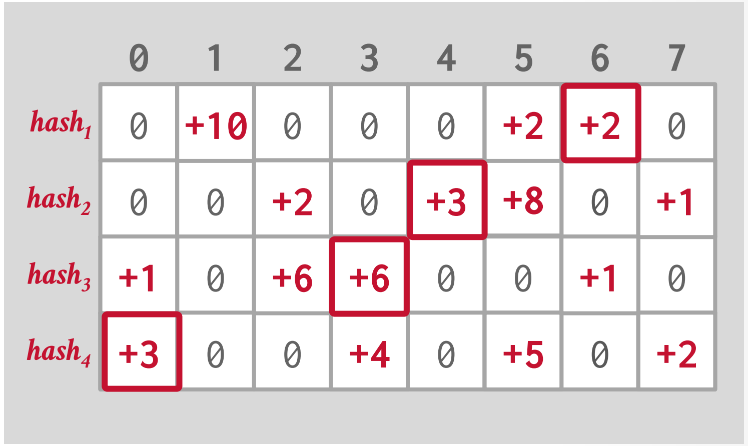
$hash_4('ODB') = 5000 \% 8 = 0$

COUNT-MIN SKETCH

Probabilistic data structure that approximates frequency counts of elements in a data stream using hash functions and a multi-dimensional array of counters.

Approximates answers with tunable accuracy and space trade-offs.

Count-Min Sketch



	0	1	2	3	4	5	6	7
hash$_1$	0	+10	0	0	0	+2	+2	0
hash$_2$	0	0	+2	0	+3	+8	0	+1
hash$_3$	+1	0	+6	+6	0	0	+1	0
hash$_4$	+3	0	0	+4	0	+5	0	+2

`GET 'ODB'` $\left\{ \; \mathit{Min}(2,3,6,3) = 2 \right.$

HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

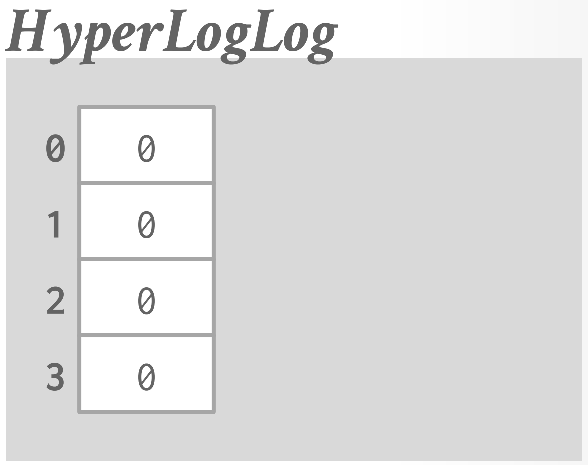
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



```
INSERT 'ODB'
```

hash('ODB') = 9022

HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

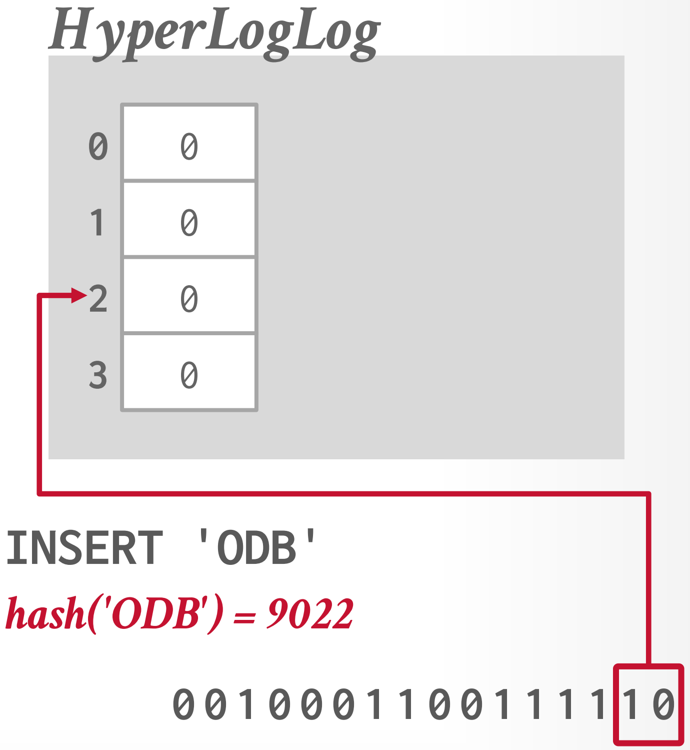
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

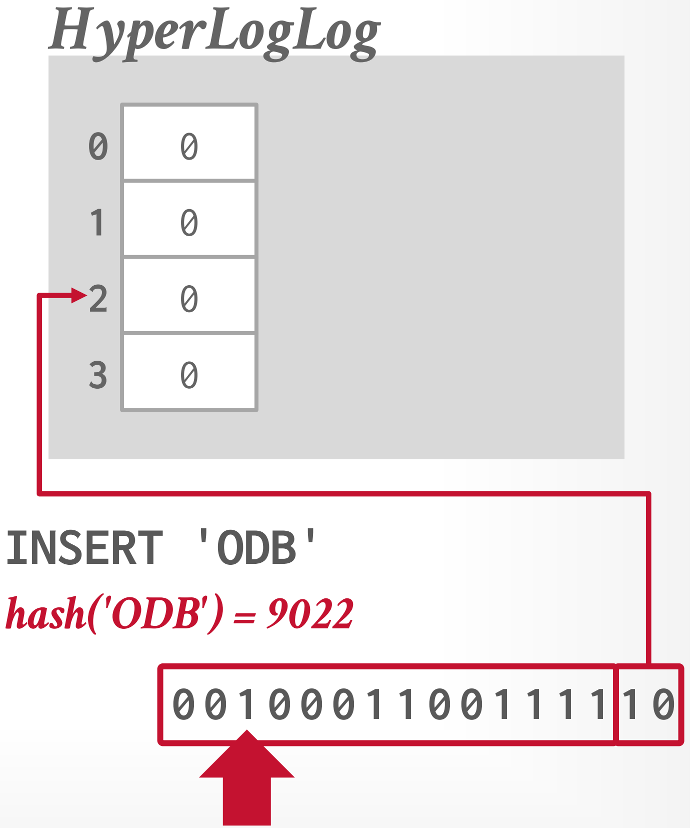
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

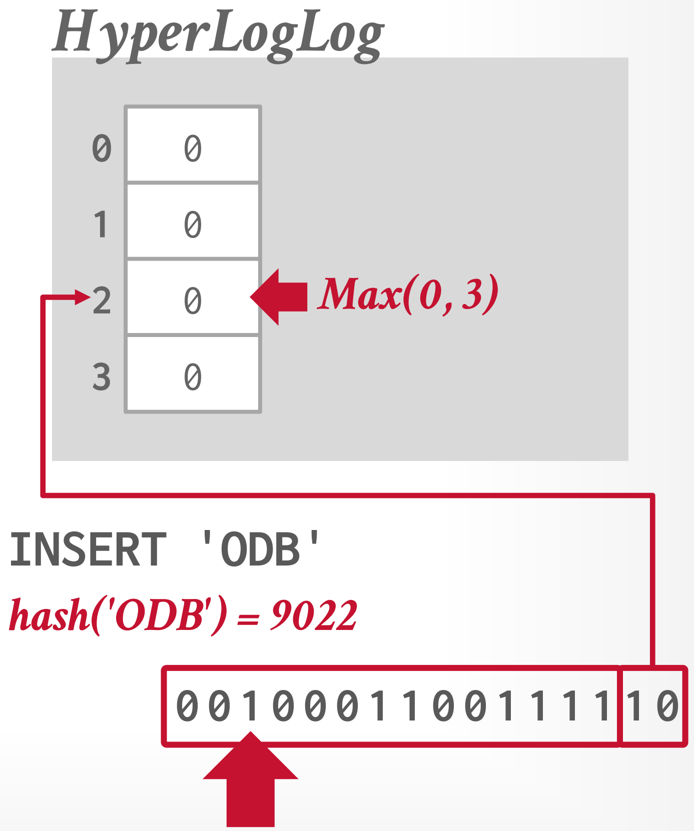
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

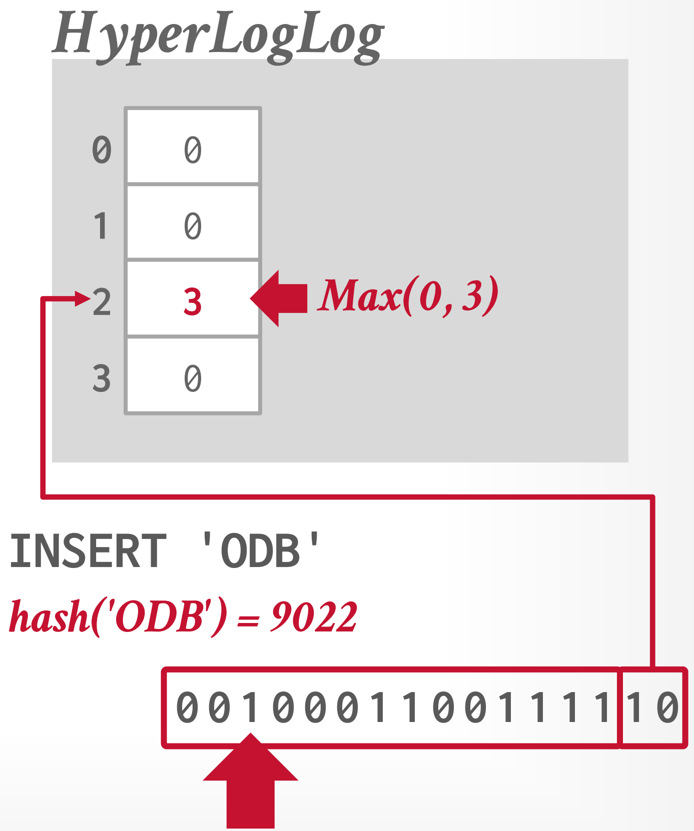
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



HYPERLOGLOG

Probabilistic data structure to approximate cardinality of a multiset.

→ Store ***m*** fixed-size array of counters.

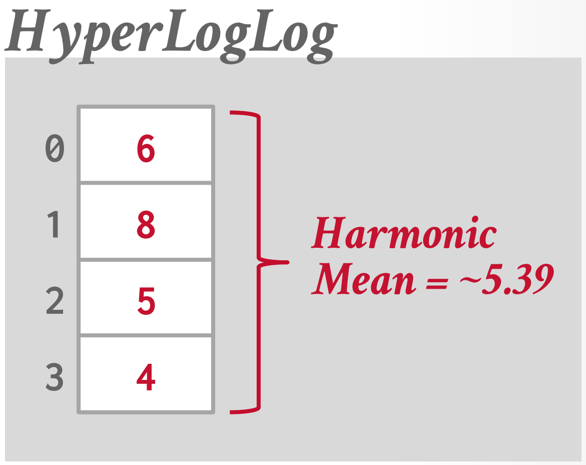
Update:

→ The first ***b*** bits of the hash determine which counter to update.

→ Calculate the position of the leftmost 1-bit in remaining bits.

Estimate:

→ Compute the Harmonic mean across counters and correct with a corrective fudge factor.



MERGING SKETCHES

Combine sketches from partitioned data sources while preserving accuracy constraints.

Count-Min Sketch:

→ Merge by summing corresponding counters in arrays.
→ Requires same parameters and hash functions for all sketches

HyperLogLog:

→ Merge by taking register-wise maximum values
→ Preserves estimation properties of individual sketches
→ Enables union of distinct values from multiple sets.

Source: EQOP Book

SAMPLING

Execute a predicate on a random sample of the target data set. The number of tuples to examine depends on the size of the original table.

Approach #1: Maintain Read-Only Copy

→ Periodically refresh to maintain accuracy.

Approach #2: Sample Real Tables

→ Use **READ UNCOMMITTED** isolation.

→ May read multiple versions of same logical tuple.

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	63	Rested
1002	Biden	82	Old
1003	Tupac	25	Dead
1004	Bieber	30	Crunk
1005	Andy	43	Sickly
1006	TigerKing	61	Jailed

⋮

1 billion tuples

Table Sample

1001	Obama	63	Rested
1003	Tupac	25	Dead
1005	Andy	43	Sickly

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age)
  FROM people
 WHERE age > 50
```

id	name	age	status
1001	Obama	63	Rested
1002	Biden	82	Old
1003	Tupac	25	Dead
1004	Bieber	30	Crunk
1005	Andy	43	Sickly
1006	TigerKing	61	Jailed

⋮

1 billion tuples

Table Sample

1001	Obama	63	Rested
1003	Tupac	25	Dead
1005	Andy	43	Sickly

sel(age>50) = 1/3

COST MODEL IMPLEMENTATIONS

PostgreSQL

IBM Db2

Smallbase (TimesTen)

DuckDB

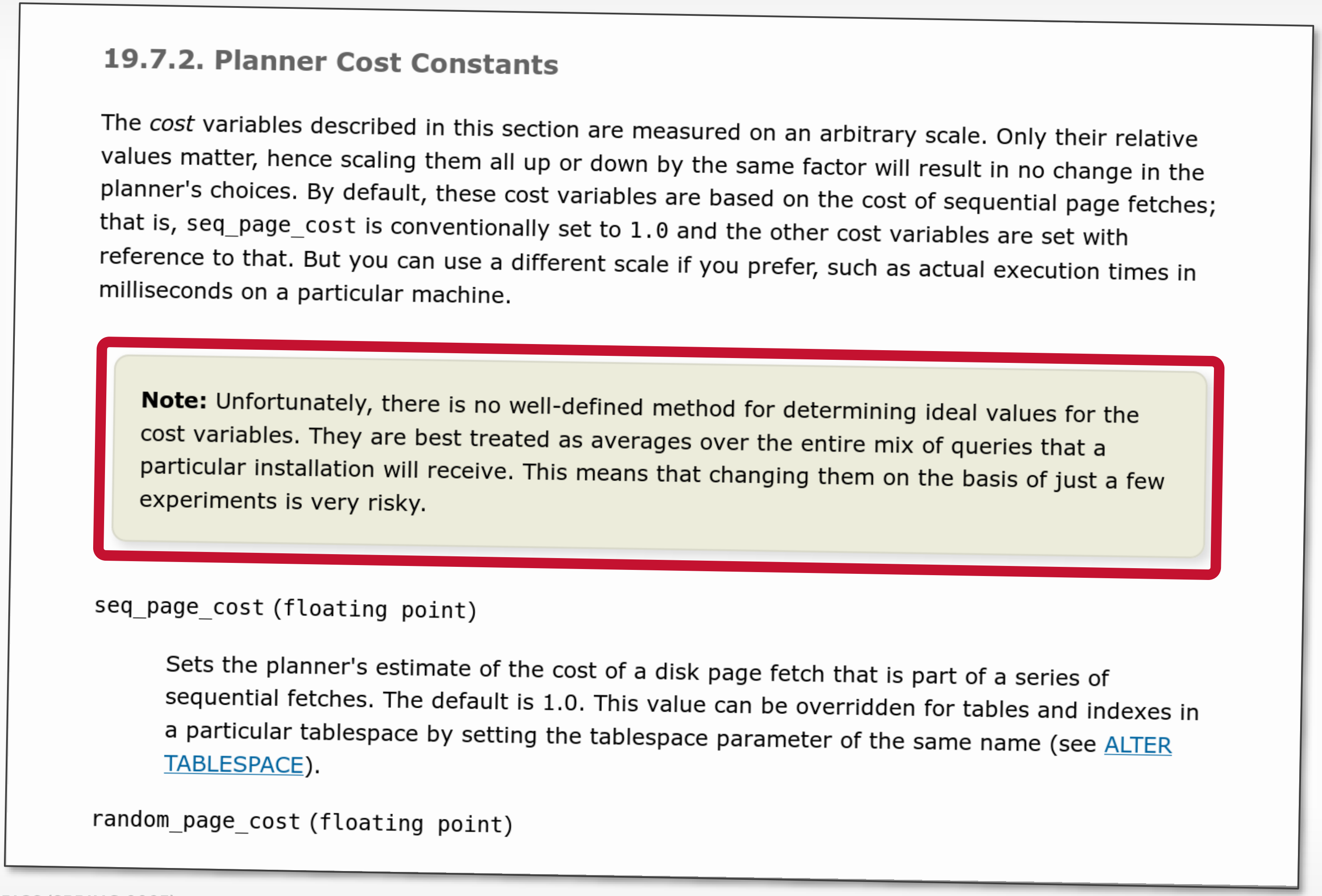
POSTGRESQL COST MODEL

Uses a combination of CPU and I/O costs that are weighted by “magic” constant factors.

Default settings are obviously for a disk-resident database without a lot of memory:

→ Processing a tuple in memory is **400x** faster than reading a tuple from disk.

→ Sequential I/O is **4x** faster than random I/O.



19.7.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, `seq_page_cost` is conventionally set to `1.0` and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

Note: Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`random_page_cost` (`floating point`)

COST MODEL CALIBRATION

The physical cost of an operator in a query plan is the amount of resources it will consume for the number of tuples it ingests/emits.

Enterprise DBMSs automatically determine the relative weights of hardware resources.
→ Hand-crafted Synthetic Queries (System R)
→ Micro-benchmarks (IBM DB2)
→ Machine Learning

Source: EQOP Book

IBM DB2 COST MODEL

Database characteristics in system catalogs

Hardware environment (microbenchmarks)

Storage device characteristics (microbenchmarks)

Communications bandwidth (distributed only)

Memory resources (buffer pools, sort heaps)

Concurrency Environment
- → Average number of users
- → Isolation level / blocking
- → Number of available locks

Source: Guy Lohman

SMALLBASE COST MODEL

Two-phase model that automatically generates hardware costs from a logical model.

Phase #1: Identify Execution Primitives

→ List of ops that the DBMS does when executing a query
→ Example: evaluating predicate, index probe, sorting.

Phase #2: Microbenchmark

→ On start-up, profile ops to compute CPU/memory costs
→ These measurements are used in formulas that compute operator cost based on table size.

MODELLING COSTS FOR A MM-DBMS
REAL-TIME DATABASES 1996

OBSERVATION

We discussed how query optimizers rely on cost models derived from statistics extracted from data.

But how can the DBMS optimize a query if there are <u>no</u> statistics?
→ Data files the DBMS has never seen before.
→ Query APIs from other DBMSs (connectors).

IN-SITU DATA PROCESSING

Execute queries on data files residing in shared storage (e.g., object store) without first ingesting them into the DBMS (i.e., managed storage).

→ This is what people usually mean when they say **data lake**.
→ A **data lakehouse** is the DBMS that sits above all this.

The goal is to reduce the amount of prep time needed to start analyzing data.

→ Users are willing to sacrifice query performance to avoid having to re-encode / load data files.

DREMEL: A DECADE OF INTERACTIVE SQL ANALYSIS AT WEB SCALE
VLDB 2020

DUCKDB COST MODEL

Cannot assume there are statistics because the DBMS may be seeing a data file for the first.

When there are no statistics, the DBMS uses number of distinct values to determine worst-case cardinality estimation for joins.

→ Assumes primary-foreign key joins.
→ Assume independence and uniformity of data.
→ If HyperLogLog is available, use that when possible (e.g., `value=10`). $\boxed{\text{Otherwise, assume 20\% selectivity.}}$

JOIN ORDER OPTIMIZATION WITH (ALMOST) NO STATISTICS
TOM EBERGEN – VRIJE UNIVERSITEIT MS THESIS (2022)

PARTING THOUGHTS

Statistics allow the optimizer to summarize the contents of the database.

→ These data structures are only approximations of real data.

Then the optimizer guesses how many tuples it will examine or emit at each operator in a query plan.

→ These guesses are also going to be approximations of what a real predicate will do.

We're generating approximations on top of approximations...

NEXT CLASS

How to use the data structures from today's lecture to estimate the cardinality of operators + predicates.