



**Welcome to**  
**Elastic Bangalore User Group!**

# Who Am I?



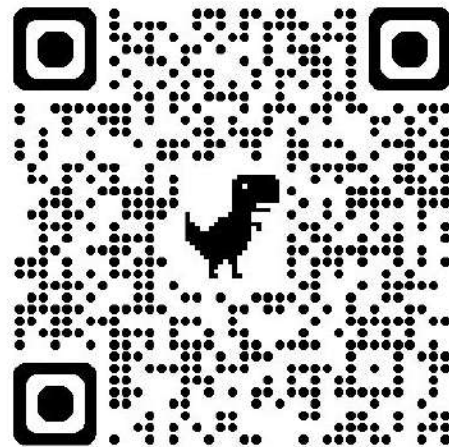
**Someshwaran Mohan Kumar**

@som23x  
[blog.someshwaran.me](https://blog.someshwaran.me)

**Developer Advocate, India**



Elasticsearch Logstash Kibana



# Let's Make this **Interactive?**

*Make a promise( )*



**Let's understand Who is Here?**



# Let's understand Who is Here?

- AI Engineers or Backend Developers? (anyone working with Vectors?)
- SREs or DevOps or FinOps engineers here?
- Any students or learners exploring AI for the first time?



# Why Am here?

From Float32 to BBQ: Practical Vector Search  
Optimization

# Quick Activity - Guess the Image/Person/Movie?

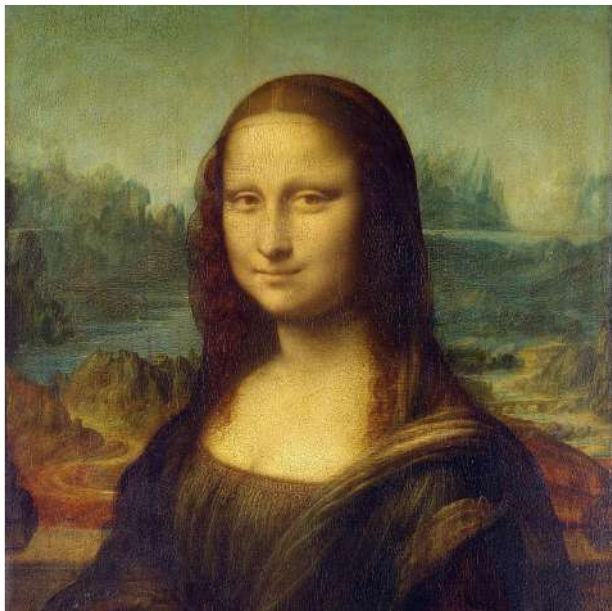
(If you know, **don't say it**, just raise your hand)

# Babu Bhaiya!



# 1 billion vectors @ 1024 dimensions

64 nodes



float32

16 nodes



int8

8 nodes



int4

= 32× RAM  
savings  
(vs. float32)



2 nodes



bit



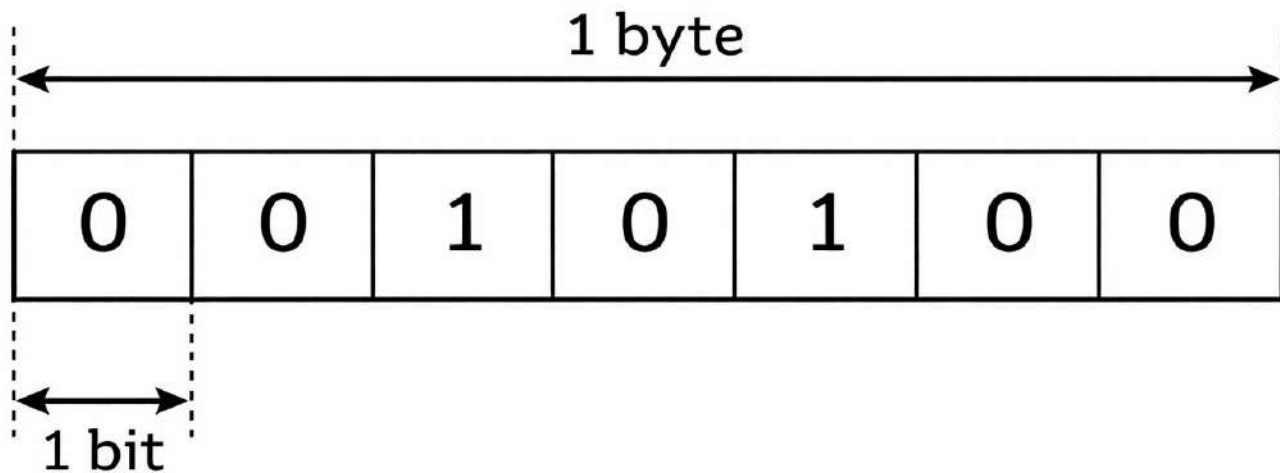
BBQ

# What's the Problem?

Why are we talking about it now?

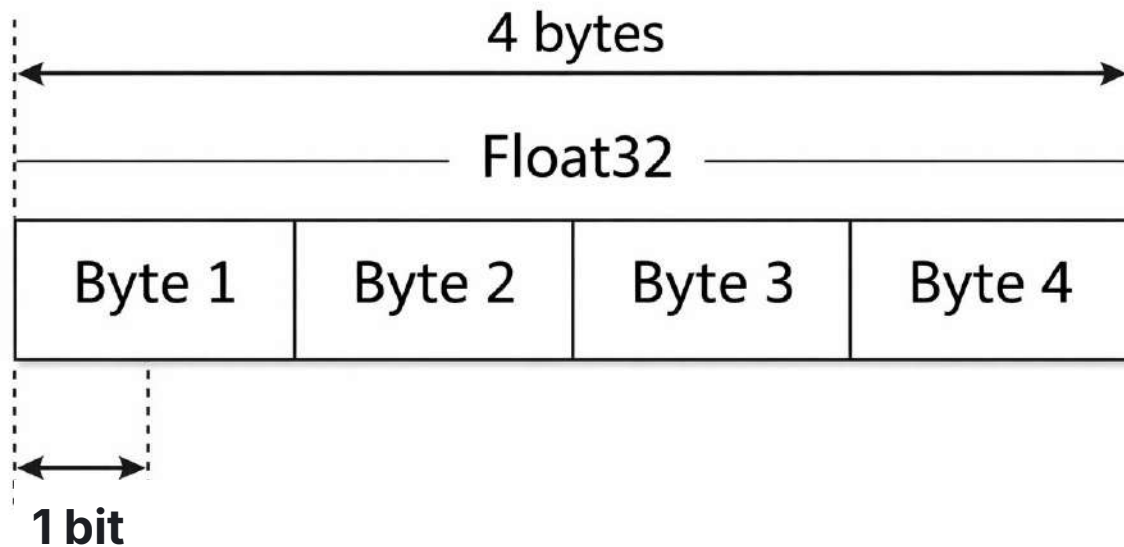
# Let's do some Math

Q1: How many **Bits** are 1-byte?



Q2: How many **Bits** are 1-float32 vector?

Q3: How many **Bytes** are 1-float32 vector?



**Q4: How many Bytes does a float32 Image (100 × 100 pixels) need?**

**Let's see in Practice!**  
**(notebook)**

# What's the Problem now?

Did we find it?

# Vector Traversal!

# Numbers you should know

## Latency Numbers Everyone Should Know

Operation	Time in ns	Time in ms (1ms = 1,000,000 ns)
L1 cache reference	1	
Branch misprediction	3	
L2 cache reference	4	
Mutex lock/unlock	17	
Main memory reference	100	
Compress 1 kB with Zippy	2,000	0.002
Read 1 MB sequentially from memory	10,000	0.010
Send 2 kB over 10 Gbps network	1,600	0.0016
SSD 4kB Random Read	20,000	0.020
Read 1 MB sequentially from SSD	1,000,000	1

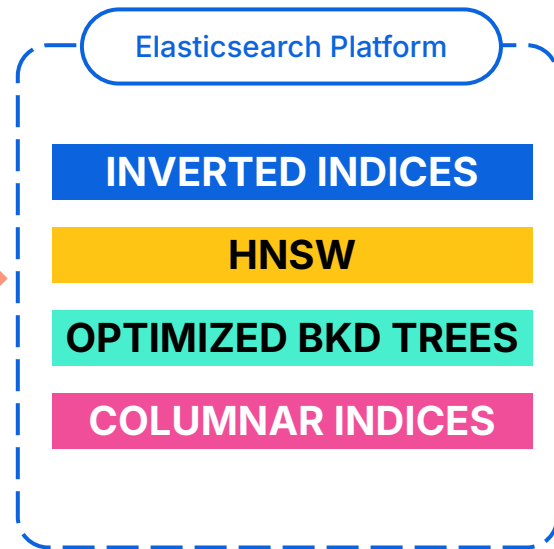
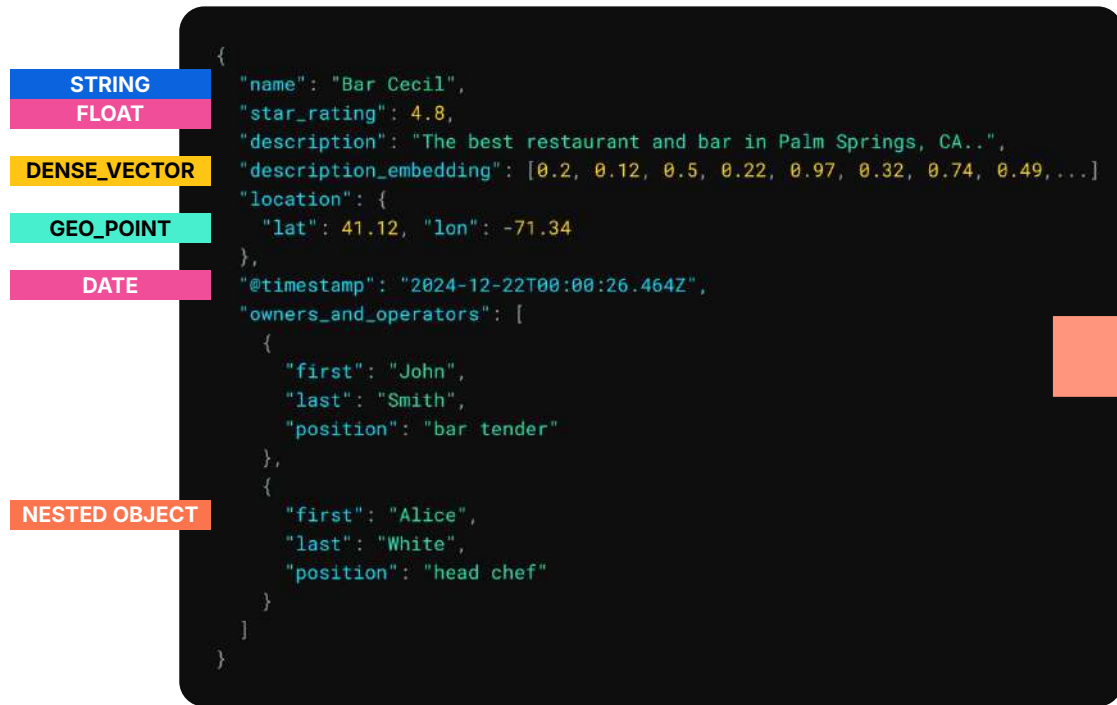
Vector from memory

Vector from disk

Lovingly borrowed from:

<https://static.googleusercontent.com/media/sre.google/en//static/pdf/rule-of-thumb-latency-numbers-letter.pdf>

# Your data is more than just vectors



# Vector Retrieval+Storing Techniques

**HNSW** (RAM-Resident)

**BBQ** (disk-resident vectors + RAM metadata)

If you're using HNSW, the graph must also be in memory. To estimate the required bytes, use the following formula below. The default value for the HNSW `m` parameter is 16.

$$\begin{aligned} \text{estimated bytes} &= \text{num\_vectors} \times 4 \times m \\ &= \text{num\_vectors} \times 4 \times 16 \end{aligned}$$

The following is an example of an estimate with the HNSW indexed `element_type: float` with no quantization, `m` set to 16, and 1,000,000 vectors of 1024 dimensions:

$$\begin{aligned} \text{estimated bytes} &= (1,000,000 \times 4 \times 16) + (1,000,000 \times 4 \times 1024) \\ &= 64,000,000 + 4,096,000,000 \\ &= 4,160,000,000 \\ &= 3.87\text{GB} \end{aligned}$$

If you're using DiskBBQ, a fraction of the clusters and centroids need to be in memory. When doing this estimation, it makes more sense to include both the index structure and the quantized vectors together as the structures are dependent. To estimate the total bytes, first compute the number of clusters, then compute the cost of the centroids plus the cost of the quantized vectors within the clusters to get the total estimated bytes. The default value for the number of `vectors_per_cluster` is 384.

$$\text{num\_clusters} = \frac{\text{num\_vectors}}{\text{vectors\_per\_cluster}} = \frac{\text{num\_vectors}}{384}$$

$$\begin{aligned} \text{estimated centroid bytes} &= \text{num\_clusters} \times \text{num\_dimensions} \times 4 \\ &\quad + \text{num\_clusters} \times (\text{num\_dimensions} + 14) \end{aligned}$$

$$\text{estimated quantized vector bytes} = \text{num\_vectors} \times ((\text{num\_dimensions}/8 + 14 + 2) \times 2)$$

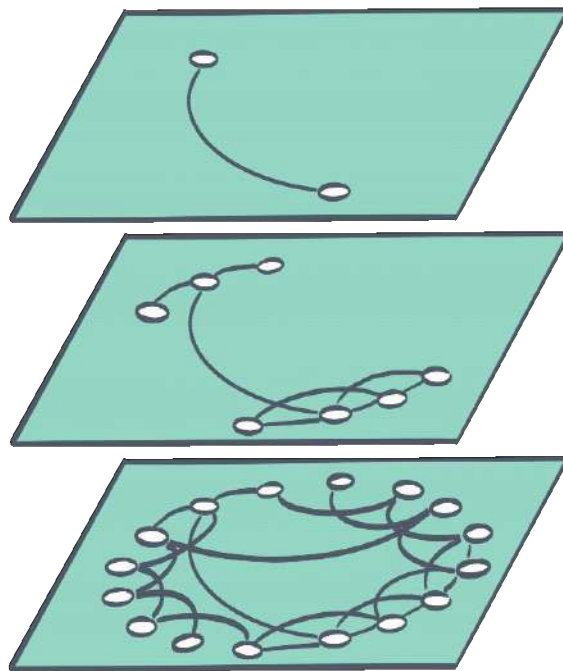
Note that the required RAM is for the filesystem cache, which is separate from the Java heap.

**Now, what's the problem this quantization solves?**

**We're trying to solve the **RAM** problem (For NOW)!**

# HNSW retrieval

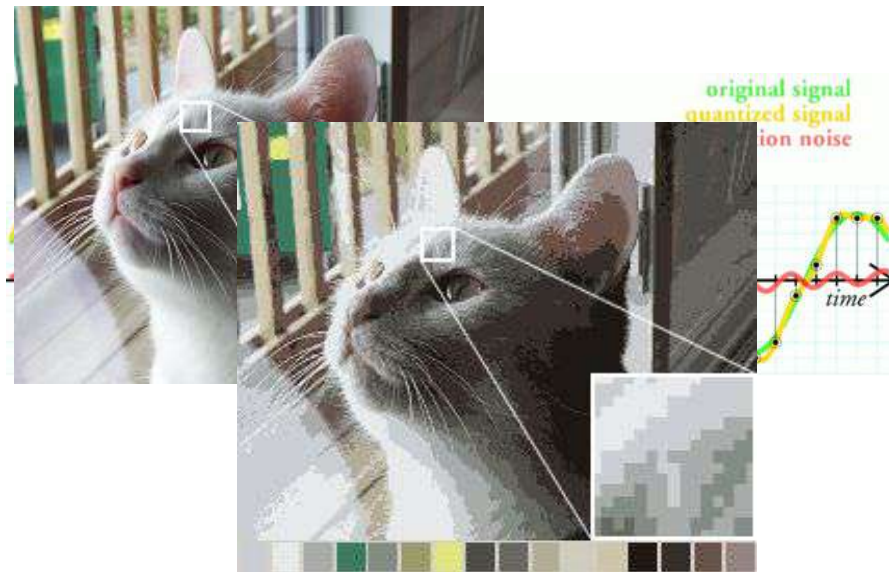
- Graph based indices (like Vamana & HNSW) are fastest
- Logarithmic search scale
- Random access of single vectors



*Linear scaling is for losers*

# Quantization 101

- Take continuous and make it discrete
- Lossy, but keeping the important stuff
- But, let's do it for vectors



float32

-1.0



1.0



-127



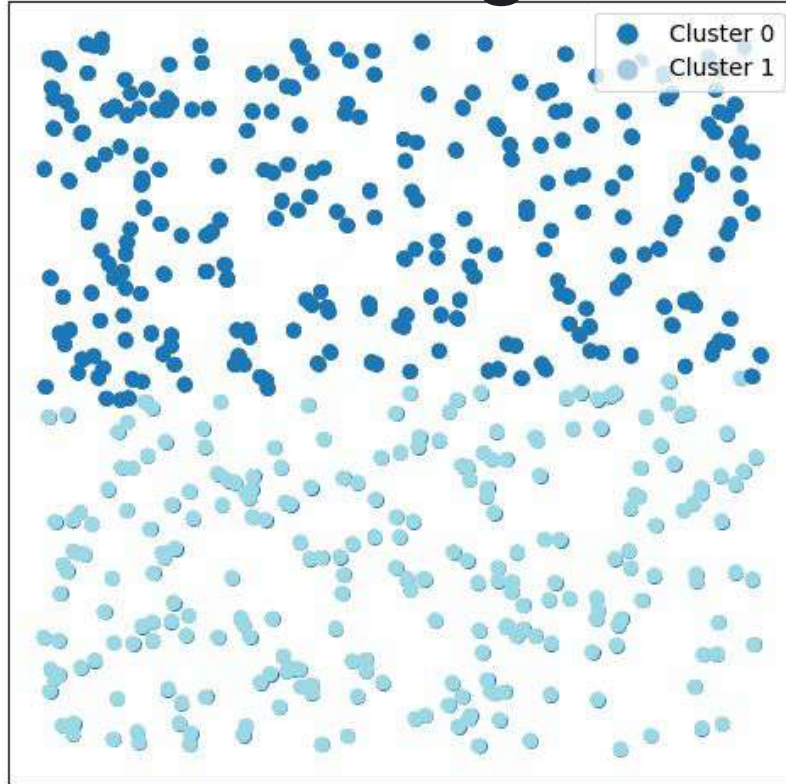
127

int8

**Now, we have a Second Problem!  
How do we load this to RAM from  
Disk?**

**DiskBBQ to the Rescue!**

# Clusterize the vectors while Indexing!



# Let's Connect the Dots!

- You have 100 million vectors
- each with 1536 dimensions

What is the size per Vector?

# 100 Million Vectors, 1536 Dimensions

## Memory Usage by Data Type



Note: Generated Image for cleaner representation

**Now, it makes everything Possible !**  
**And, Not Expensive !**

# Documentation References

## Quantization (BBQ, Scalar Quantization)

- <https://www.elastic.co/search-labs/blog/better-binary-quantization-lucene-elasticsearch>
- <https://www.elastic.co/search-labs/blog/optimized-scalar-quantization-elasticsearch>
- <https://www.elastic.co/search-labs/blog/bit-vectors-elasticsearch-bbq-vs-pq>
- <https://www.elastic.co/docs/reference/elasticsearch/mapping-reference/bbq>

## DiskBBQ

- <https://www.elastic.co/search-labs/blog/diskbbq-elasticsearch-introduction>
- <https://www.elastic.co/search-labs/blog/elasticsearch-latency-low-memory-diskbbq-hnswbbq-benchmark>

## HNSW Graphs

- <https://www.elastic.co/search-labs/blog/hnsw-graph>
- <https://www.elastic.co/search-labs/blog/hnsw-graphs-speed-up-merging>

## Related Benchmarks/Comparisons

- <https://www.elastic.co/search-labs/blog/elasticsearch-bbq-vs-opensearch-fais>