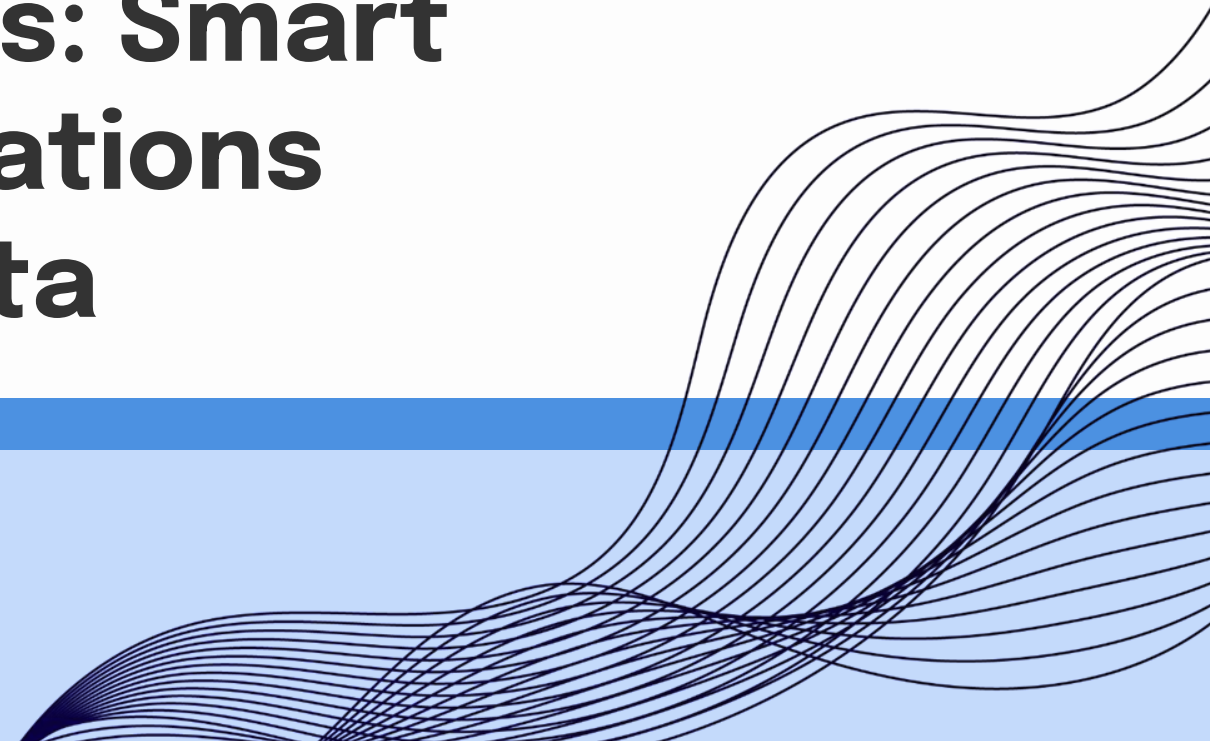
Devoxx France 2025

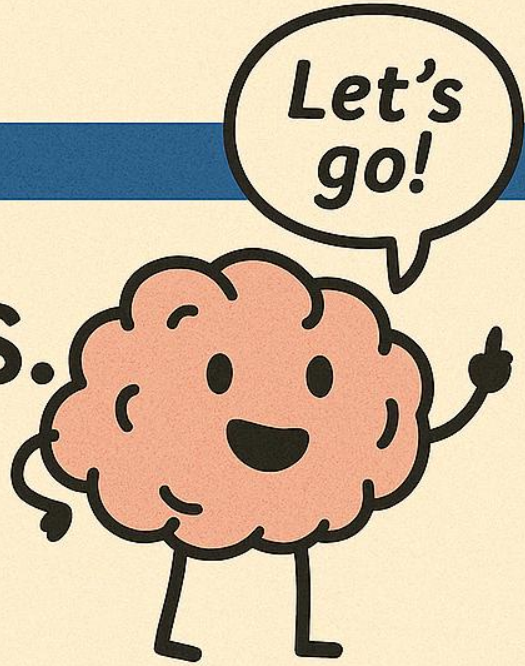
Probabilistic Data Structures: Smart Approximations for Big Data

Ben Khalfallah Héla



PARCE QUE MÈME MES SLIDES ON BESOIN DE PRATIQUER LEUR ANGLAIS:

Slides en anglais.
Présentation
en français.



BIG DATA

UNDER THE HOOD

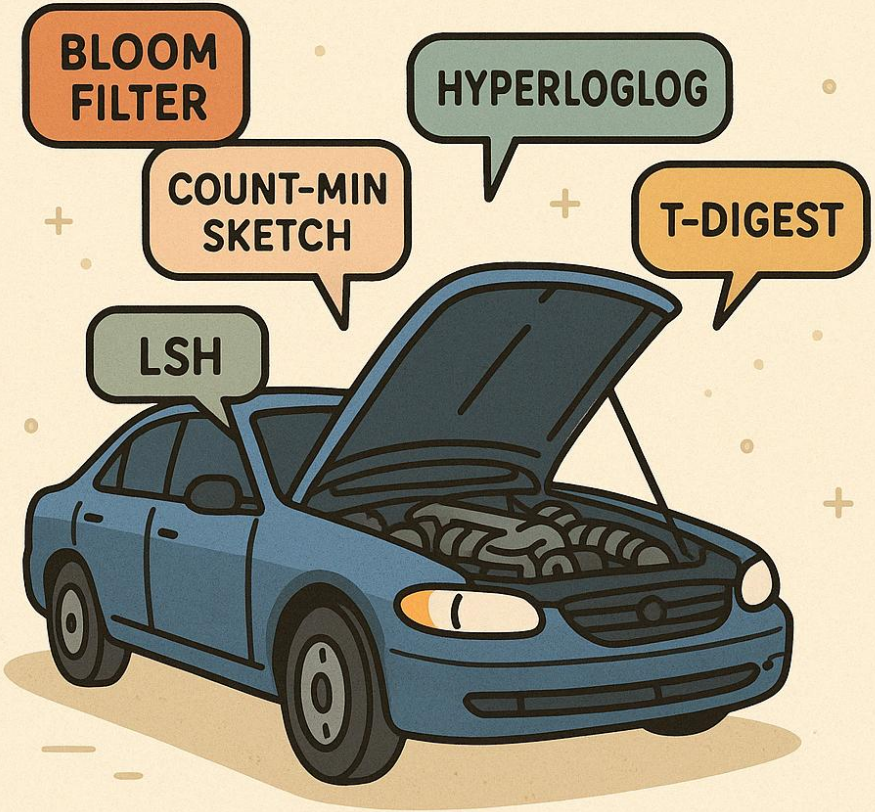


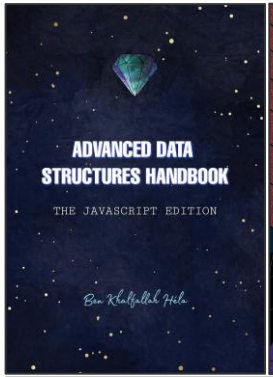
Table of contents

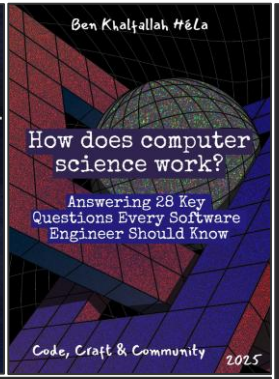
Who am I?

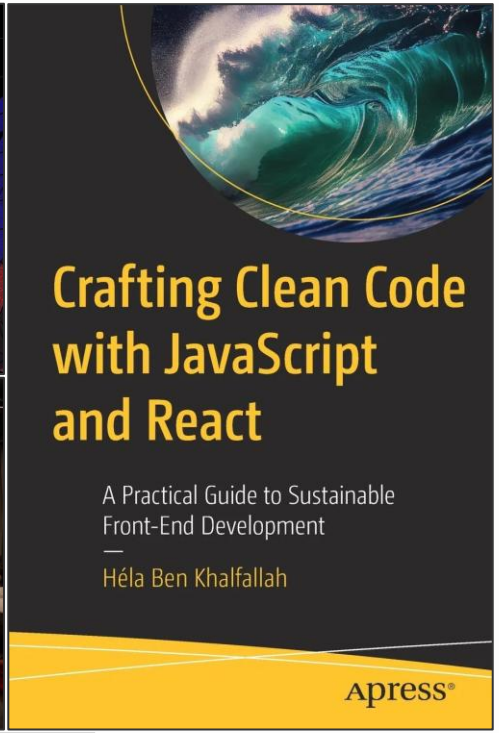
I'm Héla Ben Khalfallah, a senior software engineer passionate about building scalable, maintainable, and high-performance applications. I specialize in web solutions, deep software design, and architecture, always seeking deterministic solutions to complex problems.

I've authored books, spoken at conferences, and led numerous software meetups. Driven by curiosity, I love exploring how mathematical and scientific concepts bring determinism, structure, and clarity to software engineering.

Today, I'll show you how probabilistic data structures help us tackle Big Data efficiently through smart, memory-friendly approximations.





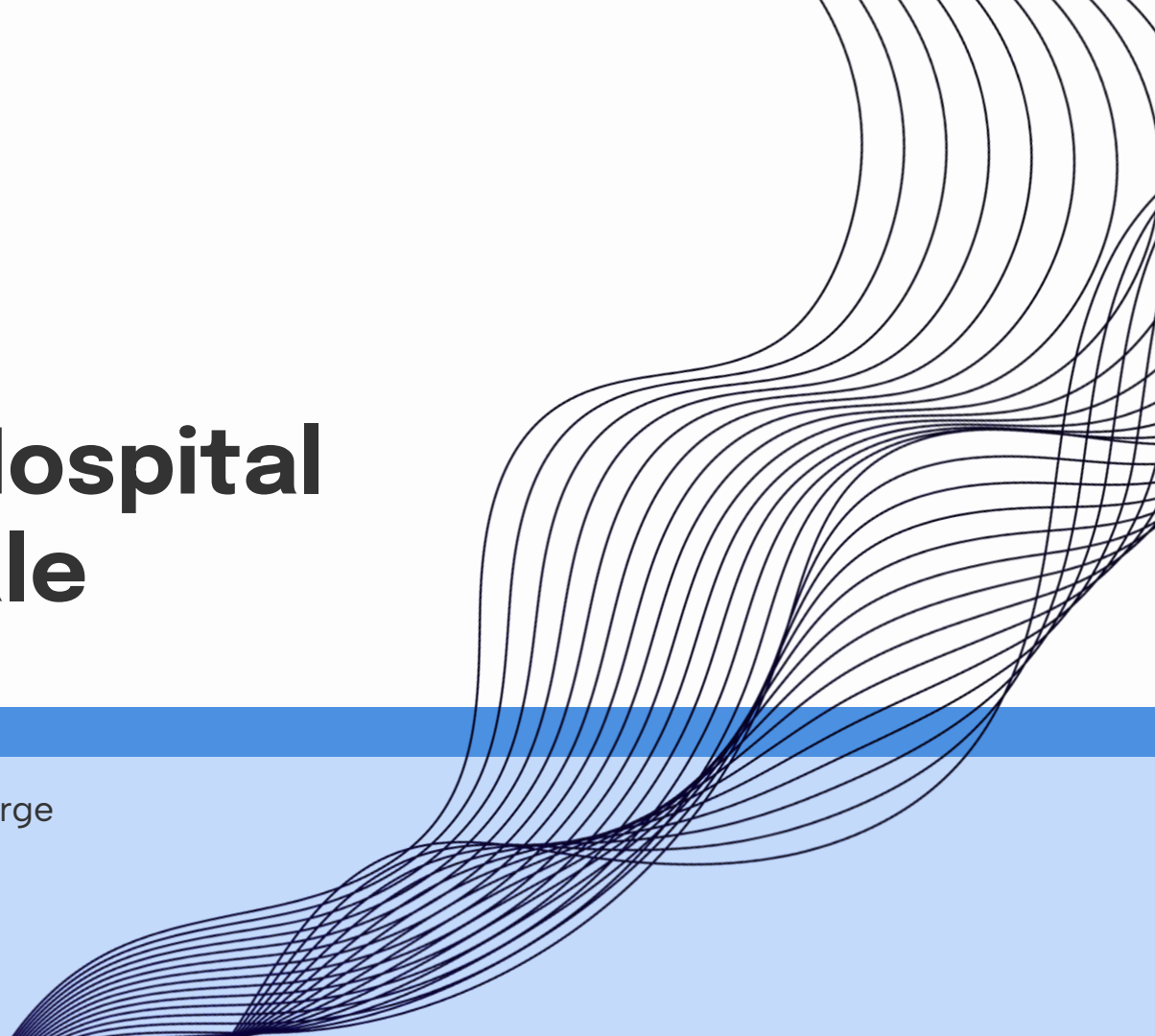


https://helabenkhalfallah.com/
https://helabenkhalfallah.medium.com/
https://x.com/b_k_hela
https://github.com/helabenkhalfallah

01

Use Case: Hospital Data at Scale

Designing a data system for a large hospital network that processes millions of events daily across departments and locations.

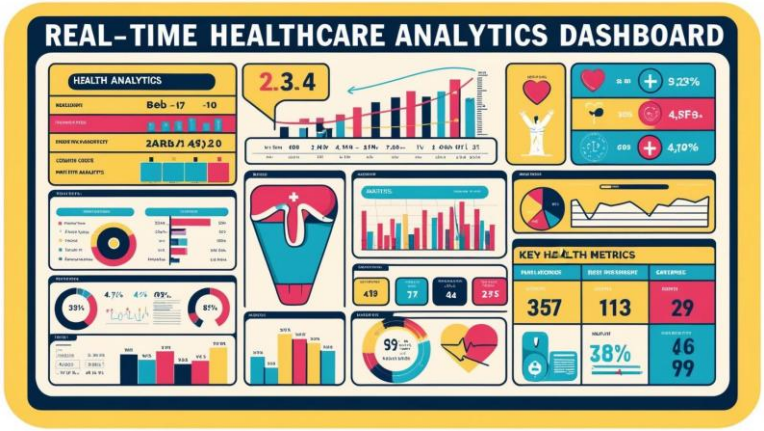


Use Case: Hospital Data at Scale

- A large hospital system handles millions of patient-related events daily: admissions, diagnostics, prescriptions, vitals, lab results, and device monitoring.
- Data flows from multiple departments and locations, and must be processed in real time.
- The system must support:
    - Fast decision-making (triage, alerts, clinical recommendations)
    - Resource optimization (ER wait times, medication inventory)
    - Scalability without overwhelming memory or compute resources

Challenges in Real-Time Healthcare Analytics

- Has this patient been admitted before across any facility?
- How many unique patients have visited this month across all hospitals?
- What are the most frequently prescribed medications over the past 24 hours?
- Which symptoms are reported most frequently this week?
- What is the 95th percentile of emergency room wait times today?
- Which patients have similar symptom histories or diagnoses?



Limitations of Traditional Data Structures & Databases

Not Memory-Efficient

Even with log-time access, structures like B-trees and LSMs require growing space as data scales.

Not Designed for Streams

Most assume bounded or batch data, not continuous event-driven input.

Latency Under Load

Aggregations like distinct counts, quantiles, and top-k slow down as volume increases.

Merge Overhead

Combining data across nodes or systems requires coordination and consistency handling.

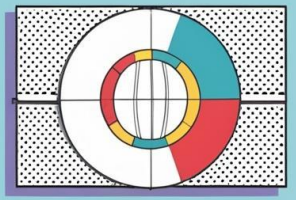
Precision-Centric

Built for exactness, even when approximate answers would suffice operationally.

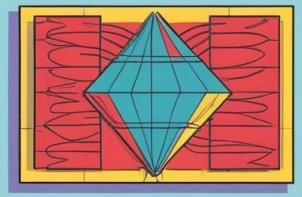
Limited Scalability in Real-Time Systems

Performance and responsiveness degrade under high-frequency, distributed workloads.

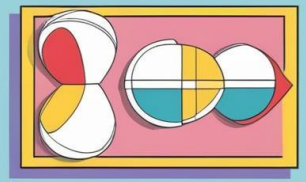

WE NEED
SOMETHING
NEW



BLOOM FILTER

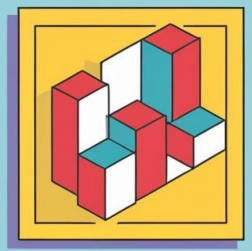


COUNT-MIN SKETCH

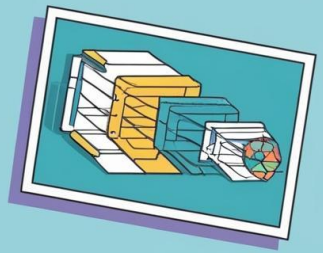


T-DIGEST

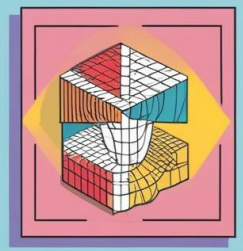
PROBABILISTIC DATA STRUCTURES



HYPERLGLOG



MINHASH



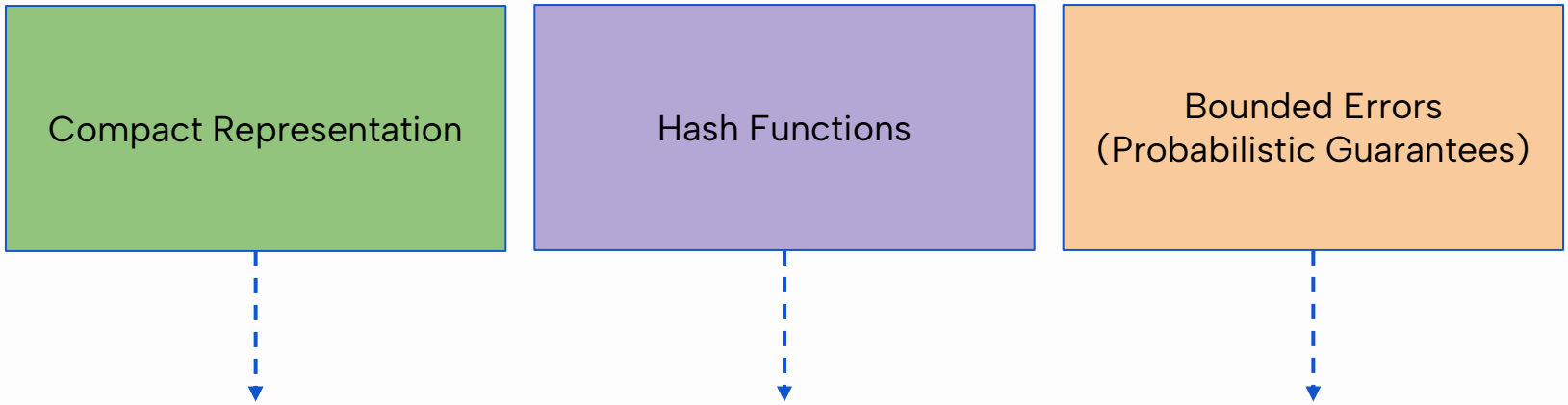
SIMHASH

02

Foundations of Probabilistic Data Structures

Introducing the key principles behind probabilistic structures, and why they're a better fit for large-scale, real-time analytics.

Core principles



Probabilistic data structures use fixed-size, space-efficient memory (e.g., bit arrays, sketches) to represent large data sets.

Input elements are transformed into positions or values in a compact structure using one or more hash functions.

They sacrifice perfect accuracy to gain performance and efficiency, but within known error bounds.

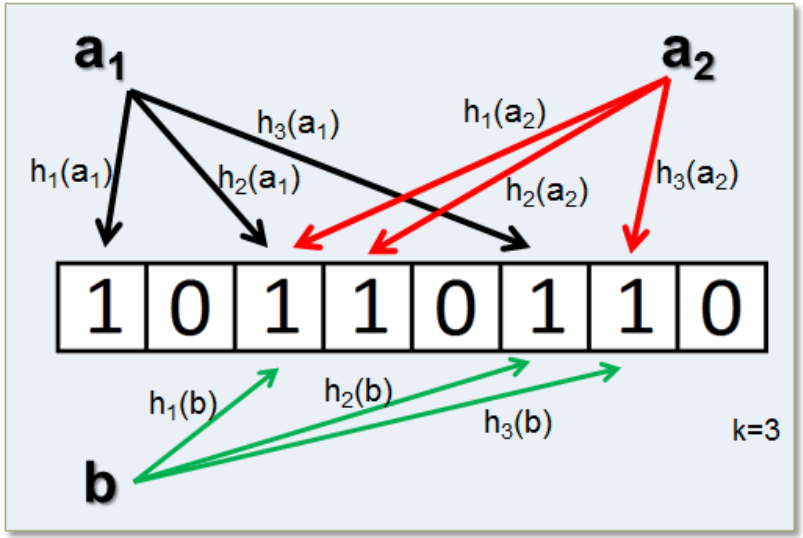
Compact representation and Hash Functions

Compact Bit Array Representation:

- Concept: Each item updates a few bits in a fixed-size array; the item itself is not stored.
- Why: Keeps memory usage constant, even as the number of elements grows.

Hash Functions as a Lightweight Mapping:

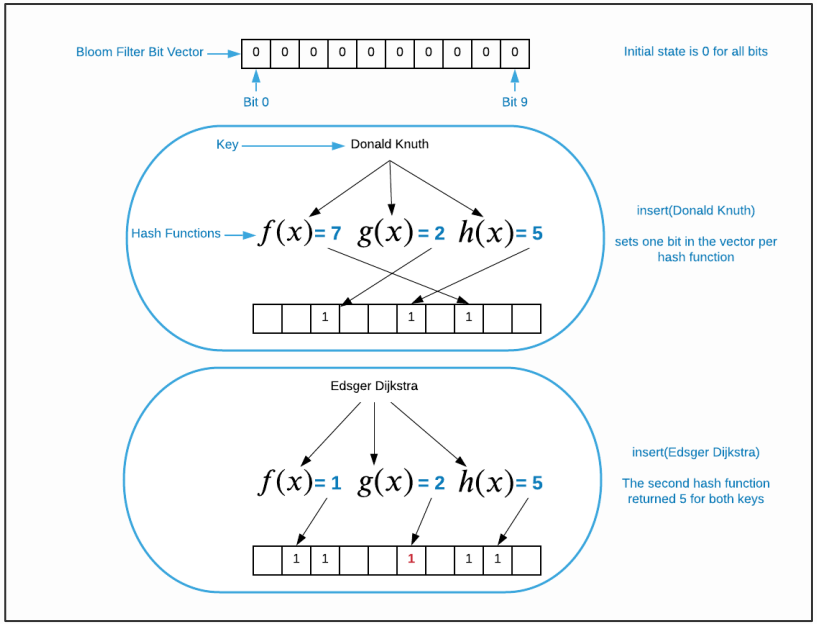
- Concept: Items are transformed into numerical positions via deterministic hash functions.
- Why: Enables fast and uniform mapping to fixed-size structures, with no extra memory per item.



https://www.allthingsdistributed.com/2017/02/bloom-filters.html

Compact representation and Hash Functions - Insertion

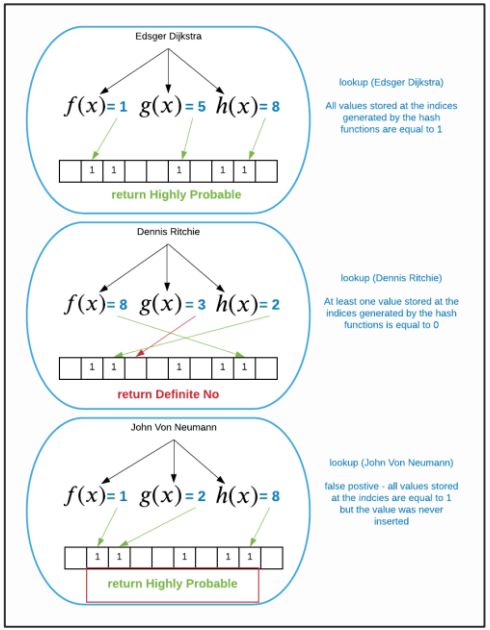
- Each hash function f(x), g(x), h(x) maps a key to a specific bit position:
  - Donald Knuth: sets bits at positions 7, 2, 5.
  - Edsger Dijkstra: sets bits at positions 1, 2, 5.
- Inserting an item means setting bits in a fixed-size bit array.
- No element is stored, only bitwise evidence of presence.



https://hideoushumpbackfreak.com/algorithms/data-struct-bloom-filter.html

Compact representation and Hash Functions - Lookup

- Lookup is the reverse of insert: instead of setting bits, it checks if they are all set to 1.
- Each hash function computes a position in the bit array.
  - If all bits at those positions are 1, the item is probably present.
    - Edsger Dijkstra: All bits are 1.
    - Return: "Highly Probable".
  - If any bit is 0, the item is definitely not present.
    - Dennis Ritchie: One bit (at position 3) is 0.
    - Return: "Definite No".



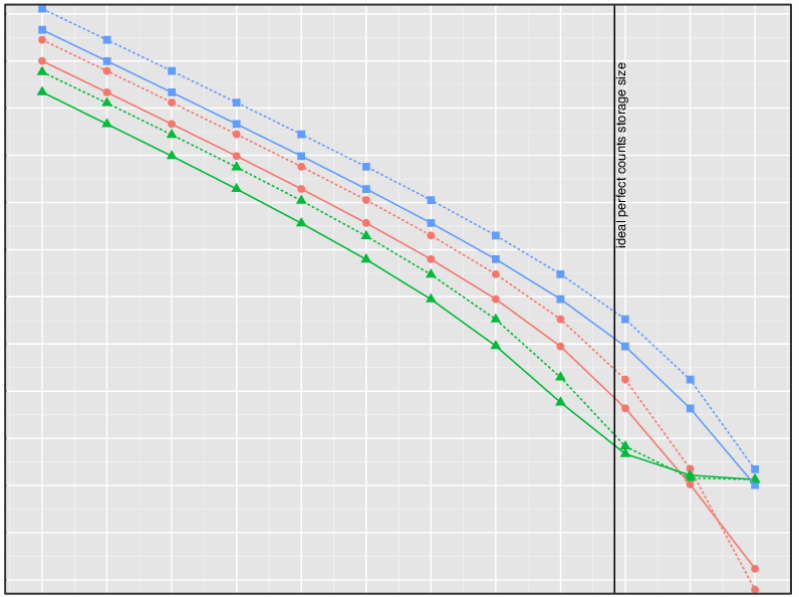
https://hideoushumpbackfreak.com/algorithms/data-struct-bloom-filter.html

Bounded Errors - Probabilistic Guarantees

For example, Count-Min Sketch often overestimates frequencies, but we can bound the error using two tunable parameters:

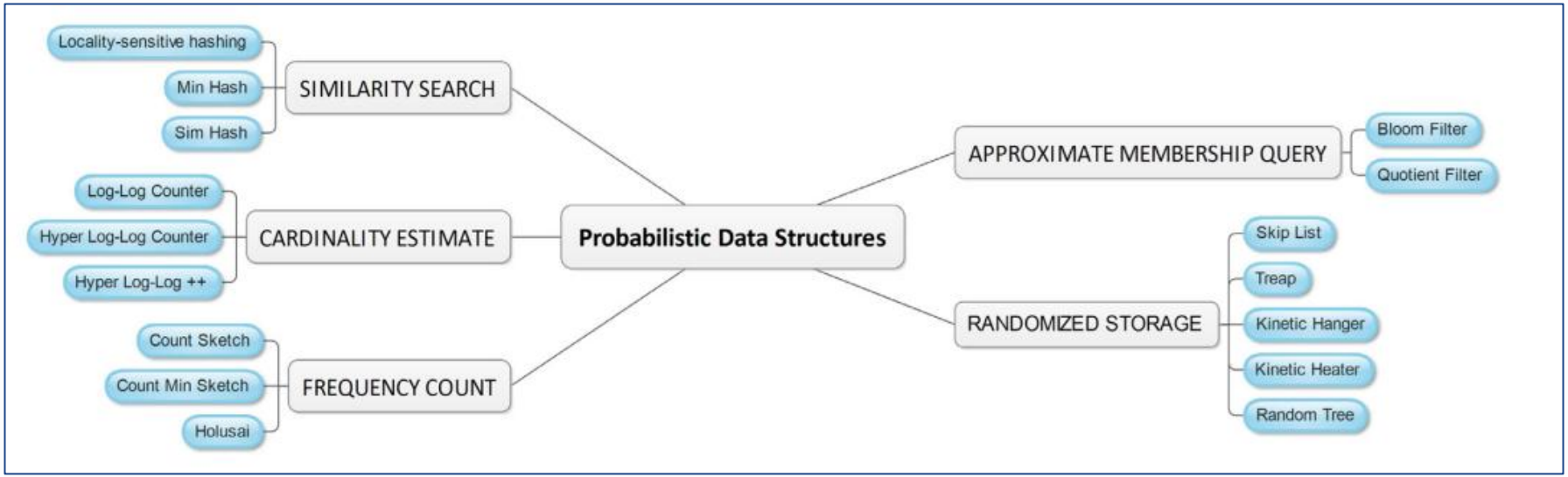
- Relative Error (ε): Width
  - $w = \lceil e / \varepsilon \rceil$ limits the overestimation to a factor of ε.
- Failure Probability (δ): Depth
  - $d = \lceil \ln(1 / \delta) \rceil$ bounds the error probability to within δ.

Together, w and d are configured to balance memory efficiency with an acceptable error rate for the application.



Average Relative Error of estimated counts with Count-Min Sketch, Count-Min-Log 16bits and Count-MinLog 8bits.

Probabilistic Data Structures Types



https://www.sciencedirect.com/science/article/abs/pii/S0950705119304071

03

Structure-by-Structure Walkthrough

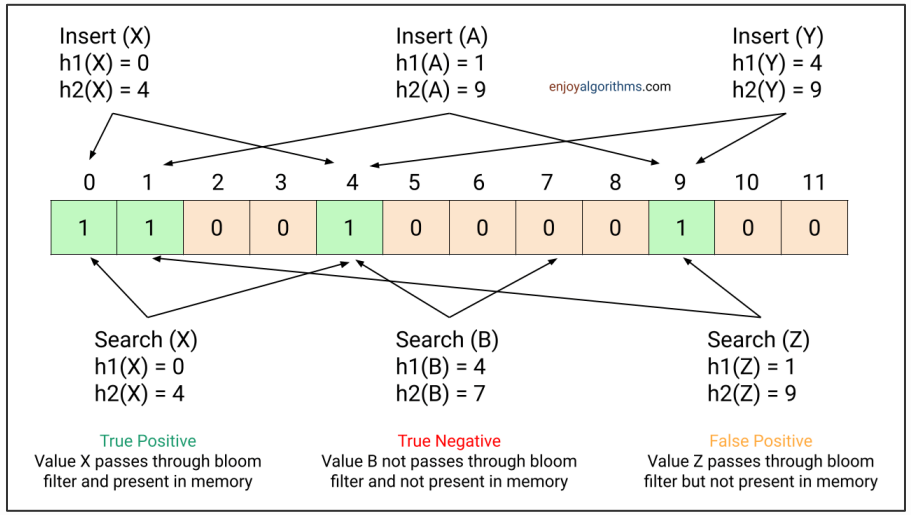
Mapping each real-world challenge to a specific probabilistic structure, with concepts, trade-offs, and real-world examples.

Bloom Filter

Has this patient been admitted before across any facility?

Bloom Filter - Foundation

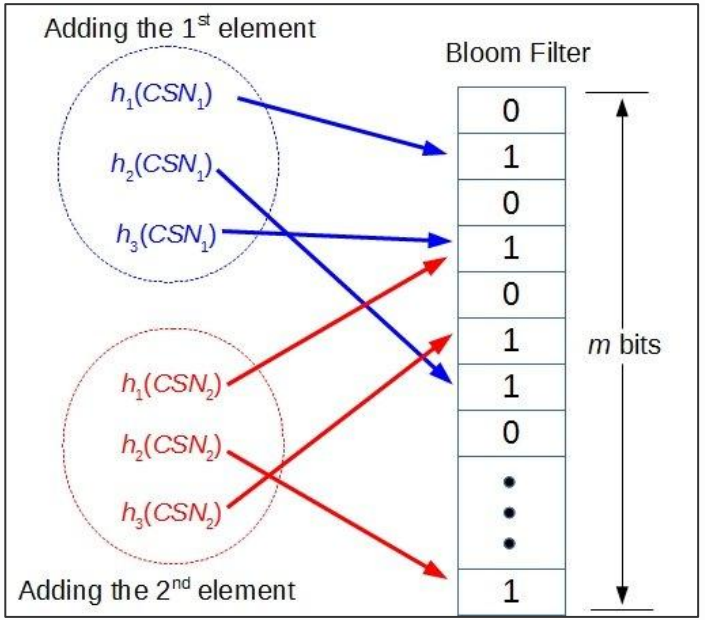
- Bloom filters are a compact, memory-efficient data structure ideal for applications that require rapid, probabilistic membership testing rather than absolute accuracy.
- It provides a definitive ‘no’ for elements not in the set and a highly probable ‘yes’ for those that are.
- Bloom filters can yield false positives but never false negatives.



https://www.enjoyalgorithms.com/blog/bloom-filter

Bloom Filter - Components

- Bloom Filter has two primary components:
  - Bit Array (or bit vector, bucket array):
    - A fixed-size array of m bits, all initially set to 0.
    - Serves as the compact memory space in which information about the inserted elements is encoded.
  - A set of k independent hash functions used to transform each input item into k indices within the bit array.



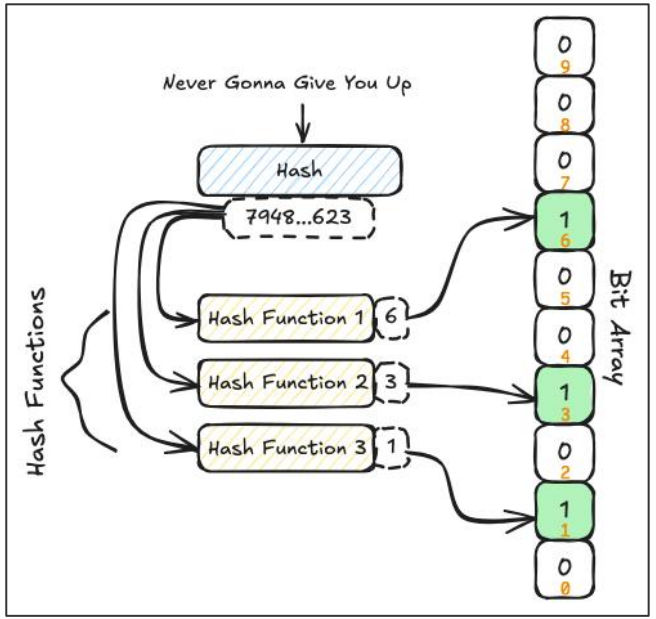
https://www.researchgate.net/figure/The-Bloom-filter-concept-adding-two-elements-K-3-hash-functions_fig2_318440316

Choosing Hash Functions

Selecting appropriate hash functions is critical for Bloom filter performance and accuracy. The ideal hash functions must satisfy two key properties:

- Independence: Each hash function should produce results uncorrelated with the others to avoid redundant bit assignments.
- Uniform Distribution: Hash outputs should evenly spread across the bit array to prevent clustering and excessive false positives.

Additionally, performance is paramount. While cryptographic hash functions (e.g., SHA-1, MD5) offer strong collision resistance, they are computationally expensive and generally unnecessary for Bloom filters.



https://www.bytedrum.com/posts/bloom-filters/

Recommended Hash Functions

Hash Function	Advantages	Use Case
MurmurHash	Fast, high-quality distribution	Widely used in databases and large-scale systems
xxHash	Extremely fast with low collision rate	Ideal for high-throughput applications
FNV (Fowler–Noll–Vo)	Simple and efficient	Suitable for small-scale or embedded systems

Note: Benchmarks have shown that switching from MD5 to MurmurHash can yield performance improvements of up to 800% in insertion-heavy applications.

Sizing Bloom Filter

- One of the advantages of Bloom filters is their adjustable false positive rate. The false positive rate is approximately:
  - `P(false positive) ≈ (1 - e^(-kn/m))^k`
    - m: number of bits.
    - k: number of hashes functions.
    - n: the number of items expected to be stored.
- To find the **optimal number of hash functions** given **m** and **n**, use:
  - `k = (m/n) * ln(2)`
- The number of hash functions k affects both the Bloom filter's accuracy and speed:
  - Too few hash functions: higher false positive rate
  - Too many hash functions: slower insertions and lookups

Steps for Configuring a Bloom Filter:

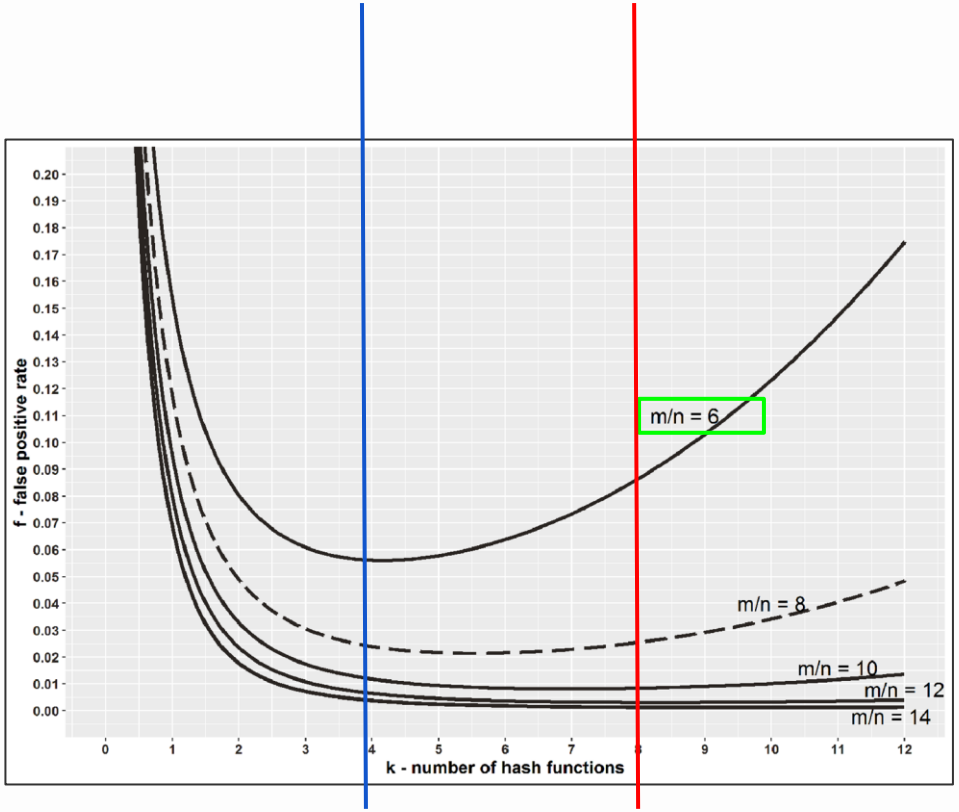
1. Estimate the number of elements n to be inserted.
2. Choose a bit array size m based on acceptable memory use and error rate.
3. Calculate the optimal number of hash functions k using the formula.
4. Adjust m if needed to meet your target false positive rate.

https://hur.st/bloomfilter/

Sizing Bloom Filter

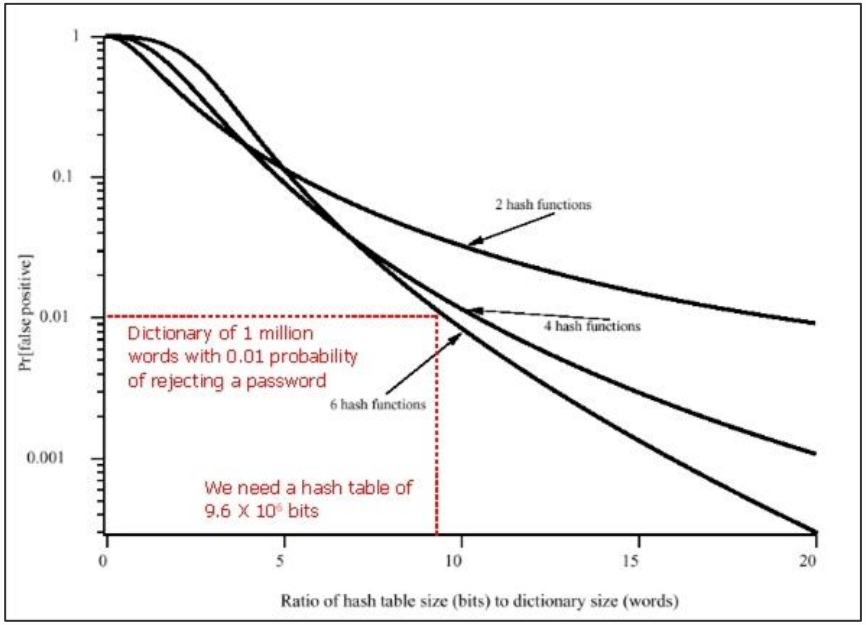
For a given Bloom filter size, there's an optimal number of hash functions that minimizes false positives, and exceeding that number increases error due to oversaturation of the bit array.

Example:
At m/n=6, the optimal k=4 (hash functions) gives a false positive rate of ~6%, while increasing to k=8 raises it to ~10% due to oversaturation.



https://www.manning.com/books/algorithms-and-data-structures-for-massive-datasets

Performance of Bloom Filter

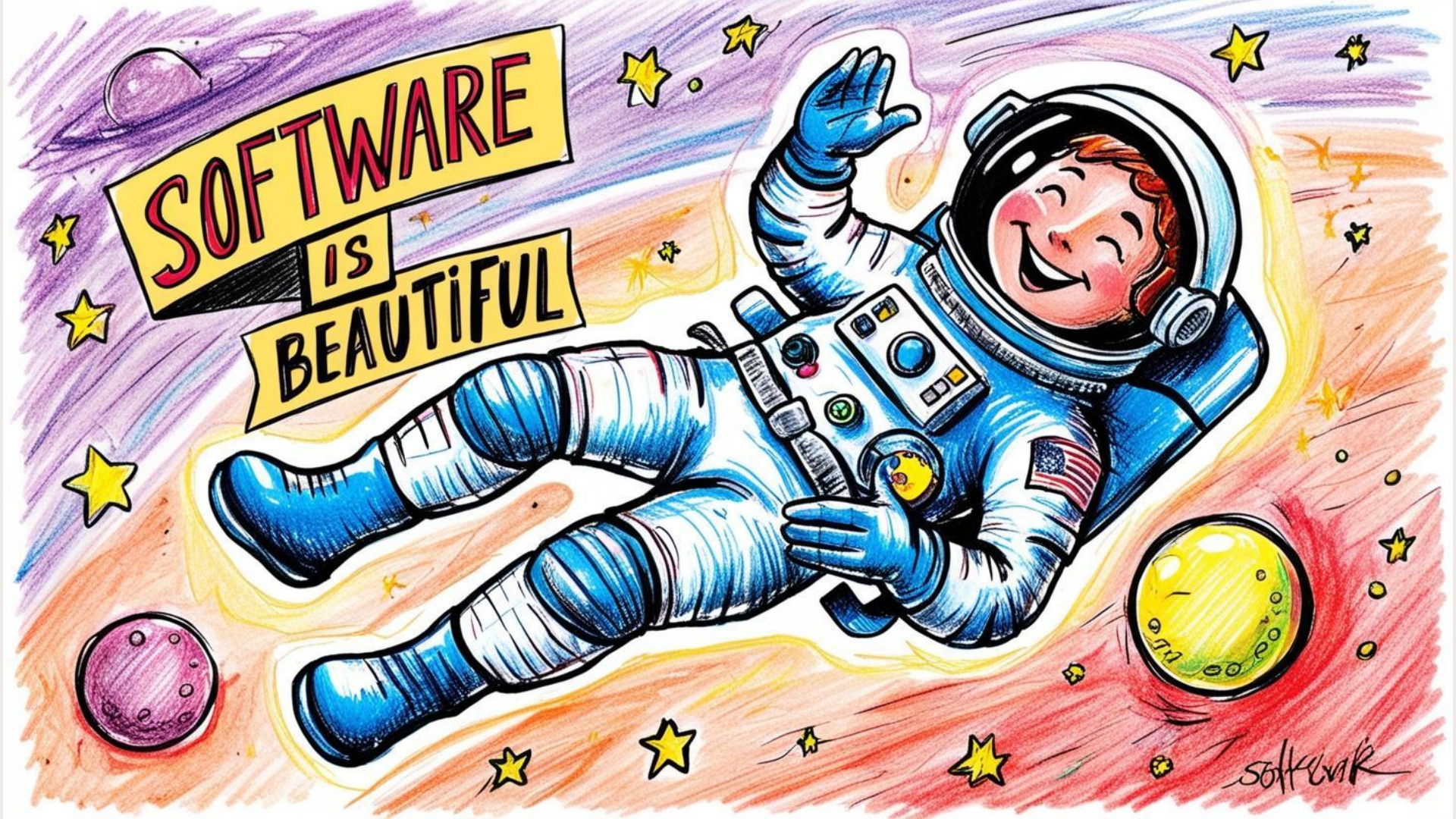


https://cs.hofstra.edu/~cscvjc/Spring06/Slides/Sess09/img62.html

A Bloom filter-based rejector is commonly used to:

1. Prevent use of weak or common passwords
   Check against large known datasets of breached passwords (e.g., “123456”, “password”, etc.).
2. Prevent password reuse
   Quickly verify if the user is attempting to reuse one of their previously chosen passwords.
3. Enforce custom banned password policies
   Block passwords containing blacklisted substrings, patterns, or organizational terms.

A Bloom filter with ~9.6 million bits and 6 hash functions can efficiently block 1 million known bad passwords with a false positive rate of 1%!


SOFTWARE
IS
BEAUTIFUL

Bloom Filter - Implementation

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/probabilistic/membership/BloomFilter.ts

Bloom Filter - Visualisation

https://www.jasondavies.com/bloomfilter/

Real-World Applications - Redis / RedisBloom

RedisBloom extends Redis with probabilistic data structures:

- BF.RESERVE key error_rate capacity
  - Initializes a Bloom filter with expected capacity and target error rate.
  - Example: BF.RESERVE myfilter 0.01 1000000 → 1M entries with 1% error.
- BF.ADD key item
  - Adds an item. If the item was not present, returns 1, otherwise 0.
- BF.EXISTS key item
  - Checks if the item might exist.
  - Returns:
    - 1 if maybe present
    - 0 if definitely not present
- BF.MADD key item1 item2 ...
  - Bulk add multiple items (multi-threading friendly)
- BF.MEXISTS key item1 item2 ...
  - Bulk check for existence

```
> BF.RESERVE usernames 0.001 1000000
OK

> BF.ADD usernames 'alice'
(integer) 1

> BF.EXISTS usernames 'alice'
(integer) 1

> BF.EXISTS usernames 'bob'
(integer) 0
```

Real-World Applications - Redis / RedisBloom

Action	Performance
Insertion	~370,000 ops/sec
Lookup	~440,000 ops/sec
Scaling	Automatic (new filters appended)
Memory	~5 bytes per element with 0.01% FPR

https://redis.io/blog/bloom-filter/
https://www.youtube.com/watch?v=2SpYbEfp4vA

Use Cases:

- Prevent reuse of usernames or passwords (Username Rejection).
- Avoid reprocessing already-seen items (deduplication).
- Prefilter expensive storage (disk, DB, network) with a fast in-memory test.

Real-World Applications - PostgreSQL Bloom Index vs B-Tree Index

https://www.postgresql.org/docs/current/bloom.html

Feature	Bloom Index	B-Tree Index
Supported predicates	Equality (=) only	Equality, inequality, range, sorting
Index size (for many columns)	Very compact (bit signatures, e.g., 153 MB)	Grows linearly with columns (e.g., 531 MB)
Multi-column support	One Bloom index for many columns	Requires compound or separate indexes
False positives	Possible → needs heap recheck	No false positives
False negatives	Never	Never
Sorting support	Not supported	Supported
Range scans	Not supported	Supported
Use case	Wide-table filtering by multiple equality conditions	Indexes supporting range/ordering

Real-World Applications - PostgreSQL Bloom Index vs B-Tree Index

```
// Dataset
-- 10 million rows, 6 int columns
CREATE TABLE tbloom AS
SELECT (random() * 1e6)::int AS i1, ..., (random() * 1e6)::int AS i6
FROM generate_series(1, 10000000);
```

```
// ❌ Full Table Scan
EXPLAIN ANALYZE SELECT * FROM tbloom WHERE i2 = 898732 AND i5 = 123451;
-- Time: ~357 ms
-- Scan: Sequential (no index used)
```

```
// ❌ B-tree on (i1, ..., i6)
CREATE INDEX btreeidx ON tbloom(i1, i2, i3, i4, i5, i6);
-- Size: 386 MB
-- Query time: ~351 ms (still sequential)
```

```
// ✅ Bloom Index on (i1 to i6)
CREATE EXTENSION bloom;
CREATE INDEX bloomidx ON tbloom USING bloom(i1, i2, i3, i4, i5, i6);
-- Size: 153 MB
-- Query time: ~22 ms (Bitmap + Heap recheck)
```

Why Bloom Wins:

- Even with false positives, it can eliminate most of the 10M rows early.
- It avoids full scans by using Bitmap Index Scan.
- Much more space-efficient than B-trees for many columns.

https://www.postgresql.org/docs/current/bloom.html

Real-World Applications - Apache Spark

Apache Spark is a big data engine used to process huge amounts of data quickly.

Behind the Scenes: Bloom Filters in Spark

- Index Creation: Spark builds Bloom filters on selected columns during query execution, based on the query plan.
- Metadata Storage: Filters are stored in file metadata (e.g., Parquet, ORC) for future queries.
- Query Optimization: Spark uses Bloom metadata to skip irrelevant partitions, reducing I/O and speeding up reads.
- Fallback Strategy: If filters are ineffective (e.g., high false positives), Spark automatically falls back to full scans or alternative indexes.

https://spark.apache.org/docs/3.5.3/api/java/org/apache/spark/util/sketch/BloomFilter.html

```
from pyspark.sql import SparkSession
   from pyspark.sql.functions import col, udf
from pyspark.sql.types import BooleanType
   from pybloom_live import BloomFilter

# Start Spark
spark = SparkSession.builder.appName("BloomExample").getOrCreate()

# Sample DataFrame
df = spark.createDataFrame([
   ("alice",), ("bob",), ("carol",), ("dave",)
], ["name"])

# Create Bloom filter and add names
bloom = BloomFilter(capacity=100, error_rate=0.01)
for name in ["alice", "carol"]:
bloom.add(name)

# Broadcast the filter
bf_broadcast = spark.sparkContext.broadcast(bloom)

# Define a UDF to check membership
def might_exist(name):
return name in bf_broadcast.value

bloom_udf = udf(might_exist, BooleanType())

# Use the Bloom filter to filter the DataFrame
filtered_df = df.filter(bloom_udf(col("name")))
filtered_df.show()
```

```
+-----+
|name |
+-----+
|alice|
|carol|
+-----+
```

Real-World Applications - Apache Cassandra

Apache Cassandra is a distributed NoSQL database designed for high availability, fault tolerance, and scalability.

Read Path Optimization

- Data is stored both in RAM (Memtables) and on disk (SSTables).
- To avoid scanning every SSTable during a read, Cassandra uses Bloom filters.

Bloom Filters efficiently determine:

- The key definitely does not exist in an SSTable → skip file.
- The key might exist → perform further checks.

Bloom Filters are stored in off-heap RAM, minimizing JVM memory pressure.

https://cassandra.apache.org/doc/stable/cassandra/operating/bloom_filters.html

Adjust accuracy vs. memory usage with bloom_filter_fp_chance:

```
ALTER TABLE my_table WITH
bloom_filter_fp_chance = 0.1;
```

Lower values reduce false positives but increase memory usage.

After updating, regenerate Bloom filters by:

- Initiating compaction, or
- Running nodetool upgradesstables.

Bloom Filter - Limitations

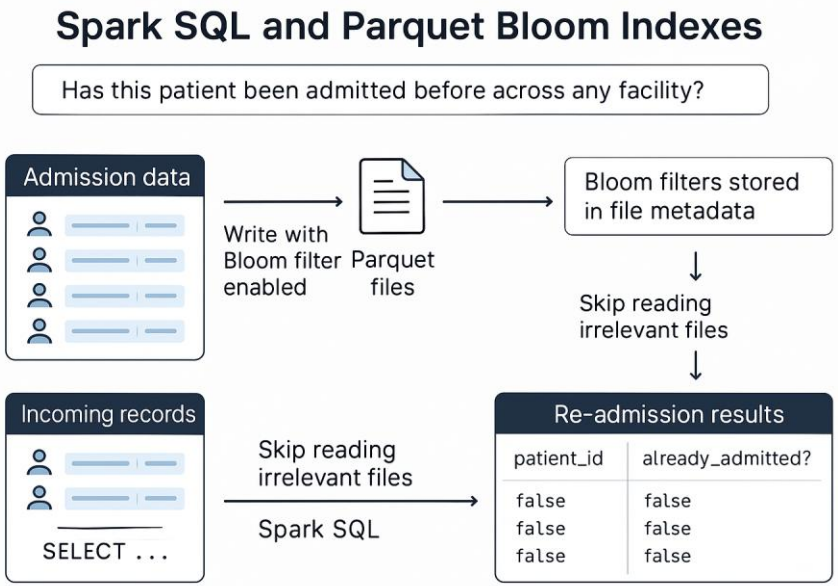
Limitation	Description
False Positives	May report that an element is in the set when it is not
No Deletion Support	Cannot remove elements once added (standard version)
No Value Storage	Only indicates presence; does not store actual data
Requires Parameter Tuning	Accuracy depends on estimated input size and false positive rate
Not Ideal for Small Sets	Overhead may be higher than benefits for small datasets
Fixed Size	Cannot resize dynamically; overfilling increases error rate

Bloom Filter - Variants

Variant	Key Feature	Supports Deletion	Use Case
Standard Bloom Filter	Probabilistic membership test using multiple hash functions	No	Simple, space-efficient presence checks
Counting Bloom Filter	Uses counters instead of bits to allow removals	Yes	Dynamic datasets with insertions and deletions
Cuckoo Filter	Uses cuckoo hashing; supports deletion and better space usage	Yes	High-performance membership tests with deletions
Parallel Bloom Filter	Optimized for parallel insert/query operations (e.g., on GPUs)	Limited	High-throughput applications and concurrent queries

Use Case: Hospital Data at Scale

- Has this patient been admitted before across any facility?
- We want to efficiently check if a patient ID already exists without scanning the entire admissions dataset.
- Bloom filters can be written into Parquet file metadata during data ingestion. Later, Spark automatically uses them to skip files that definitely don’t contain a patient ID.



Use Case: Hospital Data at Scale

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("WriteAdmissionsWithBloom").getOrCreate()

# Sample DataFrame of prior admissions
admissions_df = spark.read.csv("admissions.csv", header=True, inferSchema=True)

# Enable Bloom filter on patient_id column
spark.conf.set("spark.sql.parquet.bloomFilter.enabled", "true")
spark.conf.set("spark.sql.parquet.bloomFilter.columns", "patient_id")
spark.conf.set("spark.sql.parquet.bloomFilter.expectedNumItems", "10000000")
spark.conf.set("spark.sql.parquet.bloomFilter.fpp", "0.01")  # 1% false positive rate

# Write Parquet with Bloom filter
admissions_df.write.mode("overwrite").parquet("hdfs://hospital/admissions_parquet_bloom/")
```

Write Admission Data with Bloom Filter Enabled

Use Case: Hospital Data at Scale

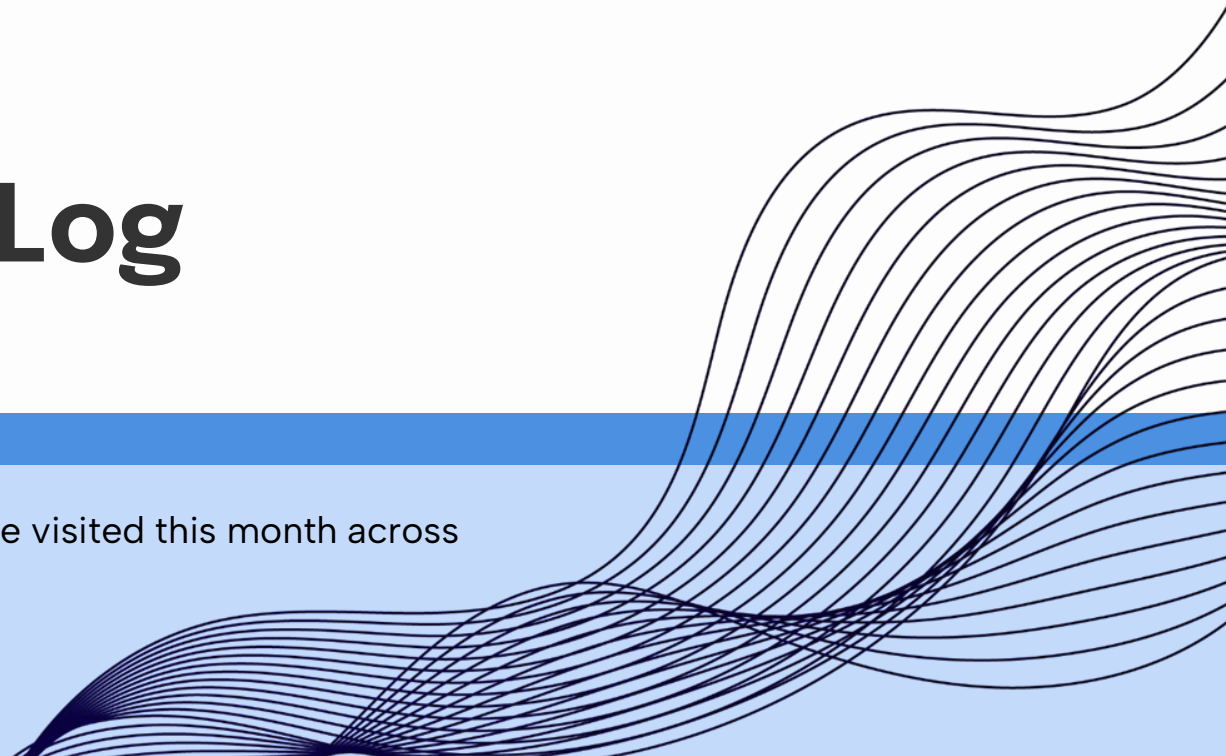
```
# Load admissions data (Bloom filters are now in metadata)
admissions_bf = spark.read.parquet("hdfs://hospital/admissions_parquet_bloom/")
admissions_bf.createOrReplaceTempView("admissions")

# Load today's incoming records
today_df = spark.read.csv("today.csv", header=True, inferSchema=True)
today_df.createOrReplaceTempView("incoming")

# Join to find re-admitted patients
result_df = spark.sql("""
SELECT i.patient_id, CASE WHEN a.patient_id IS NOT NULL THEN TRUE ELSE FALSE END AS
already_admitted
FROM incoming i
LEFT JOIN admissions a
ON i.patient_id = a.patient_id
""")
result_df.show()
```

Query Using Spark SQL

Spark will automatically prune Parquet row groups/files that do not contain the patient_id using Bloom filters. So even if there are thousands of Parquet files, it only scans the few relevant ones.

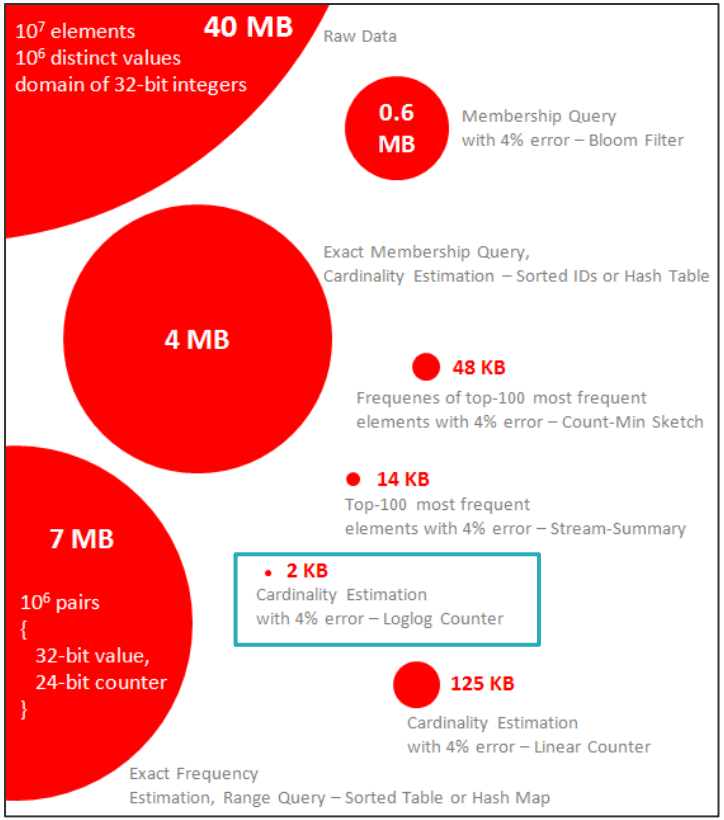
HyperLogLog

How many unique patients have visited this month across all hospitals?



HyperLogLog

- HyperLogLog solves the cardinality estimation problem: How many unique elements are in a large dataset?
- Estimate the number of distinct elements (cardinality) in a massive dataset (e.g., unique IPs), using very little memory.
- Traditional methods (e.g., sets, hash tables) require O(n) memory.
- HyperLogLog estimates the number of unique elements using only O(log log n) space.
- HyperLogLog allows storing one billion elements (e.g., User IDs) using only 1.5 KB of memory and a ~2% error rate!
- Ideal for real-time analytics where speed > exactness.



Common probabilistic data structures with errors and memory requirements


I LOVE
SOFTWARE

HyperLogLog - Estimating the Cardinality

$$\hat{n} = \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

- α_m: A bias-correction constant dependent on m, the number of registers.

$$\alpha_m = \begin{cases} 0.7213/(1 + 1.079/m) & \text{if } m > 16 \\ \text{Precomputed constant} & \text{if } m \leq 16 \end{cases}$$

- m: The number of registers (usually $m = 2^p$, where p is the precision parameter).
- $M[j]$: The value stored in the j-th register, representing the maximum number of leading zeros observed for hash values mapped to that register.

Precision (p) determines how many groups or buckets the algorithm uses to organize the data.

The number of buckets is m=2^p, where p is the precision parameter:

- If p=4, then m = 2^4 = 16 buckets.
- If p=10, then m = 2^10 = 1024 buckets.

Higher precision (larger) means more buckets and better accuracy, but it requires more memory.

$$\alpha_m \approx \begin{cases} 0.673, & \text{for } m = 16; \\ 0.697, & \text{for } m = 32; \\ 0.709, & \text{for } m = 64; \\ \frac{0.7213}{1+1.079/m}, & \text{for } m \geq 128. \end{cases}$$

HyperLogLog - Estimating the Cardinality

$$\hat{n} = \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

- α_m: A bias-correction constant dependent on m, the number of registers.

$$\alpha_m = \begin{cases} 0.7213/(1 + 1.079/m) & \text{if } m > 16 \\ \text{Precomputed constant} & \text{if } m \leq 16 \end{cases}$$

- m: The number of registers (usually $m = 2^p$, where p is the precision parameter).
- $M[j]$: The value stored in the j-th register, representing the maximum number of leading zeros observed for hash values mapped to that register.

Let the precision parameter p = 3. This means that the number of buckets m = 2^p = 8.

We have the following data items: A, B, C, D. Each data item is hashed into a binary value using a hash

Data Item	Hash Value (Binary)
A	000101101011
B	110010110100
C	010011101001
D	100100111010

The first p=3 bits of each hash value determine the bucket index:

Data Item	Hash Value (Binary)	First $p = 3$ Bits	Bucket Index (Decimal)
A	000101101011	000	0
B	110010110100	110	6
C	010011101001	010	2
D	100100111010	100	4

HyperLogLog – Estimating the Cardinality

$$\hat{n} = \alpha_m \cdot m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

- α_m: A bias-correction constant dependent on m, the number of registers.

$$\alpha_m = \begin{cases} 0.7213/(1 + 1.079/m) & \text{if } m > 16 \\ \text{Precomputed constant} & \text{if } m \leq 16 \end{cases}$$

- m: The number of registers (usually $m = 2^p$, where p is the precision parameter).
- $M[j]$: The value stored in the j-th register, representing the maximum number of leading zeros observed for hash values mapped to that register.

Each data item is now assigned to a bucket based on its bucket index:

Bucket Index	Assigned Data Items
0	A
1	(Empty)
2	C
3	(Empty)
4	D
5	(Empty)
6	B
7	(Empty)

For each bucket, consider the remaining bits (after the first p bits) and count the number of leading zeros.

HyperLogLog - Estimating the Cardinality

Data Item	Hash Value (Binary)
A	000101101011
B	1100010110100
C	010011101001
D	100100111010

- For A, remaining bits: 101101011 → Leading zeros = 0.
- For C, remaining bits: 011101001 → Leading zeros = 1.
- For D, remaining bits: 100111010 → Leading zeros = 0.
- For B, remaining bits: 010110100 → Leading zeros = 1.

Data Item	Bucket Index	Leading Zeros	Bucket Value ($M[j]$)
A	0	0	0
C	2	1	1
D	4	0	0
B	6	1	1

The complete bucket array (M) looks like this: M=[0,-,1,-,0,-,1,-], where - represents buckets that remain unused or empty.

The harmonic mean is calculated as:

$$Z = \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

For $M = [0, -, 1, -, 0, -, 1, -]$, substitute the non-empty values:

$$Z = \left(2^{-0} + 2^{-0} + 2^{-1} + 2^{-1}\right)^{-1}$$

1. Compute $2^{-M[j]}$ for the non-empty buckets:
   - $2^{-0} = 1$
   - $2^{-0} = 1$
   - $2^{-1} = 0.5$
   - $2^{-1} = 0.5$
2. Sum these values:

$$\sum_{j=1}^{m} 2^{-M[j]} = 1 + 1 + 0.5 + 0.5 = 3$$

3. Take the inverse:

$$Z = \frac{1}{3} \approx 0.333$$

HyperLogLog – Small Range Correction (Linear Counting)

The estimated cardinality is given by:

$$\hat{n} = \alpha_m \cdot m^2 \cdot Z$$

Where:

- α_m: Bias-correction constant. For $m = 8$, $\alpha_8 \approx 0.673$.
- m: Number of buckets ($m = 8$).
- Z: Harmonic mean ($Z \approx 0.333$).

Substitute the values:

$$\hat{n} = 0.673 \cdot 8^2 \cdot 0.333$$

1. Compute m^2:

$$m^2 = 8^2 = 64$$

2. Multiply:

$$\hat{n} = 0.673 \cdot 64 \cdot 0.333$$

3. Simplify:

$$\hat{n} \approx 14.33$$

Since $\hat{n} \approx 14.33$ and $\hat{n} < 2.5 \cdot m = 2.5 \cdot 8 = 20$, apply the small dataset correction:

$$\hat{n} = m \cdot \log(m/V)$$

Where:

- V: Number of empty buckets. In this case, $V = 4$ (buckets 1, 3, 5, 7 are empty).

Substitute the values:

$$\hat{n} = 8 \cdot \log(8/4)$$

1. Simplify $8/4$:

$$\hat{n} = 8 \cdot \log(2)$$

2. Approximate $\log(2)$:

$$\log(2) \approx 0.693$$

3. Compute:

$$\hat{n} = 8 \cdot 0.693 \approx 5.544$$

The final estimated number of unique elements is approximately:

$$\hat{n} \approx 5.5$$

Initially we have the following data items: A, B, C, D

HyperLogLog – Small Range Correction (Linear Counting)

When the estimated cardinality $\hat{n}$ is small compared to the number of buckets m, the HyperLogLog algorithm may underestimate the number of unique elements. This happens because:

1. In small datasets, many buckets remain **empty** (no hash values mapped to them).
2. Empty buckets indicate that the dataset is small, and the formula needs to account for this.

To correct this, HyperLogLog uses a **small dataset correction formula**.

$$\hat{n} \leq \frac{5}{2} \cdot m$$

$$\hat{n} = m \cdot \log\left(\frac{m}{V}\right)$$

HyperLogLog - Tuning

$$\text{Error} \approx \frac{1.04}{\sqrt{m}}$$

- For $p = 10$ ($m = 1024$), the error is about 3.25%.
- For $p = 14$ ($m = 16,384$), the error drops to about 0.8%.

The total memory usage in bits is:

$$\text{Memory (bits)} = m \cdot b$$

Where:

- $m = 2^p$: The number of buckets, determined by the precision p.
- b: The number of bits used to store the maximum leading zeros for each bucket.

Choose p Based on Desired Accuracy.

Memory Usage vs. Accuracy:
Memory usage grows linearly with m (or exponentially with p).

Trade-off Between p and Dataset Size:

- For small datasets: Use smaller p to avoid excessive memory usage.
- For large datasets: Use larger p to minimize the error rate.

HyperLogLog - Implementation

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/probabilistic/cardinality/HyperLogLog.ts

HyperLogLog - Visualization

http://content.research.neustar.biz/blog/hll.html

Real-World Applications - Amazon Redshift

Feature	Description
Data Type	HLLSKETCH — a native Redshift type that encapsulates the probabilistic sketch data for a given column
Precision Control	Default p = 15 (i.e., m = 2^{15} = 32,768 registers); can be increased up to p = 26 for higher accuracy in small datasets
Relative Error	Ranges from 0.01% (high precision) to 0.6% (default precision); determined by: RSE ≈ 1.04 / sqrt(2^p)
Representations	- Sparse: compact form for small sketches - Dense: automatically selected when sketch size grows beyond sparse threshold
Serialization Format	- Sparse: JSON - Dense: Base64 encoded binary
Mergeable	Yes — multiple sketches (e.g., per product/day/channel) can be merged using HLL_COMBINE()
Preaggregatable	Yes — supports sketch preaggregation and storage for faster analytics and ETL reuse
Query Acceleration	Significant reduction in memory and I/O; e.g., 50s exact query reduced to 22s using HLL with a 0.006% error
Storage Savings	Sketch tables remain at GB scale even with 100+ TB of underlying raw data
Use Cases	Interactive analytics, approximate COUNT(DISTINCT x), multi-channel trend analysis, behavioral metrics at massive scale

Real-World Applications - Redis / RedisBloom

Aspect	Details
Memory Usage	Fixed 12 KB per HLL key (m = 2^14 = 16,384 registers × 6 bits each)
Accuracy (RSE)	RSE ≈ 1.04 / sqrt(m) → ≈ 0.81% for m = 16384
Hashing	64-bit hash: 14 bits for register index, 50 bits for computing ρ(w) (leading zero count)
Cardinality Formula	n̂ = α_m * m^2 / Σ 2^(-M[i])
Small Range Correction	n̂ = m * log(m / V) if n̂ ≤ 2.5 * m
Mergeability	Yes: PFMERGE uses max(M1[i], M2[i], ..., Mn[i]) per register
Bias Correction	4th-degree polynomial regression for n̂ ∈ [40K, 72K]
Commands	PFADD, PFCOUNT, PFMERGE
Data Type	Binary-safe string, endian-neutral, 12296 bytes serialized
Use Cases	Unique visitors, analytics counters

Real-World Applications - Google BigQuery with HyperLogLog++

Feature	Description
Memory Usage	Adaptive: starts small (sparse), grows to ~32 KiB (dense) if needed
Relative Error (RSE)	RSE ≈ 1.04 / sqrt(m) → typically ~0.2% – 0.5%
Estimation Formula	n̂ = α_m * m^2 / Σ 2^(-M[i]), with Google-specific bias correction
Small Range Correction	Improved accuracy using bias tables + sparse representation
Large Range Handling	Accurate beyond 1B+ uniques; reduces overestimation from hash collisions
Mergeability	Yes: HLL_COUNT.MERGE() supports union of distributed sketches
SQL API	APPROX_COUNT_DISTINCT(), HLL_COUNT.INIT(), HLL_COUNT.MERGE()
Performance	3.2B IDs counted in 5.7s (vs 28s for exact), with only 0.2% error
Use Cases	Unique counts over massive datasets (users, sessions, devices, time windows)

HyperLogLog - Limitations

Limitation	Description
Approximate Only	HLL provides *probabilistic estimates* — unsuitable for use cases requiring exact cardinality (e.g., billing).
No Deletion Support	Elements cannot be removed once added. HLL is monotonic by design.
No Element Recovery	HLL is a lossy structure; it stores no actual elements, only statistical summaries.
Overestimation at Large Cardinalities	Without proper hash sizing, standard HLL suffers from hash collisions for $n \gg 2^{32}$ elements.
Underestimation at Small Cardinalities	Standard HLL is biased for $n \ll m$, requiring fallback to linear counting or bias correction.
Limited Merge Semantics	Merging assumes identical hash functions and precision (m); incompatible sketches cannot be merged.
Non-Deterministic Error Bounds	While error is bounded statistically ($RSE \approx 1.04 / \sqrt{m}$), actual error can vary run-to-run.
Fixed Precision (Some Systems)	Redis, for example, fixes $m = 16384$; cannot trade off accuracy vs memory dynamically.
Inefficient at Low Cardinality (Space)	Without sparse encoding (e.g., Redis), sketches always consume full memory even if only a few elements added.
Susceptible to Poor Hashing	Accuracy degrades with poorly distributed hash functions (e.g., skewed inputs or non-uniform hash output).

Use Case: Hospital Data at Scale - Redshift

How many unique patients have visited this month across all hospitals?

```
// Merge Sketches for Monthly Estimate
SELECT
  HLL_CARDINALITY(HLL_COMBINE(sketch)) AS unique_patients
FROM
  hll_patient_sketch
WHERE
  event_date BETWEEN '2025-04-01' AND '2025-04-30';
```

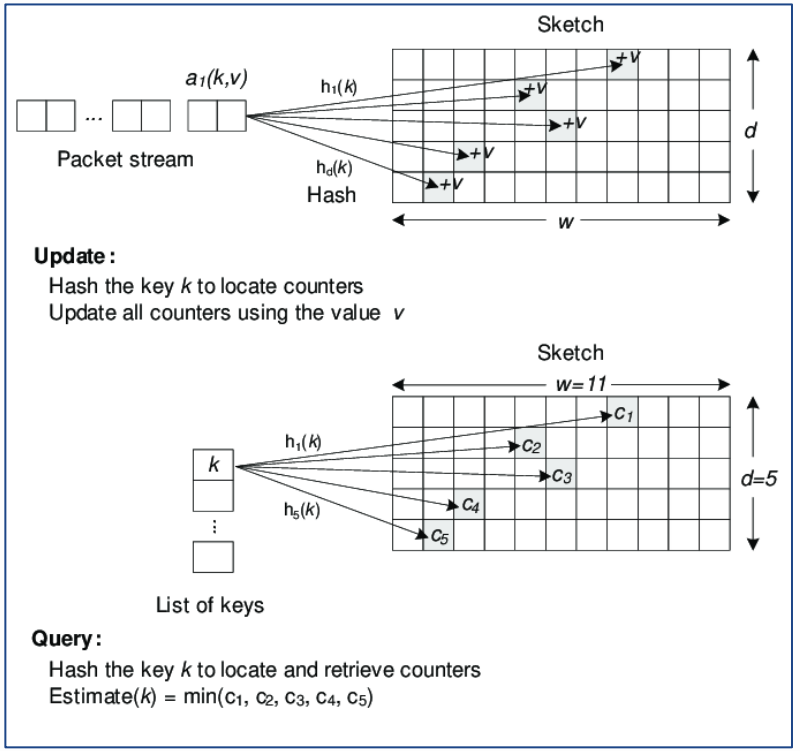
```
// Assumed Table Structure
CREATE TABLE patient_events (
   hospital_id    VARCHAR,
   patient_id     VARCHAR,
   event_time     DATE
);

// Create Daily HLL Sketch Table
CREATE TABLE hll_patient_sketch (
   hospital_id  VARCHAR,
   event_date   DATE,
   sketch       HLLSKETCH
);

// Insert Daily Sketches
INSERT INTO hll_patient_sketch
SELECT
  hospital_id,
  event_time AS event_date,
  HLL_CREATE_SKETCH(patient_id) AS sketch
FROM
  patient_events
WHERE
  event_time BETWEEN '2025-04-01' AND '2025-04-30'
GROUP BY
  hospital_id, event_time;
```

Count-Min Sketch

What are the most frequently prescribed medications over the past 24 hours?

Count-Min Sketch

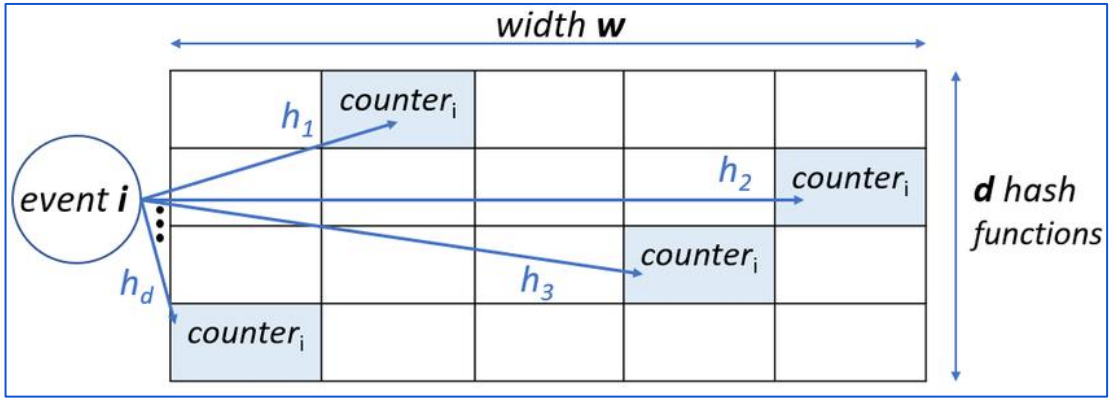
- Count-Min Sketch (CMS) is a probabilistic data structure designed for approximate frequency estimation over streaming or large-scale datasets, using bounded memory and time.
- It supports two main operations:
  - increment(x): Increments the count estimate for item x.
  - estimate(x): Returns an approximate count of how many times x has been seen.
- CMS guarantees:
  - No underestimation.
  - Controlled overestimation bounded by a tunable error ε, with high probability $1-\delta$.



https://www.researchgate.net/figure/Sketch-update-and-query-operations-Count-Min-sketch_fig2_334379141

Count-Min Sketch - Internal Working

- CMS consists of a 2D array of counters (CMS[d][w]) with:
  - d rows, corresponding to independent hash functions.
  - w columns per row (total number of buckets).
- Each row i has its own hash function h_i.
- All counters are initialized to 0.



https://www.researchgate.net/figure/Count-Min-Sketch-mapping-to-different-positions-for-each-row_fig1_324005832

Count-Min Sketch - Insertion

CMS insertion = hashing to positions + incrementing those counters.

Step 1: Hashing to Positions (Projection)

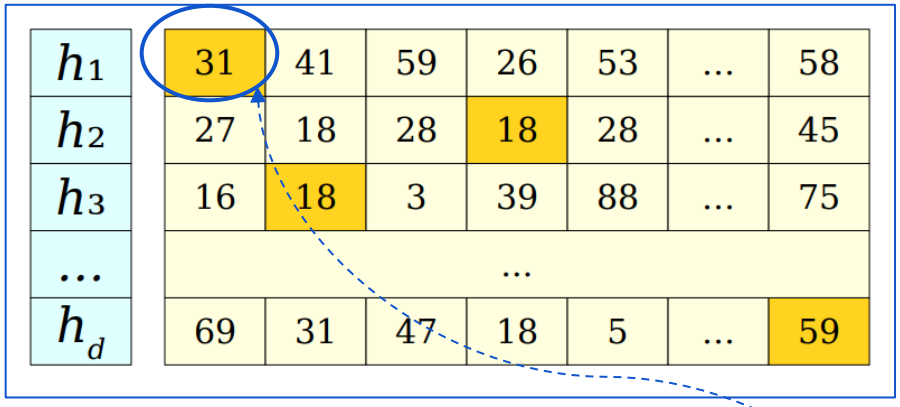
Step 2: Increment Counters

This step is analogous to Bloom filters, where an item is hashed to multiple bit positions.

For each row i, increment the counter at index j_i.

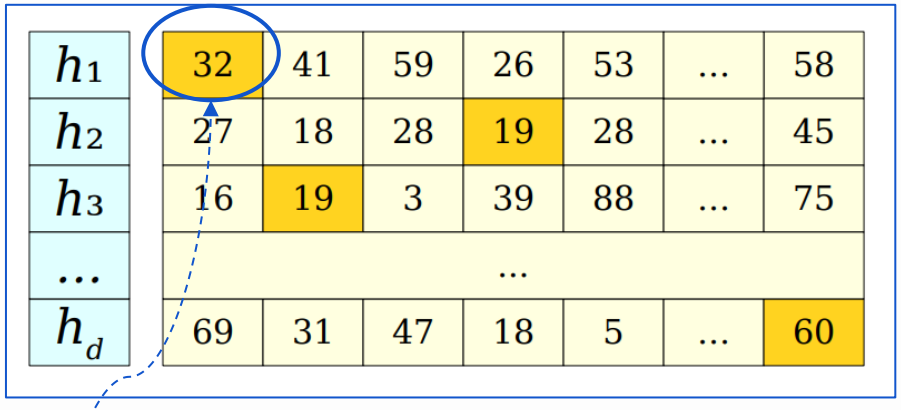
Count-Min Sketch - Increment

Before



h_1	31	41	59	26	53	...	58
h_2	27	18	28	18	28	...	45
h_3	16	18	3	39	88	...	75
...				...			
h_d	69	31	47	18	5	...	59

After



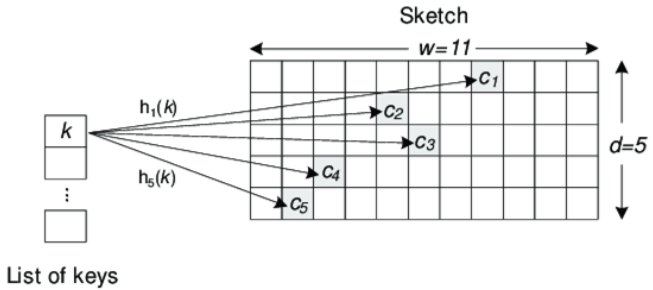
h_1	32	41	59	26	53	...	58
h_2	27	18	28	19	28	...	45
h_3	16	19	3	39	88	...	75
...				...			
h_d	69	31	47	18	5	...	60

```
increment(x):
  for i in range(1, d + 1):
     j = h_i(x) % w
     CMS[i][j] += 1
```

The CMS does not know which elements x were inserted. It only stores counters at positions determined by hash functions. It never explicitly stores x.

Each hash function h_i maps x to an index j in row i, and the corresponding counter is incremented.

Count-Min Sketch - Estimate



Query:

Hash the key *k* to locate and retrieve counters

Estimate(*k*) = min(c_1, c_2, c_3, c_4, c_5)

h_1	32	41	59	26	53	...	58
h_2	27	18	28	19	28	...	45
h_3	16	19	3	39	88	...	75
...	...						
h_d	69	31	47	18	5	...	60

```
estimate(x):
    result = ∞
    for i in range(1, d + 1):
        j = h_i(x) % w
        result = min(result, CMS[i][j])
    return result
```

f(x) = min_i { CMS[i][h_i(x)] }
= min(32, 19, 19, 60) = 19

This means CMS estimates that x has appeared at least 19 times.

Taking the minimum helps reduce the overestimation caused by hash collisions.

Count-Min Sketch - Error Bounds

To achieve:

$$P[\ \hat{f}(x) \leq f(x) + \varepsilon \cdot N\ ] \geq 1 - \delta$$

We set the matrix dimensions as:

$$w = \text{ceil}(e / \varepsilon)$$

$$d = \text{ceil}(\ln(1 / \delta))$$

- Width (w) controls the error magnitude:
  - More buckets → less chance of collision → smaller overestimation
  - Directly proportional to $1/\varepsilon$
- Depth (d) controls the confidence level:
  - More rows → more independent estimates → lower failure probability
  - Logarithmic in $1/\delta$

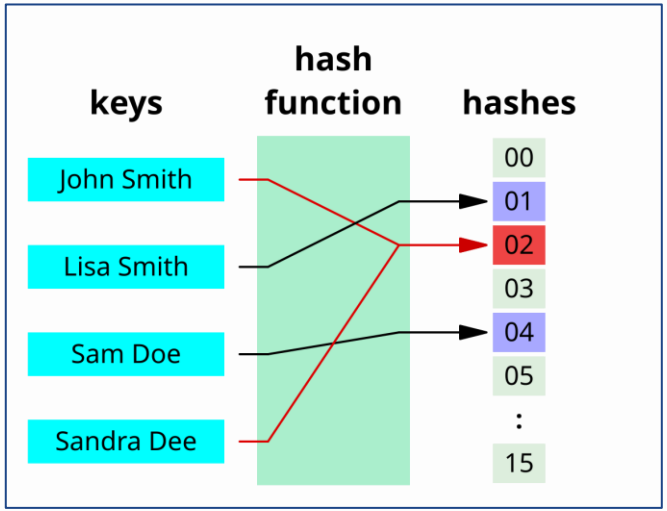
```
ε = 0.05      # Tolerate up to 5% overestimation
δ = 0.01      # Accept 1% probability of violating the error bound
w = ceil(e / 0.05) = ceil(54.598...) = 55
d = ceil(ln(1 / 0.01)) = ceil(ln(100)) = ceil(4.605...) = 5

// This yields a sketch with dimensions 5 × 55, capable of estimating any frequency with:
f^(x) ≤ f(x) + 0.05 · N    with probability ≥ 0.99
```

Count-Min Sketch - Collision

Why Collisions Occur ?

- Fixed width (w): The number of available counters per hash function is finite.
- More unique elements than buckets: By the pigeonhole principle, multiple elements will hash to the same index: If w = 4 and there are 5 unique keys, at least one collision occurs.
- Hash functions are deterministic: The same input always maps to the same index, so elements with similar hash patterns may collide.
- Counters are shared: A single bucket stores the aggregate of all elements mapped to it — it cannot separate them.

Collisions lead to overestimation



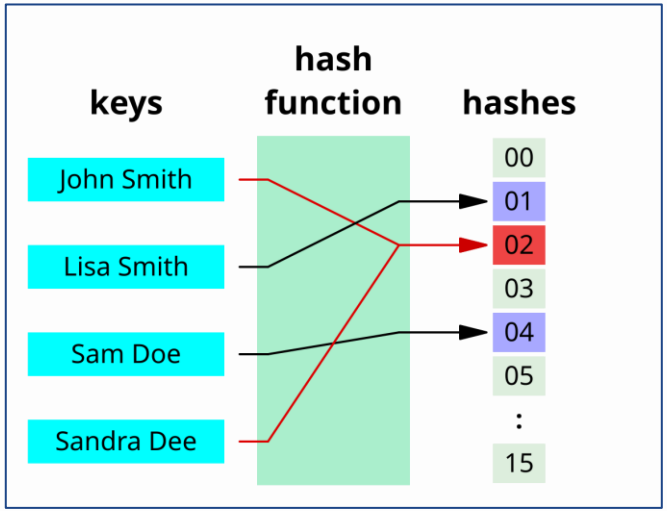
https://en.wikipedia.org/wiki/Hash_collision

Count-Min Sketch - Collision

How to Mitigate Collisions?

1. Increase width (w): Reduces collision probability by spreading keys over more buckets.
2. Increase depth (d): More hash functions reduce the impact of any single collision (via min).
3. Use good hash functions: Choose functions with uniform distribution (e.g., MurmurHash, xxHash).
4. Dynamically scale w and d: Adapt to data size and cardinality to maintain error bounds.
5. Track heavy hitters separately: Use CMS for filtering, and exact counters for top-k candidates.
6. Partition by domain: Isolate high-skew items to dedicated sketches to reduce pollution.

Collisions lead to overestimation



https://en.wikipedia.org/wiki/Hash_collision

Count-Min Sketch - Implementation

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/probabilistic/frequency/CountMinSketch.ts

Count-Min Sketch - Visualization

https://florian.github.io/count-min-sketch/

Real-World Applications - Redis / RedisBloom

Aspect	Details
Core Commands	CMS.INITBYPROB, CMS.INCRBY, CMS.QUERY, CMS.INFO, CMS.MERGE
Initialization	CMS.INITBYPROB key error probability or CMS.INITBYDIM key width depth
Parameters	error: additive error bound (ε), probability: failure probability (δ)
Size Determination	Width w = ceil(e / ε), Depth d = ceil(ln(1 / δ))
Estimate Accuracy	Guarantees f̂(x) ≤ f(x) + ε · N with probability ≥ 1 − δ
Threshold for Reliability	threshold = error × total_count — ignore results below this (considered noise)
Collision Handling	Accepts collisions; estimation uses min(count[i][h_i(x)]) to reduce overestimation
Mergeable	Yes — use CMS.MERGE to combine compatible sketches

Count-Min Sketch - Limitations

Limitation	Description
Overestimation Bias	CMS always overestimates due to hash collisions; never underestimates
No Key Storage	Does not store actual elements — only hashed counts; cannot list frequent items
Poor for Low Frequencies	Estimates below $\varepsilon \times$ total_count are unreliable and treated as noise
Initialization Required	Must be initialized with error/confidence parameters before use
Parameter Sensitivity	Incorrect choice of ε and δ leads to poor accuracy or excessive memory use
No Deletions (basic form)	Standard CMS cannot decrement counts; requires advanced variants
Hash Function Dependence	Accuracy depends on quality and independence of hash functions
Cannot Distinguish Collisions	Different elements sharing counters are indistinguishable

Use Case: Hospital Data at Scale - Redis / RedisBloom

What are the most frequently prescribed medications over the past 24 hours?

```
#!/bin/bash

# --- Config ---
ERROR=0.001
PROBABILITY=0.001
KNOWN_MEDICATIONS=("Ibuprofen" "Paracetamol" "Amoxicillin" "Aspirin" "Metformin"
"Omeprazole" "Atorvastatin")

# --- Step 1: Initialize CMS sketches for each of the past 24 hours ---
for i in {0..23}; do
   HOUR=$(date -u -d "$i hour ago" +"%Y-%m-%dT%H")
redis-cli CMS.INITBYPROB cms:medications:$HOUR $ERROR $PROBABILITY > /dev/null
done

# --- Step 2: Simulate increments for current hour (for demo) ---
   NOW_HOUR=$(date -u +"%Y-%m-%dT%H")
for MED in "${KNOWN_MEDICATIONS[@]}"; do
   COUNT=$((RANDOM % 100 + 1))
redis-cli CMS.INCRBY cms:medications:$NOW_HOUR "$MED" $COUNT > /dev/null
done
```

Use Case: Hospital Data at Scale - Redis / RedisBloom

What are the most frequently prescribed medications over the past 24 hours?

```
# --- Step 3: Aggregate counts across 24 hours ---
declare -A MED_COUNTS
TOTAL_INSERTS=0

for i in {0..23}; do
   HOUR=$(date -u -d "$i hour ago" +"%Y-%m-%dT%H")

# Query frequencies
QUERY_RESULT=$(redis-cli CMS.QUERY cms:medications:$HOUR "${KNOWN_MEDICATIONS[@]}" | xargs)
I=0
for MED in "${KNOWN_MEDICATIONS[@]}"; do
   COUNT=$(echo $QUERY_RESULT | cut -d ' ' -f $((I + 1)))
MED_COUNTS[$MED]=$((MED_COUNTS[$MED] + COUNT))
((I++))
done

# Get total inserts (for threshold)
   COUNT=$(redis-cli CMS.INFO cms:medications:$HOUR 2>/dev/null | awk 'NR==6 {print $2}')
TOTAL_INSERTS=$((TOTAL_INSERTS + COUNT))
done
```

Use Case: Hospital Data at Scale - Redis / RedisBloom

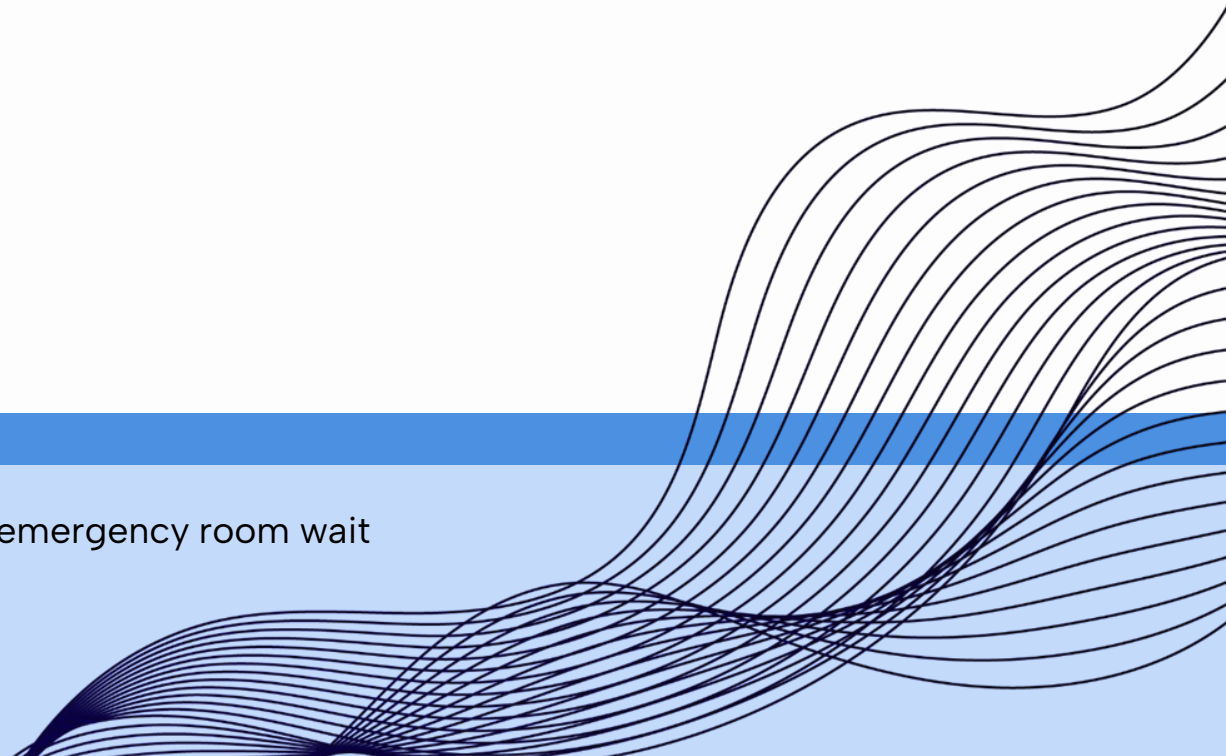
What are the most frequently prescribed medications over the past 24 hours?

```
# --- Step 4: Filter and Print ---
THRESHOLD=$(echo "$ERROR * $TOTAL_INSERTS" | bc | awk '{print int($1+0.5)}')

echo ""
echo "📊 Top Prescribed Medications (Past 24 Hours)"
echo "Threshold (ε · total) = $THRESHOLD"
echo "------------------------------------------------"
for MED in "${KNOWN_MEDICATIONS[@]}"; do
   if [[ ${MED_COUNTS[$MED]} -ge $THRESHOLD ]]; then
printf "%-20s → %d\n" "$MED" "${MED_COUNTS[$MED]}"
fi
done
```

Top prescribed medications over the past 24h:
1. Paracetamol => 1623
2. Ibuprofen => 1455
3. Amoxicillin => 1087
...

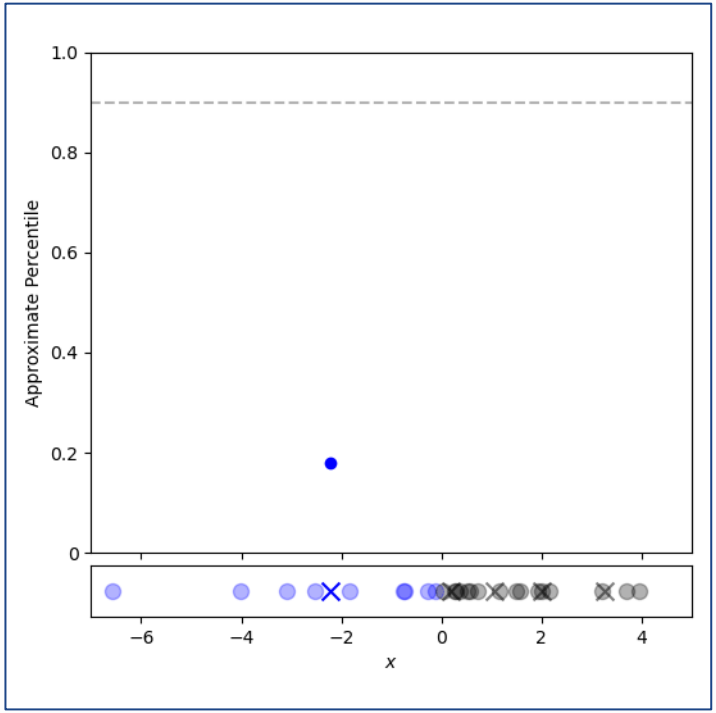
t-digest

What is the 95th percentile of emergency room wait times today?



t-digest

- t-digest is a data structure designed for efficient quantile approximation on large-scale or streaming datasets.
- It achieves high accuracy with low memory usage, avoiding the costs of full sorting or storing raw data.
- Supports incremental updates, making it well-suited for real-time analytics.
- Enables parallel and distributed computation by merging multiple digests without loss of significant accuracy.
- Widely used in monitoring systems, anomaly detection, and distributed processing frameworks.



Estimating the 90th percentile is done by iterating through the t-digest centroids until the weight fraction has exceeded 90%

t-digest - Algorithm

1) Initial Centroid Creation: The t-digest starts by creating initial centroids, often representing the first few data points individually.

2) Data Compression with Centroids: T-Digest groups similar values into centroids, where:

- Each centroid represents a cluster of values.
- Each centroid has:
  - A mean: The center of the cluster.
  - A weight: The number of values it represents.

3) Merge Centroids: As new data arrives, it is considered for merging into existing centroids.

- If a new value is considered 'close enough' to an existing centroid (based on a defined scale), it is added to that centroid, updating its mean and weight.
- If a value is not close enough to any existing centroid, a new centroid may be created.

Step 1: Insert 100

```
(100, weight=1)
```

Step 2: Insert 120

```
(100, weight=1)
(120, weight=1)
```

Step 3: Insert 110 (merged with 100)

```
(105, weight=2)
(120, weight=1)
```

Step 4: Insert 500 (new centroid)

```
(105, weight=2)
(120, weight=1)
(500, weight=1)
```

Step 5: Insert 115 (merged with 120)

```
(105, weight=2)
(117.5, weight=2)
(500, weight=1)
```

Step 6: Insert 102, 104, 106, 108, 109 (merged into 105)

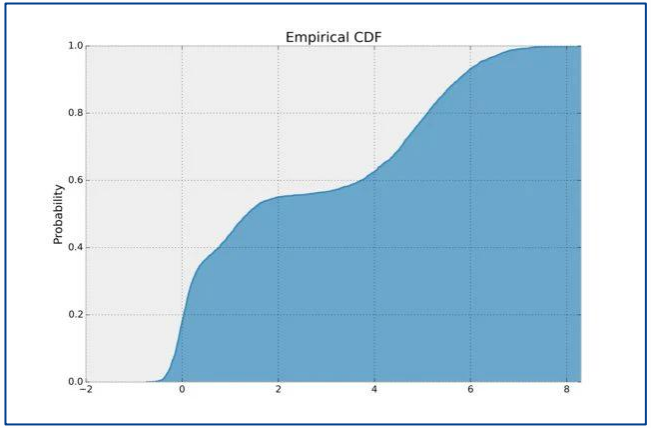
```
(106, weight=7)
(117.5, weight=2)
(500, weight=1)
```

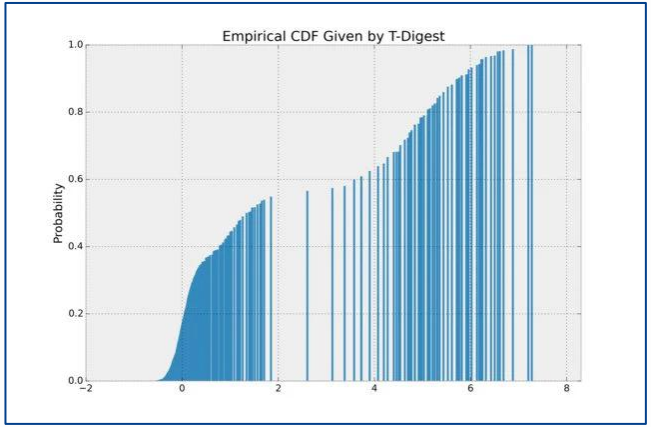
t-digest - Algorithm

4) Compression Algorithm: T-Digest employs a compression algorithm to ensure that:

- Centroids near the tails of the distribution (e.g., very small or very large values) are small and precise.
- Centroids near the middle (where values are dense) are larger and coarser.
- Periodically, this algorithm merges nearby centroids to maintain a bounded number of centroids while preserving quantile accuracy.

This adaptive resolution allows T-Digest to maintain higher precision at the tails, which is critical for accurate quantile estimation.





https://dataorigami.net/2015/03/19/Percentile-and-Quantile-Estimation-of-Big-Data-The-t-Digest.html

t-digest - Configuration

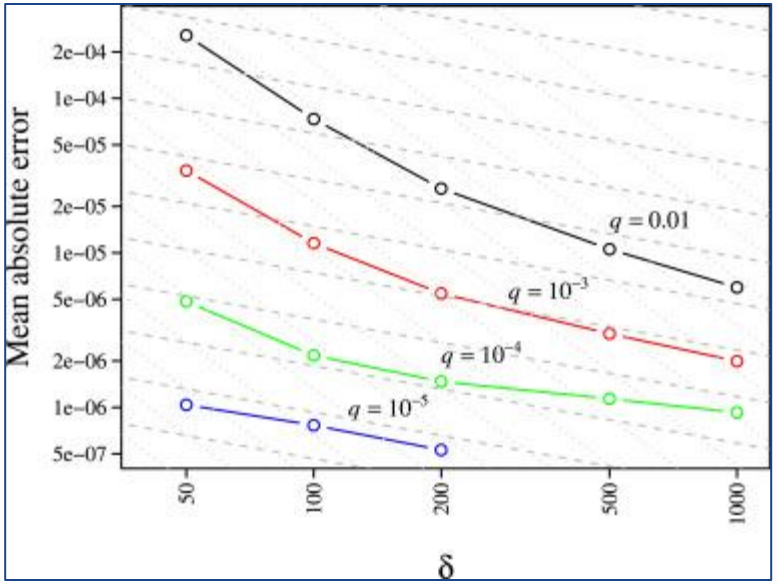
Compression Parameter (δ):

- Controls: Maximum number of centroids (memory usage).
- Higher **δ**:
  - More centroids → Higher accuracy (especially tails) → More memory.
- Lower **δ**:
  - Fewer centroids → Lower accuracy → Less memory.
- Approximate Centroids (C):
  - C = **δ** * log(N)
    - C: Number of centroids.
    - δ: Compression parameter.
    - N: Total number of data points.

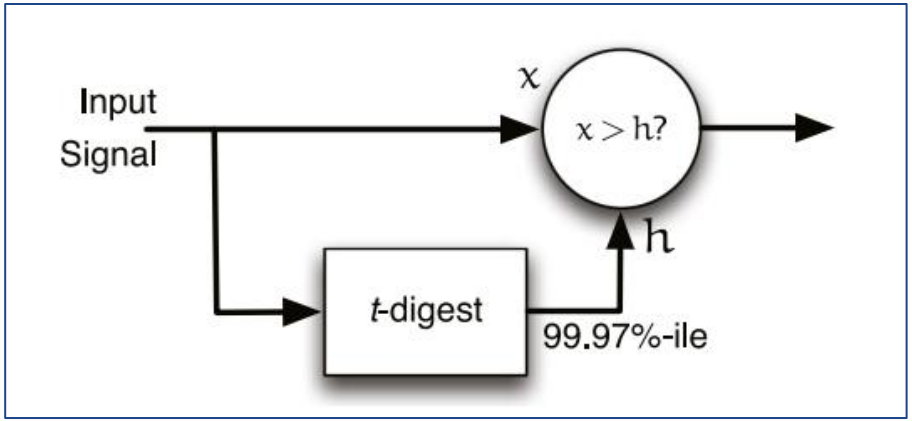
Scaling for Tail Accuracy:

- Goal: High accuracy for extreme percentiles (tails).
- Scaling Function (S(q)):
  - S(q) = **δ** * min(q, 1 - q)
    - q: Quantile (0 to 1).
    - min(q, 1 - q): Smallest as we approach the tails (0 or 1).
- Interpretation:
  - Tails (q near 0 or 1):
    - S(q) is small → Limits cumulative weight → Forces smaller, more precise centroids.
  - Median (q near 0.5):
    - S(q) is larger → Allows larger, coarser centroids.

t-digest - Configuration



The scaling of quantile estimation for various values of compression factor δ and q for 106 samples



Users can set thresholds in terms of score quantiles instead of scores by using a t-digest.

t-digest - Formula for Quantile Estimation

Quantile Estimation: Finding Values at Percentiles

- Goal: Estimate the value at a given quantile (q).
- Target Rank: Target Weight = q * N (N = total data points).
- Find Enclosing Centroids: Sum centroid weights until the cumulative weight just passes the Target Weight.
- Estimate Value: Interpolate linearly between the means of the centroids just below and above the Target Weight.

Example: Estimating the 95th Percentile

1. Target Weight = 0.95 * N
2. Sum centroid weights until crossing Target Weight.
3. The 95th percentile value is approximately between the means of the two surrounding centroids.

t-digest - Implementation

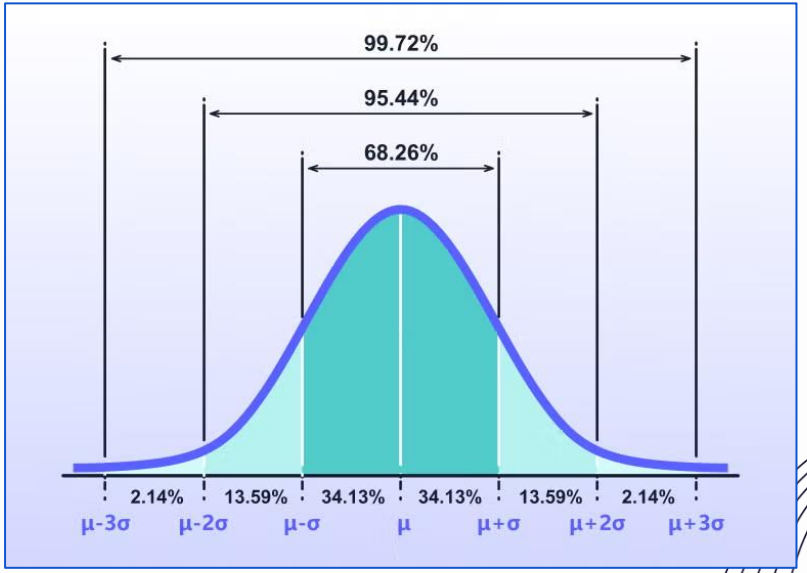
https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/probabilistic/quantile/TDigest.ts

t-digest - Visualization

https://www.gresearch.com/news/approximate-percentiles-with-t-digests/

Real-World Applications

- Apache Druid: Uses TDigestSketch for fast, approximate quantile aggregations over large OLAP datasets, enabling percentile estimation without needing raw data during queries.
- Elasticsearch: Supports percentiles and percentile_ranks aggregations using t-digest under the hood, ideal for monitoring and observability use cases where accurate tail latencies are key.
- Redis: Implements t-digest in its probabilistic data types module, allowing percentile and trimmed mean estimation with sublinear memory in streaming contexts.



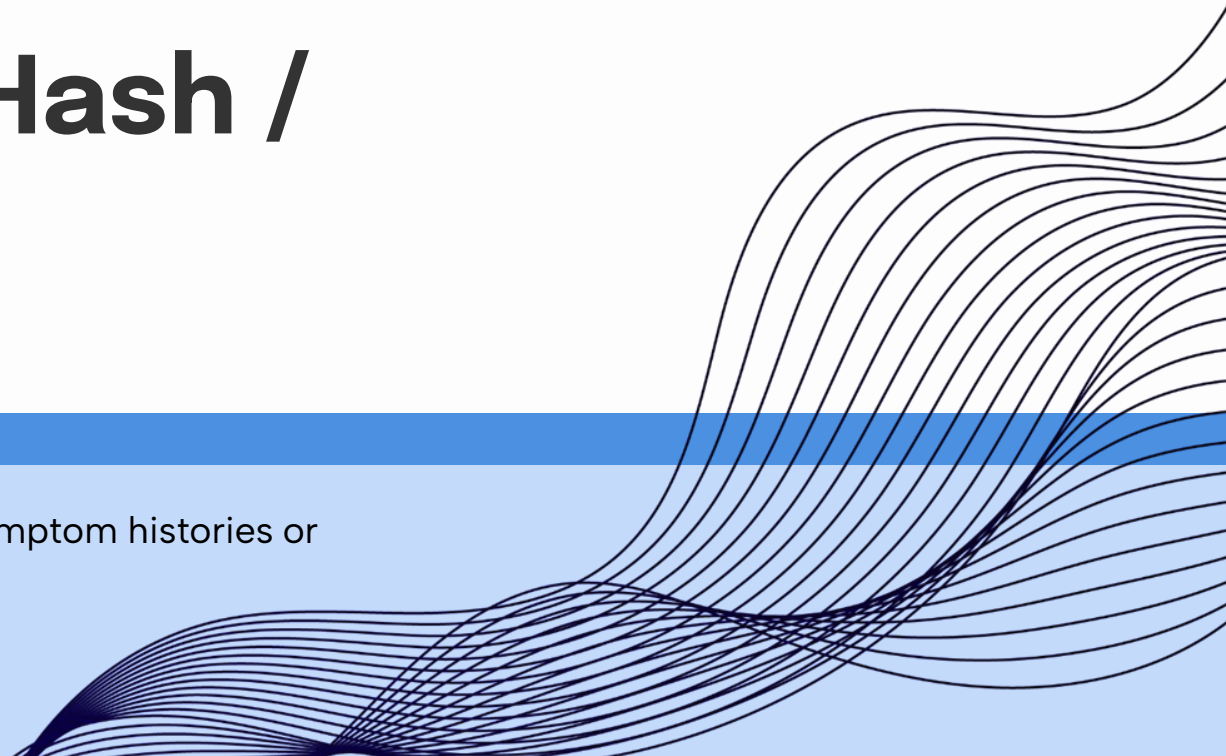
https://redis.io/blog/t-digest-in-redis-stack/

t-digest - Limitations

Category	Limitation	Details
Accuracy	No deterministic error bounds	Accuracy is empirical, not guaranteed; tail quantiles are more precise than the center.
	Interpolation may introduce bias	Especially in sparse regions with few centroids.
Memory	Accuracy–memory trade-off controlled by compression factor (δ)	Higher precision requires more memory (more centroids).
	Memory usage can grow with data distribution complexity	Heavy tails or multimodal distributions may require more centroids to maintain accuracy.
Merge Behavior	Merge order can affect results	Merging digests in different orders can yield slightly different approximations.
Use Case Boundaries	Not ideal for exact quantile computation	Use only where approximations are acceptable (e.g., p99, p95, SLA analysis).
Streaming Precision	Early centroids may dominate if compression isn't triggered frequently	Requires good centroid management or adaptive thresholds for true streaming applications.

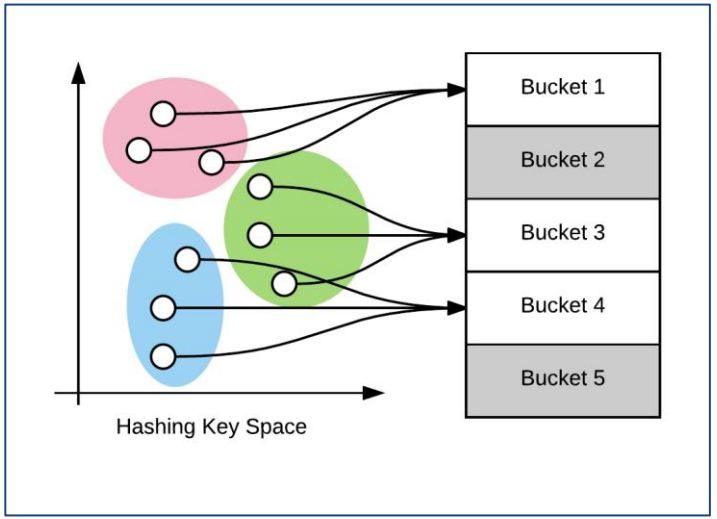
LSH / MinHash / SimHash

Which patients have similar symptom histories or diagnoses?



LSH

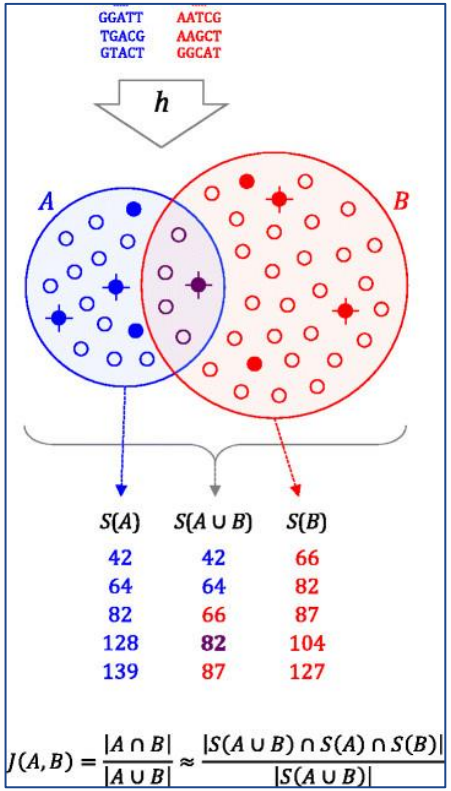
- The Challenge: Finding Similar Items in Large Datasets
  - Comparing all pairs of items (documents, images, etc.) to find similar ones becomes computationally infeasible in large datasets.
- Locality Sensitive Hashing (LSH): The Solution
  - Reduces Complexity: LSH employs hash functions to group similar items into the same "buckets" with a high probability.
  - Embraces Collisions: Unlike traditional hashing (which avoids collisions), LSH uses collisions as a strong indicator of potential similarity between items.
  - General Framework: LSH is not a single algorithm but a framework that utilizes specific locality-sensitive hash functions, such as MinHash and SimHash.



Hashing of similar data points to the same buckets

MinHash - Similarity of Sets (Jaccard Similarity)

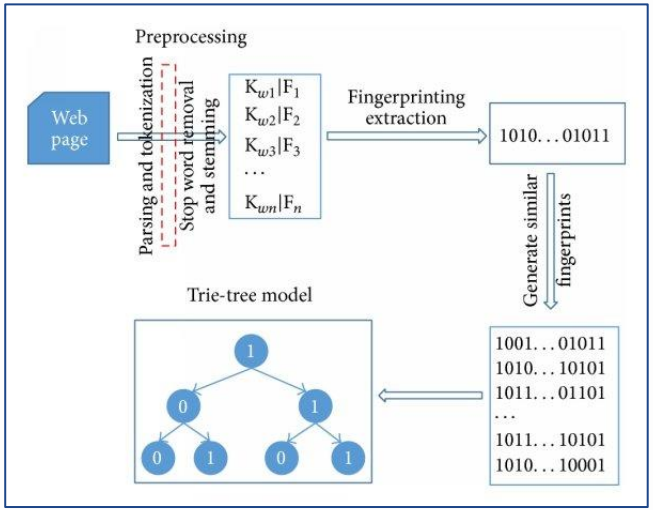
- Focus: Estimates the Jaccard Similarity between sets (size of intersection / size of union).
- Used In: Efficiently determine similarity between large sets in applications like:
  - Document and Web Page Deduplication: Identify and filter out duplicate or near-duplicate pages in search engine indexing.
  - Social Media and News Aggregation: Group similar content or articles.
  - Collaborative Filtering: Cluster users or items with overlapping features, such as shared preferences.
  - Genomic Analysis: Detect similarities in DNA sequences by treating them as sets of k-mers (small sequences).
- Mechanism: Creates a small, fixed-size "MinHash signature" for each set.



Overview of the MinHash bottom sketch strategy for estimating the Jaccard index

SimHash - Similarity of High-Dimensional Vectors

- Focus: SimHash is an LSH technique for cosine similarity between vectors, especially good for text and high-dimensional sparse data.
- Used In: Efficiently detect near-duplicates in large-scale systems, such as:
  - Search Engines: Detect and eliminate duplicate content in web crawling.
  - Social Media & News Aggregation: Group similar articles or posts for streamlined feeds.
  - Plagiarism Detection: Identify copied or slightly altered content.
- Mechanism: Generates a compact, fixed-length binary "SimHash signature" for each vector.



Near-duplicate document detection in the training phase


SOFTWARE
IS A WONDERFUL
WORLD

Example: Detecting Near-Duplicate Documents with MinHash

Documents:
Doc A: "The quick brown fox jumps over the lazy dog"
Doc B: "The quick brown fox leaps over the lazy hound"

Step 1: Shingling (3-word shingles)

Doc A:["The quick brown", "quick brown fox", "brown fox jumps",
"fox jumps over", "jumps over the", "over the lazy",
"the lazy dog"]

Doc B:["The quick brown", "quick brown fox", "brown fox leaps",
"fox leaps over", "leaps over the", "over the lazy",
"the lazy hound"]

Step 2: Set Representation
Set A = { s_1, s_2, ..., s_7 }
Set B = { t_1, t_2, ..., t_7 }

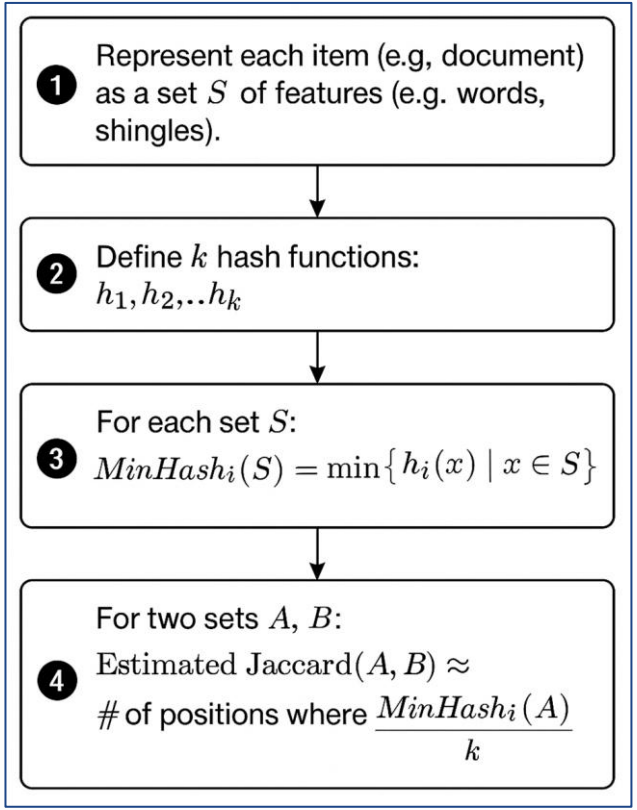
Step 3: Apply MinHash (using 3 example hash functions; precomputed hash values)

Shingle	$h_1(x)$	$h_2(x)$	$h_3(x)$
"The quick brown"	12	57	23
"quick brown fox"	33	19	41
"brown fox jumps"	55	67	39

MinHash_A = [12, 19, 23]
MinHash_B = [12, 19, 23]

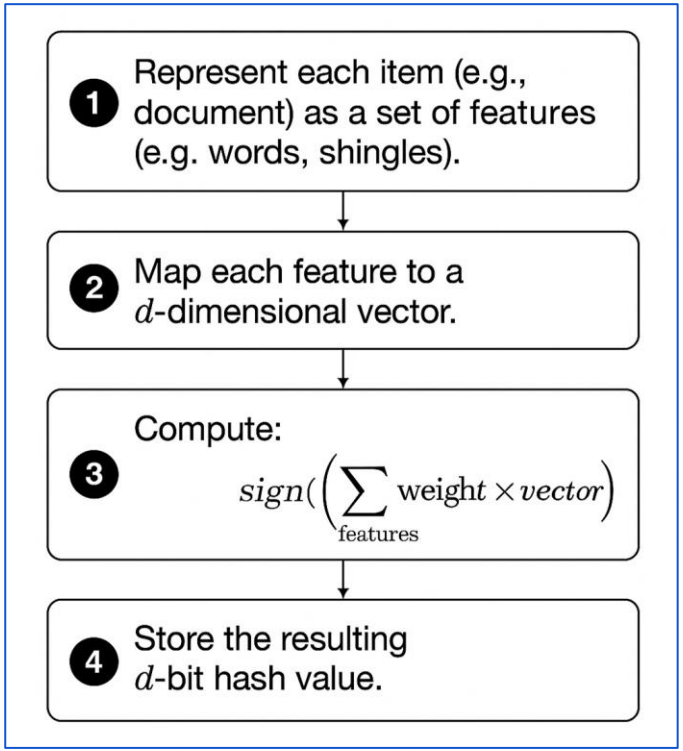
Step 4: Estimate Jaccard Similarity
Matching positions = 3 / 3
→ Estimated_Jaccard(A, B) = 1.0

MinHash - In Action



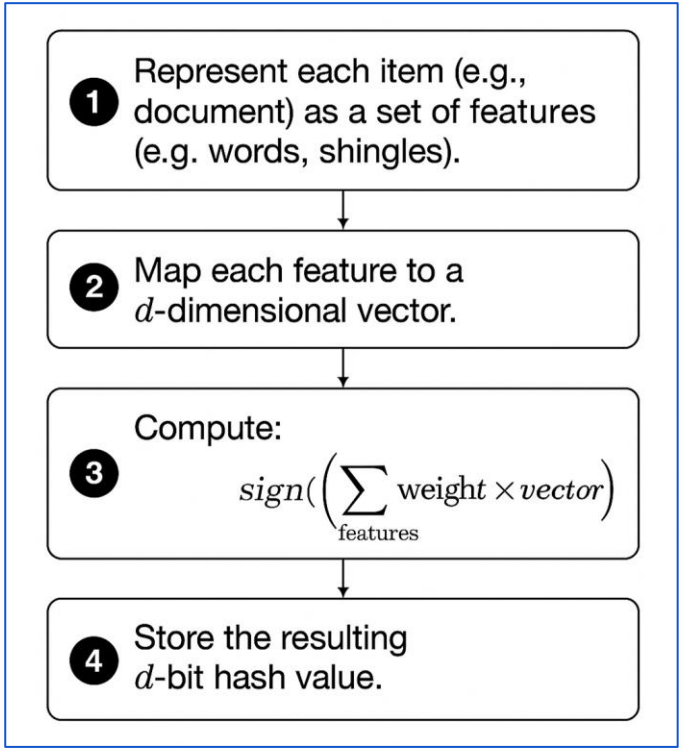
SimHash - In Action

```
Example: SimHash for Text Similarity

Documents:

Doc A: "The quick brown fox jumps over the lazy dog"
Doc B: "A quick brown dog jumps over the lazy fox"

Step 1: Tokenization and Weighting (e.g., TF-IDF or just frequency)

Features for Doc A:
"quick" → 1
"brown" → 1
"fox"   → 1
"jumps" → 1
"over"  → 1
"lazy"  → 1
"dog"   → 1
"the"   → 1

Step 2: Hash each feature into a fixed-length bit vector (Assume 4-bit
hashes for simplicity)

Example hash outputs (fictional):
"quick" →  1 0 1 1
"brown" →  1 1 1 0
"fox"   →  1 1 0 1
"jumps" →  0 1 1 1
"over"  →  1 0 0 1
"lazy"  →  0 1 0 0
"dog"   →  1 0 1 0
"the"   →  0 1 1 0
```



SimHash - In Action

```
Step 3: Weighted sum of bits (positive for 1, negative for 0)

Bit position sums:
pos 0: (+1 +1 +1 -1 +1 -1 +1 -1) = +2
pos 1: (-1 +1 +1 +1 -1 +1 -1 +1) = +2
pos 2: (+1 +1 -1 +1 -1 -1 +1 +1) = +2
pos 3: (+1 -1 +1 +1 +1 -1 -1 -1) = +0

Step 4: Take the sign of each position to form SimHash
 → SimHash_A = [1, 1, 1, 0]  (interpreted as 1110)

Repeat for Doc B:
   → SimHash_B = [1, 1, 1, 1]  (interpreted as 1111)

Step 5: Compare hashes using Hamming distance
Hamming(SimHash_A, SimHash_B) = 1 bit difference

Conclusion:
   → Very similar documents (only 1-bit difference out of 4)
```



LSH / MinHash / SimHash - Implementation

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/probabilistic/similarity/SimHash.ts

https://github.com/helabenkhalfallah/dsa-toolbox/blob/main/src/data-structures/probabilistic/similarity/MinHash.ts

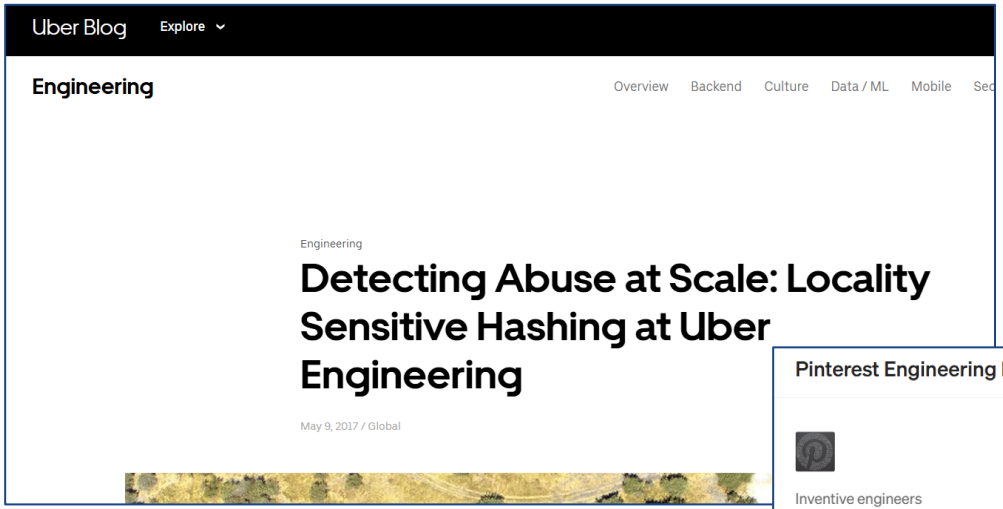
MinHash - Limitations

Aspect	Limitation
Data Type Restriction	Works only for sets—not suitable for vectors or weighted features
Accuracy	Needs many hash functions to reduce variance and improve accuracy
Random Permutations	True MinHash requires simulating random permutations (costly in practice)
Space Efficiency	Signature vectors can still be large (e.g., 100–200 integers per set)
No Weight Awareness	Cannot account for term frequency or importance (all items are equal)

SimHash - Limitations

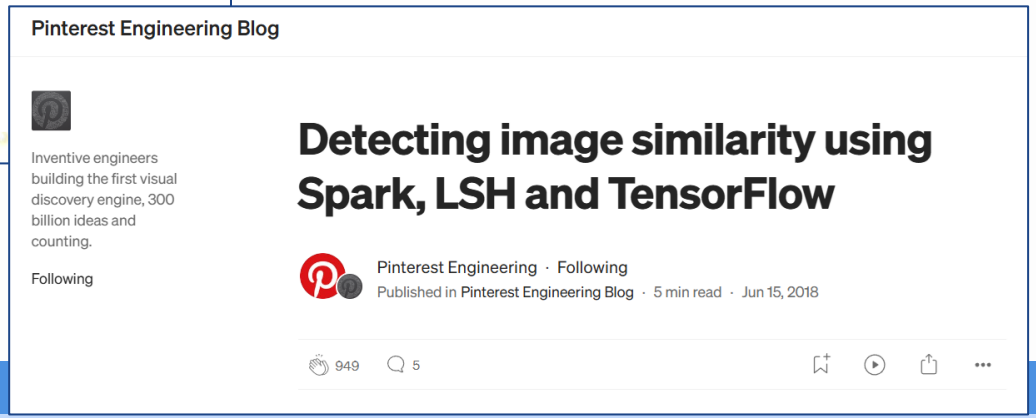
Aspect	Limitation
Sensitivity to Weights	Minor weight changes can cause large differences in hash output
Semantic Limitations	Bit-wise similarity does not capture meaning or context
Precision	Less accurate than MinHash for strict overlap-based similarity
Hash Collisions	More prone to collisions in high-dimensional or dense data
Interpretation Difficulty	Hamming distance is abstract and not always intuitive for human understanding

Real-World Applications



https://www.uber.com/en-FR/blog/lsh/

https://medium.com/pinterest-engineering/detecting-image-similarity-using-spark-lsh-and-tensorflow-618636afc939



It's time to recap!

Quick Reference

Problem	Traditional Solution	Probabilistic Structure	Why It Wins
Membership testing	Hash sets, B-trees	Bloom Filter	Fixed memory, fast lookups, no false negatives
Unique counts	Sets, COUNT(DISTINCT)	HyperLogLog	Sublinear space, mergeable, ~1% error
Frequent elements	Counters, top-k queries	Count-Min Sketch	Fast inserts, low memory, bounded overestimation
Percentiles / quantiles	Full sort, histogram	t-digest	Real-time quantile approximation, mergeable
Similarity (sets)	Jaccard comparisons	MinHash	Tiny signatures, fast similarity estimation
Similarity (vectors / text)	Cosine distance, embeddings	SimHash	Compact fingerprints, fast Hamming comparison
Scalable similarity search	Brute-force comparison	LSH (framework)	Bucket-based filtering, sublinear candidate scan

04

Conclusion, Takeaways & Resources

Summarizing the core insights, when and how to use these structures, and resources to explore them further.

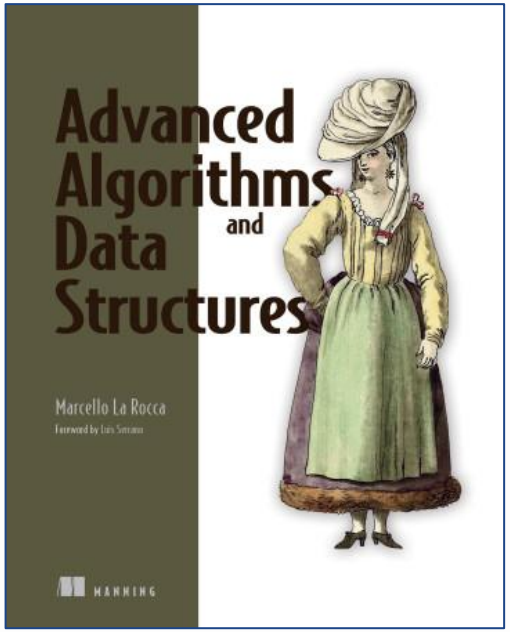
Key Takeaways

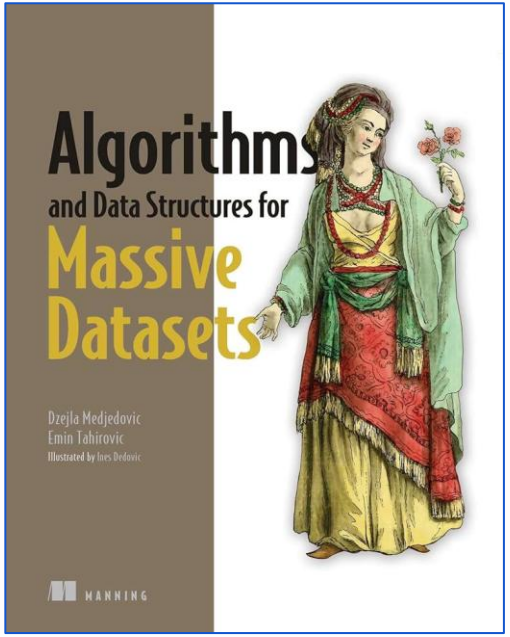
- Probabilistic ≠ Inaccurate: These structures trade exactness for speed and space, offering high-confidence approximations with massive performance gains.
- Built for Scale: Probabilistic data structures are designed for big data: they work efficiently on streaming data, in distributed systems, and with strict memory constraints.
- Mergeability Matters: Many structures (e.g., HyperLogLog, t-digest) are mergeable, making them ideal for parallel or distributed computation.
- Understand the Trade-offs: Choose the structure based on:
  - Query type (membership? frequency? similarity?)
  - Error tolerance
  - Memory budget
- Widely Integrated: These techniques are embedded in systems like:
  - Redis (Bloom Filter, HyperLogLog)
  - PostgreSQL (Bloom index)
  - Spark (HyperLogLog, MinHash, Count-Min Sketch)
  - Druid, Elasticsearch, BigQuery (t-digest, HLL)

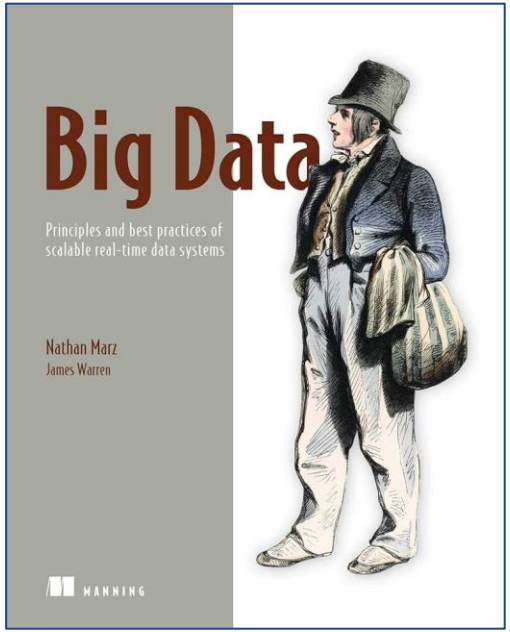
Best Practices

Situation	Recommended Practice
Estimate cardinality (e.g., unique users)	Use HyperLogLog with controlled error rate (1–2%)
Prevent duplicates under memory limits	Use a Bloom Filter; tune k and m to control false positives
Track most frequent items in a stream	Use Count-Min Sketch; combine with heavy hitter detection
Estimate quantiles/percentiles in real-time	Use t-digest; keep centroids sorted for merging
Detect duplicates or near-duplicates	Use MinHash for sets, SimHash for text or vectors
Fast nearest neighbor search in high dimensions	Use LSH with an appropriate hash family (e.g., MinHash, SimHash)
Deploying at scale	Ensure structures are mergeable, and implement bounded memory tracking

Recommended Books & Papers







Cheat Sheets

Structure	Best For	Memory	Accuracy	Mergeable	Typical Use
Bloom Filter	Membership test	Fixed (bit array)	False positives only	❌ Standard Bloom Filter ✅ Counting Bloom Filter	Redis, Cassandra, Spark
HyperLogLog	Cardinality estimation (COUNT DISTINCT)	~1.5 KB	~0.81–2% (tunable via p)	✅	Redis, BigQuery, Redshift
Count-Min	Frequency counts in streams	Tunable (ε, δ)	Overestimation only	✅	Redis, top-k in telemetry/analytics
t-digest	Quantile estimation (P95, P99, etc.)	O(log N)	High accuracy, especially in tails	✅	Druid, Elasticsearch, Redis
MinHash	Jaccard similarity (sets)	O(k)	Controlled by number of hash functions	✅	Document deduplication, clustering
SimHash	Cosine similarity (vectors, text)	Fixed (e.g., 64b)	Sensitive to weight and collisions	✅	Search, deduplication, fingerprinting
LSH (meta)	Fast approximate similarity search	Multiple tables	Depends on hash family & tuning	✅	Fast nearest-neighbor, clustering

Thanks!

Do you have any questions?
helabenkhalfallah.com
helabenkhalfallah.medium.com
@b_k_hela
www.meetup.com/code-craft-community