# Indexing Vectors

Alfan F. Wicaksono
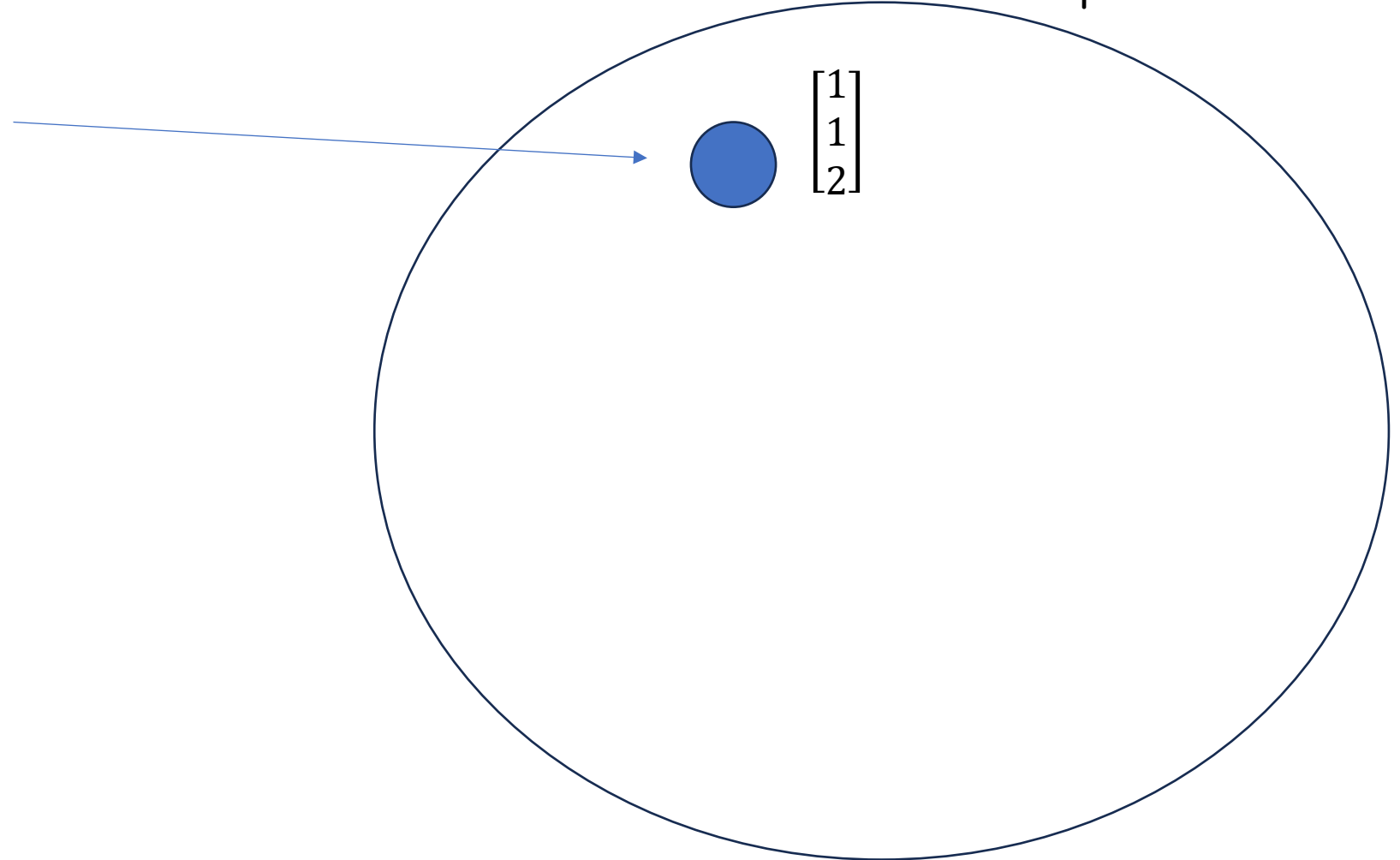
https://www.pinecone.io/learn/series/faiss/hnsw/

# Nowadays, everything can be encoded into vectors (or tensors)

inverted index (term document indexing), terlalu sparse, tidak bisa mengerti semantic meaning

**Vector Database**
"A collection of vectors from the same vector space"
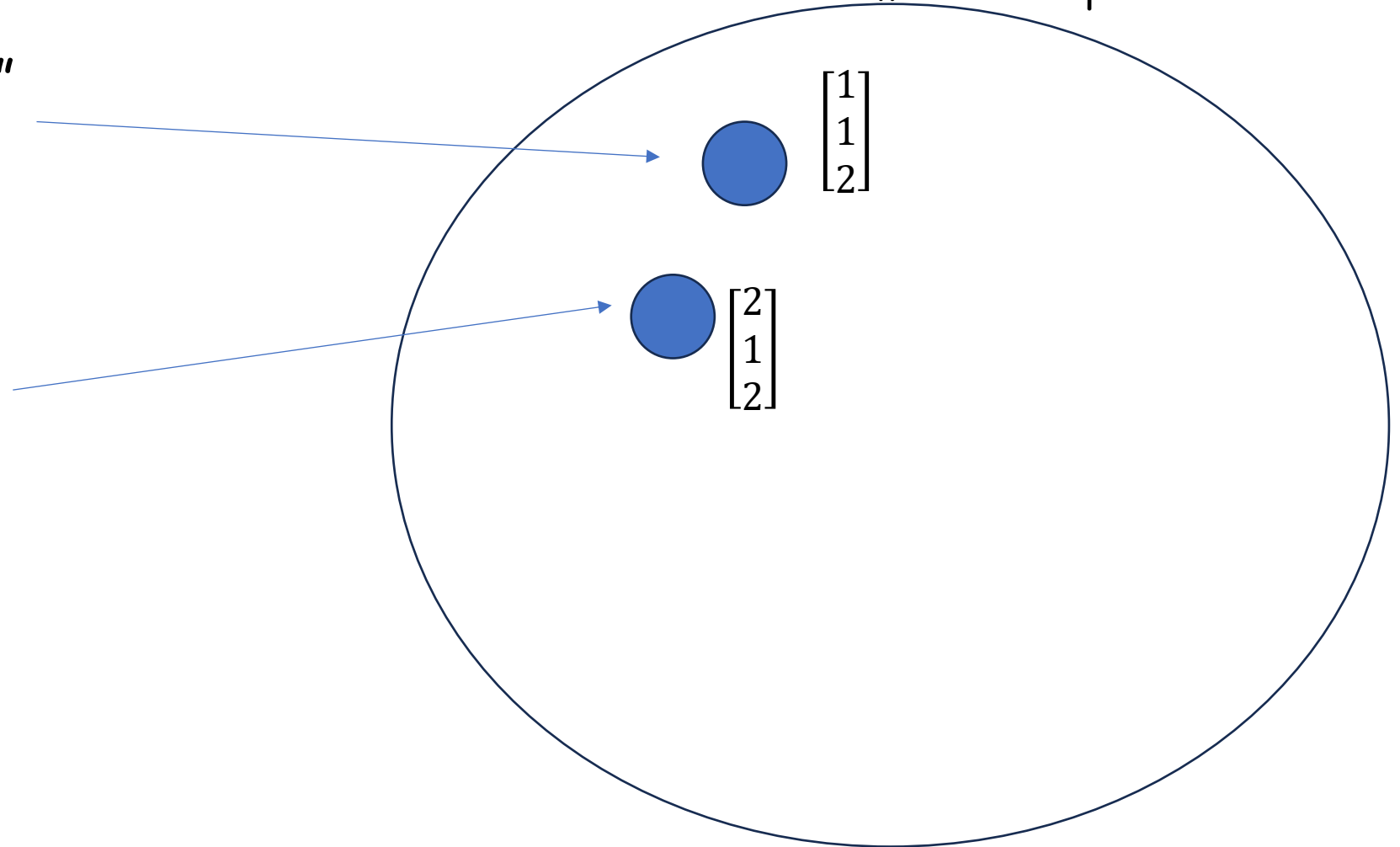
"Gedung Fasilkom UI"

$$\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

# Nowadays, everything can be encoded into vectors (or tensors)

**Vector Database**
*"A collection of vectors from the same vector space"*

"Gedung Fasilkom UI"

$$\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}$$

# Nowadays, everything can be encoded into vectors (or tensors)

**Vector Database**
"A collection of vectors from the same vector space"

"Gedung Fasilkom UI"

$$\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1.5 \\ 2 \end{bmatrix}$$

Music: Mars Fasilkom UI
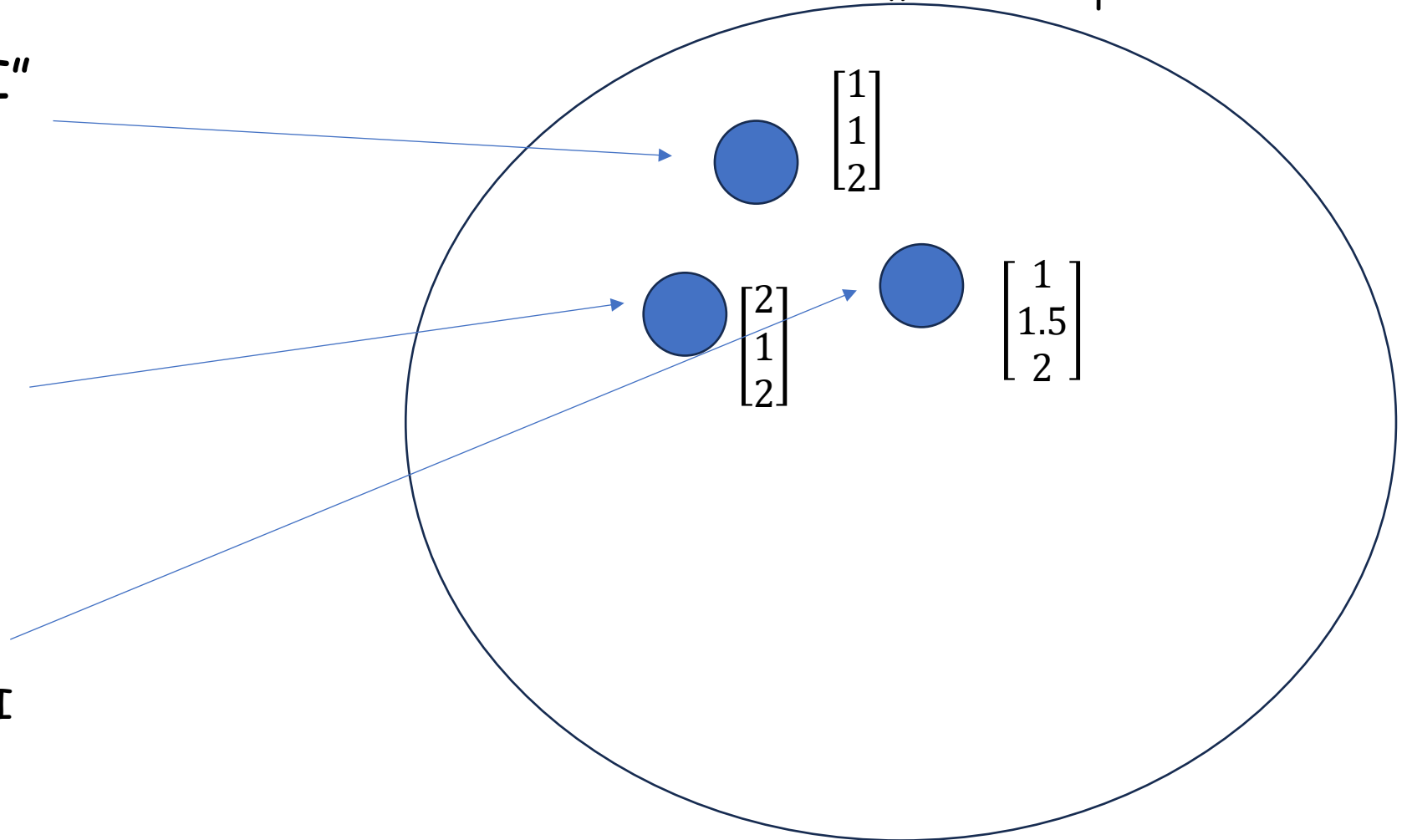
# Nowadays, everything can be encoded into vectors (or tensors)

**Vector Database**
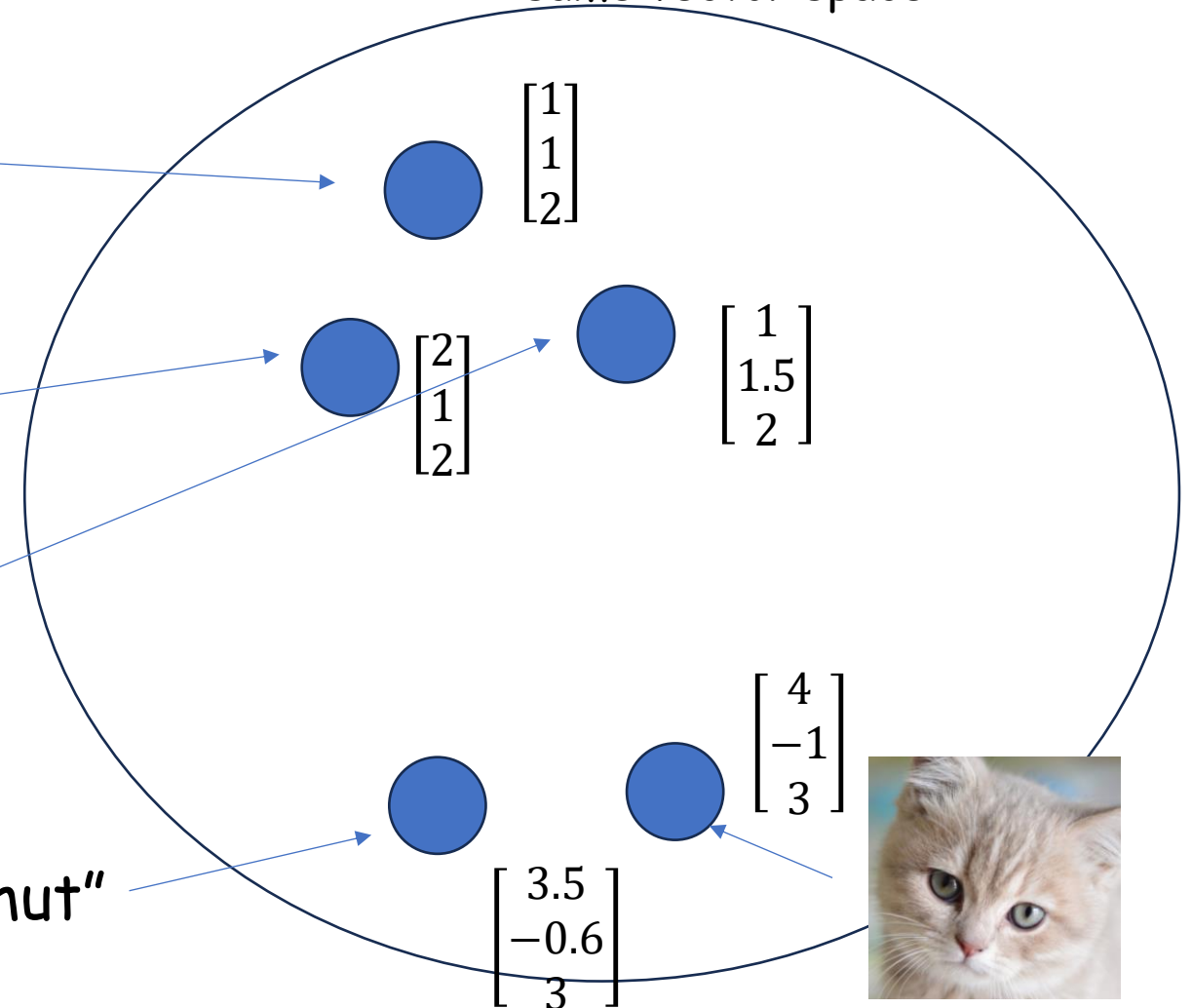"A collection of vectors from the same vector space"

"Gedung Fasilkom UI"



Music: Mars Fasilkom UI

"kucing imut"

$$\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1.5 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 4 \\ -1 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 3.5 \\ -0.6 \\ 3 \end{bmatrix}$$

# K-Nearest neighbor search

- Given a query (which can be in any format, including text, image, or audio), find other items that are "similar" to the query.
  - Nowadays, everything can be represented as **vectors**!

- Given a set $S$ of points in a space $M$ and a query point $q \in M$, find $k$ closest points in $S$ to $q$.

- $M$ is a **metric space** and similarity is expressed as a distance metric, which is symmetric and satisfies triangle inequality.
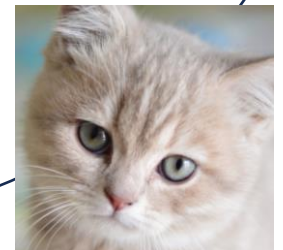  - Euclidean distance (**L2**), Inner Product, Manhattan distance, …

# Nowadays, everything can be encoded into vectors (or tensors)

**Vector Database**
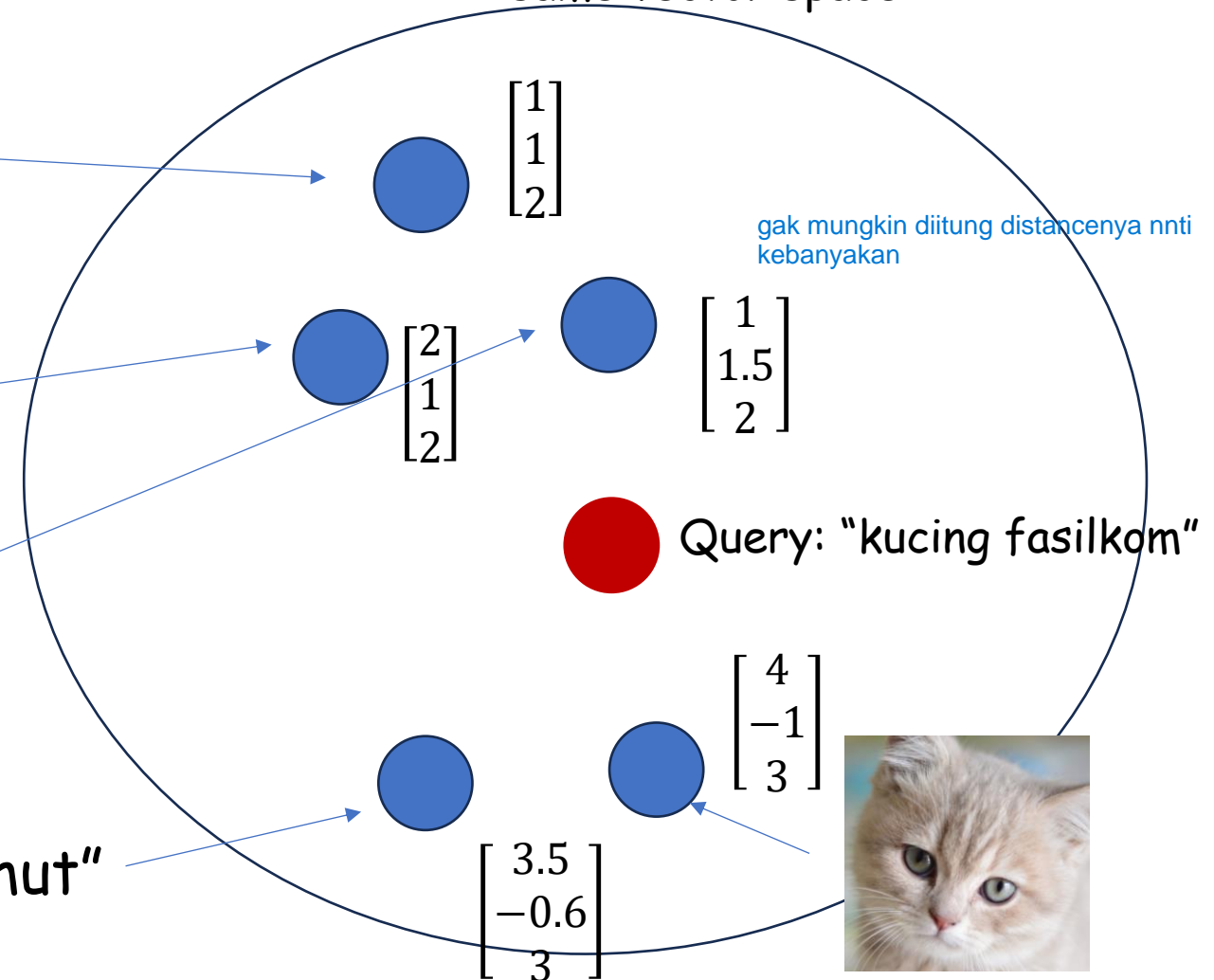"A collection of vectors from the same vector space"

"Gedung Fasilkom UI"

$$\begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$$

gak mungkin diitung distancenya nnti kebanyakan

$$\begin{bmatrix} 2 \\ 1 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ 1.5 \\ 2 \end{bmatrix}$$

Query: "kucing fasilkom"

Music: Mars Fasilkom UI

$$\begin{bmatrix} 4 \\ -1 \\ 3 \end{bmatrix}$$

"kucing imut"

$$\begin{bmatrix} 3.5 \\ -0.6 \\ 3 \end{bmatrix}$$
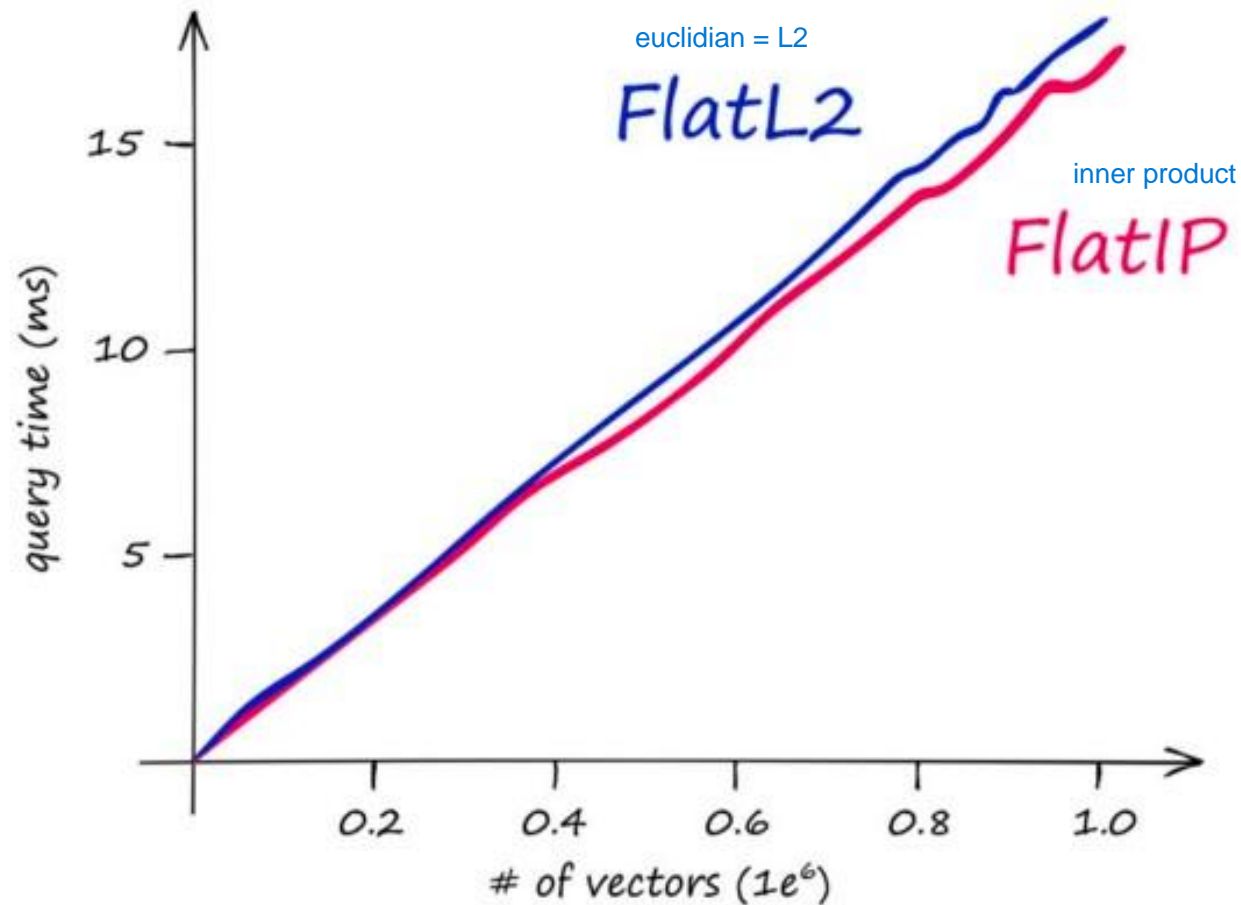
# Exact Solution – Flat Index



The query vector **xq** is against every other vectors in the index, calculating the distance to each and returning k nearest points.

# Exact Solution – Flat Index



**Tools FAISS**: Euclidean (L2) and Inner Product (IP) flat index search times using faiss-cpu on an M1 chip. Both using vector dimensionality of 100. IndexFlatIP is shown to be slightly faster than IndexFlatL2.

# Approximate nearest neighbors

- Flat index is 100% accurate, but slow and not scalable as the number of vectors in the database increases.

- One solution is to **reduce the search scope**.
  - We can somehow cluster and organize vectors into tree structures based on certain attributes or similarities.
  - We are no longer performing an exhaustive nearest-neighbors search but an **approximate nearest-neighbors**.

# Three Types ANN Algorithms

- Using Trees (e.g. k-d trees)
- **Using Locality Sensitive Hashing (e.g. Random Projection)**
- Using Graphs (Hierarchical Navigable Small Worlds)

# Typical Hashing

- Remember "Hash Table" on your last SDA course!
- We want a hash function that **minimizes collisions**; we really want to have multiple vectors mapped to a single key.
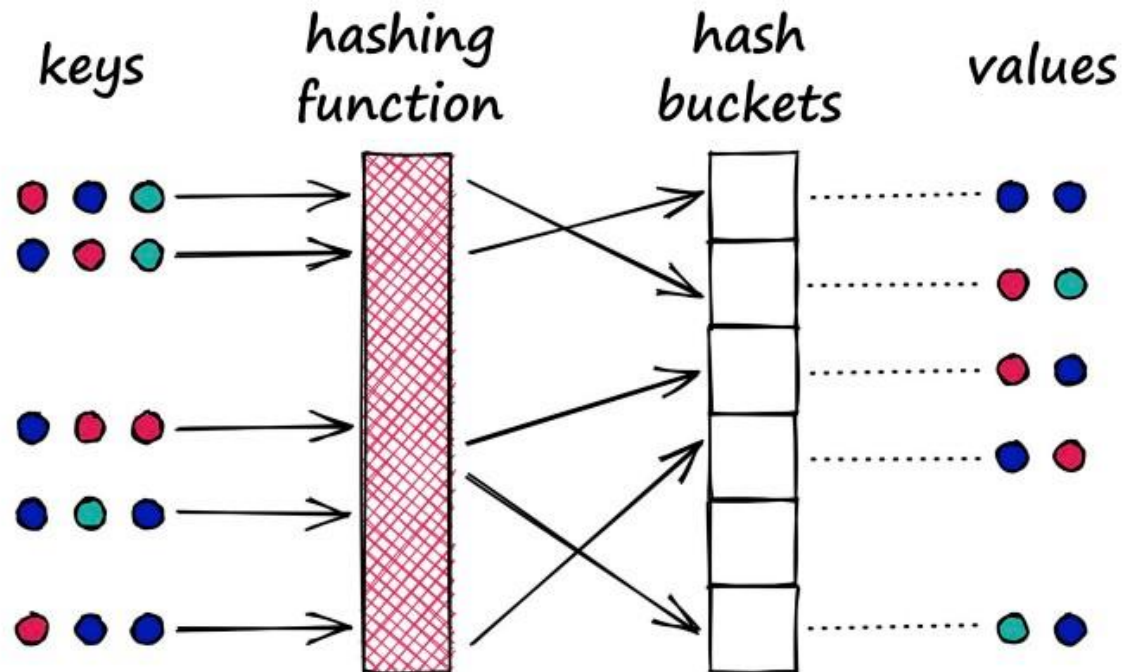
# Locality Sensitive Hashing (LSH)

local = yang similar mau di grup bareng

Instead of minimizing collisions, LSH aims at **maximizing** collisions in the sense that **similar vectors should be mapped into the same bucket**!
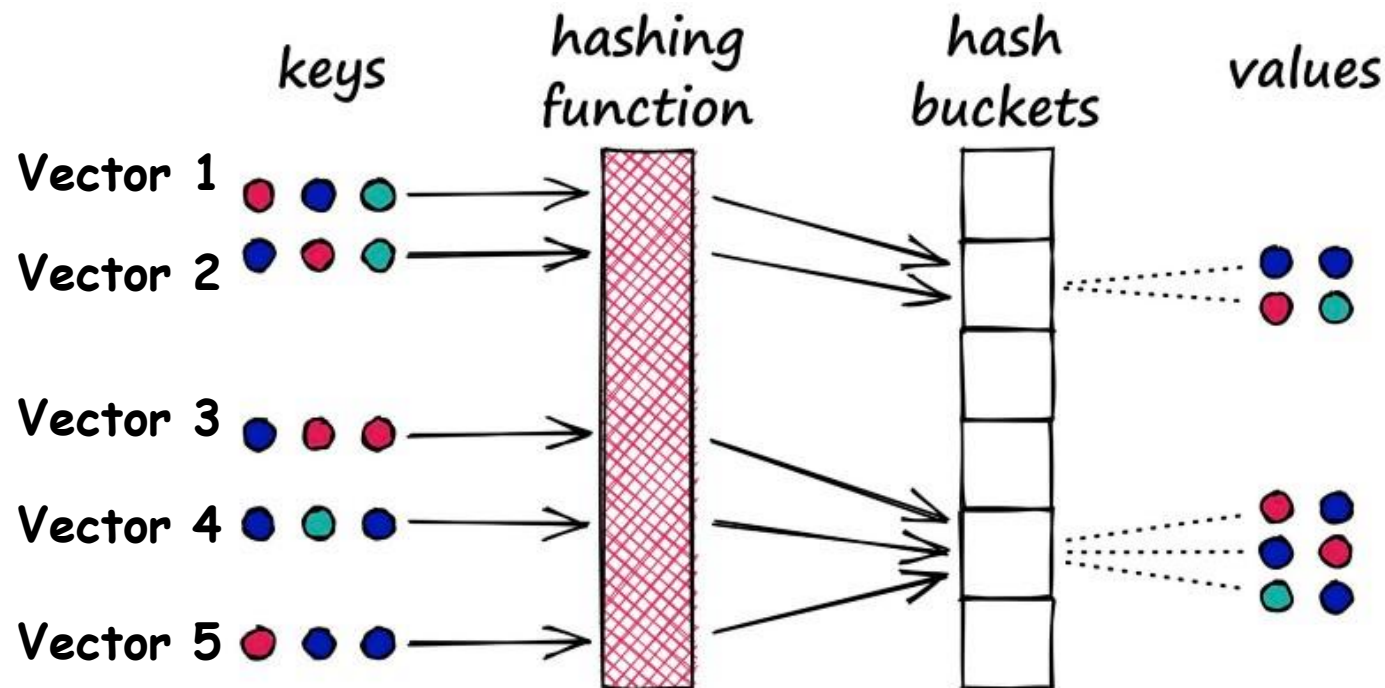
LSH: maksimalkan collision

Similar vectors mau dimapped ke bucket yang sama

Kalau masuk ke hash function, dimasukkin ke buckets yang sama.



This produces groups of vectors.

# Locality Sensitive Hashing (LSH)

Ketika ada **vector query**, ia akan di-hash dan akan dipetakan ke sebuah bucket. Perhitungan nearest neighbor hanya mempertimbangkan vector-vector di bucket yang sama.
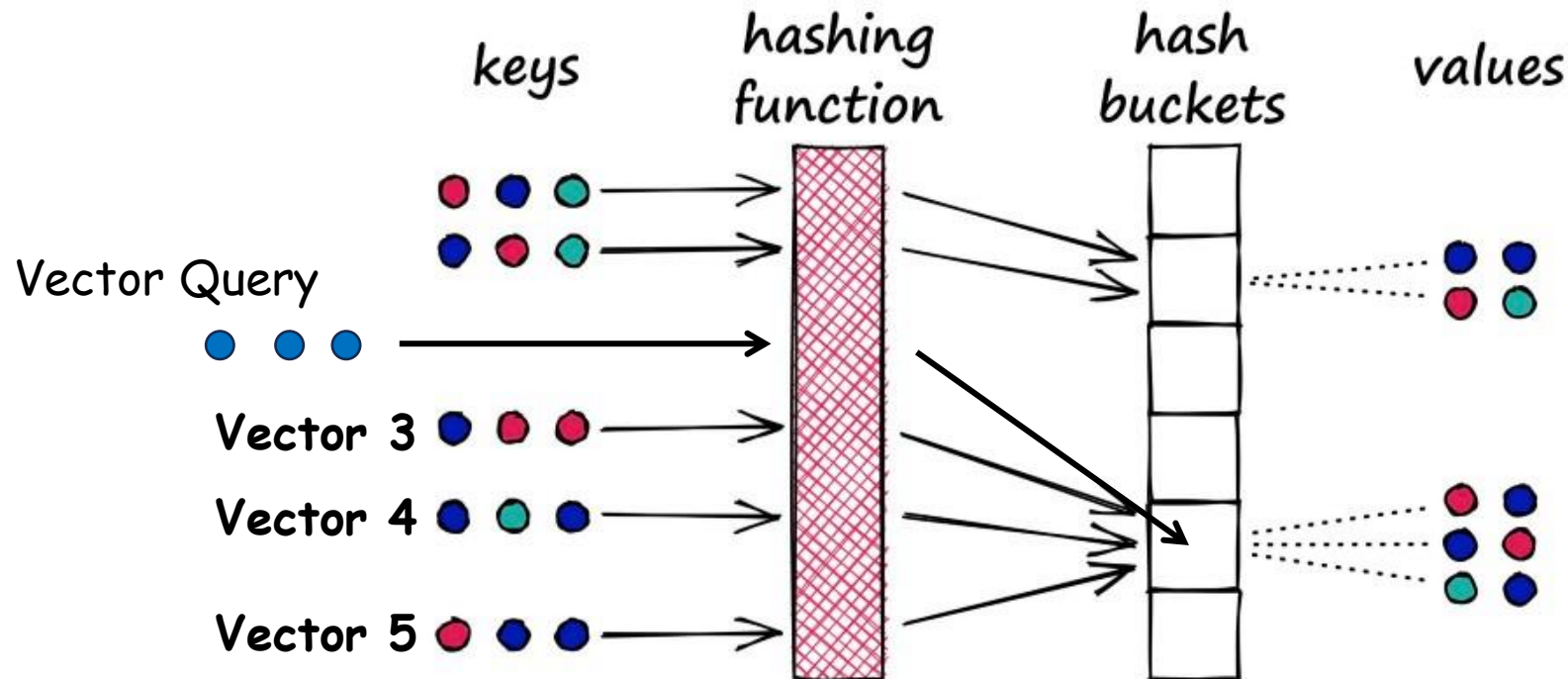


jadi kalau di vector database, dia itu udah di di gruping pas saat inserting ke database. Nah jadi proses search nya lebih bagus

Artinya, perhitungan similarity cukup dilakukan antara **Query dengan vector 3, 4, dan 5.**

# How to do LSH?

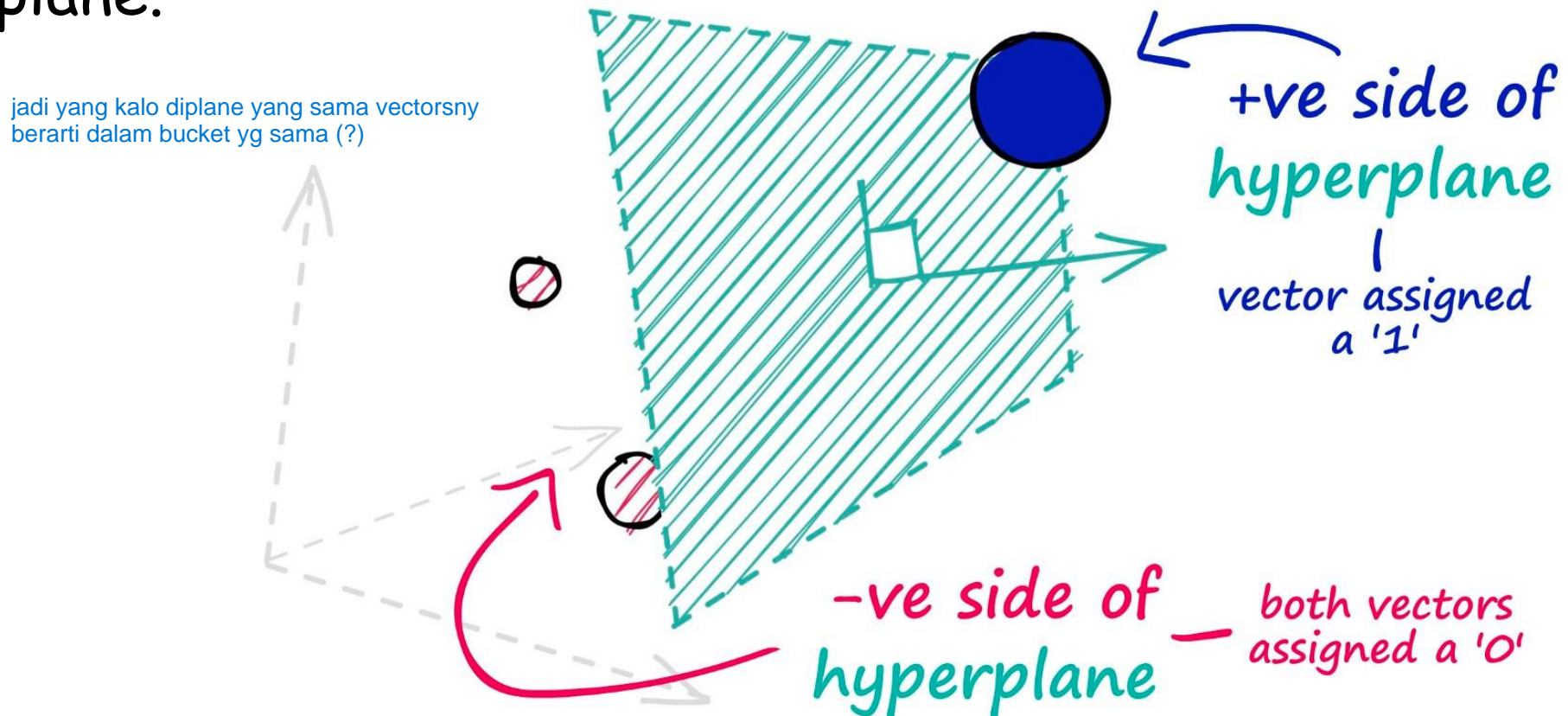One way is to perform **Random Projection** or **Random Hyperplanes**.

This is a mechanism to reduce our highly-dimensional vectors into **low-dimensionality binary vectors**.

**Two similar vectors should have the same low-dimensional binary vectors** --> this is our hash function that maximizes collisions!
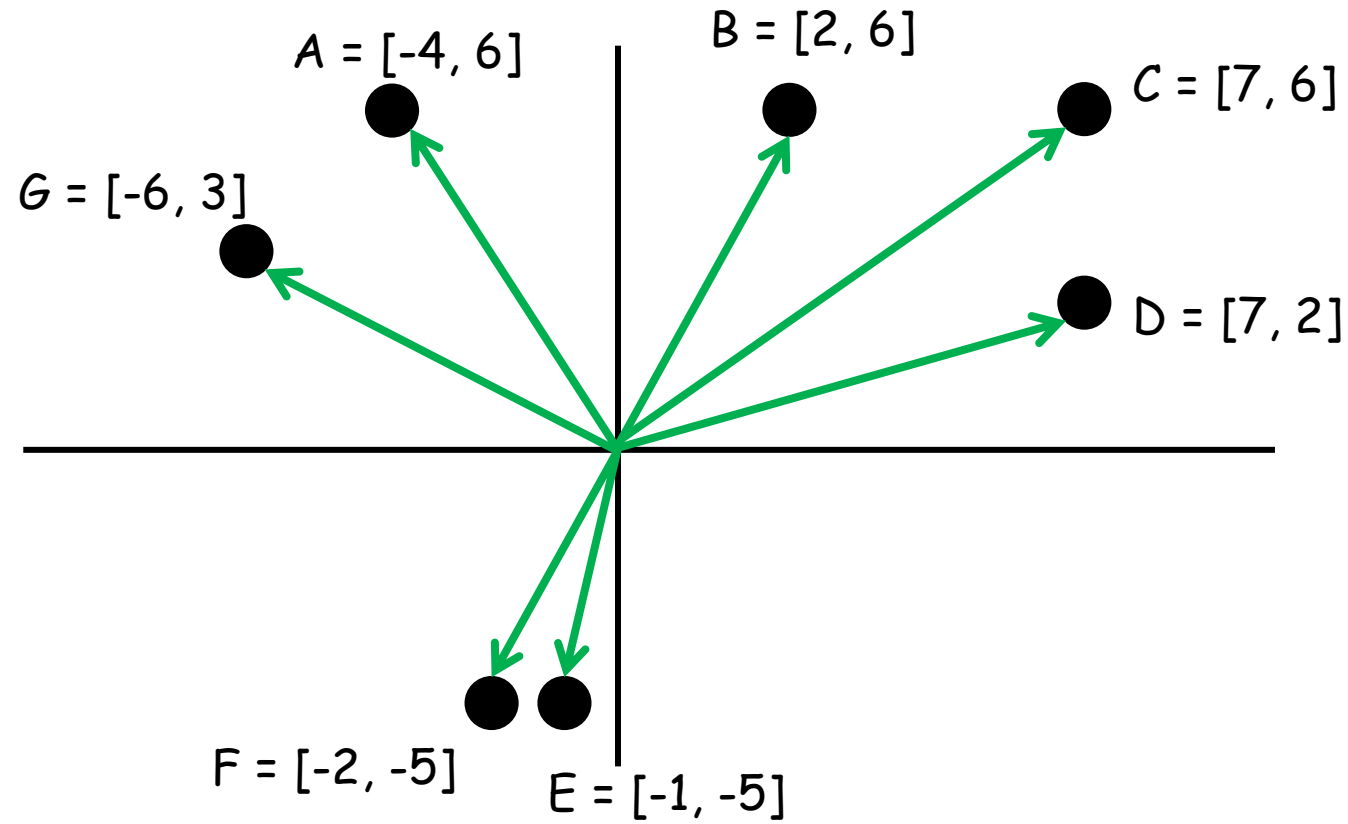
# Random Projections or Hyperplanes

We assign **a value of 1** to vectors on the +ve side of our hyperplane and **a value of 0** to vectors on the -ve side of the hyperplane.

jadi yang kalo diplane yang sama vectorsny
berarti dalam bucket yg sama (?)



+ve side of
hyperplane
|
vector assigned
a '1'

-ve side of
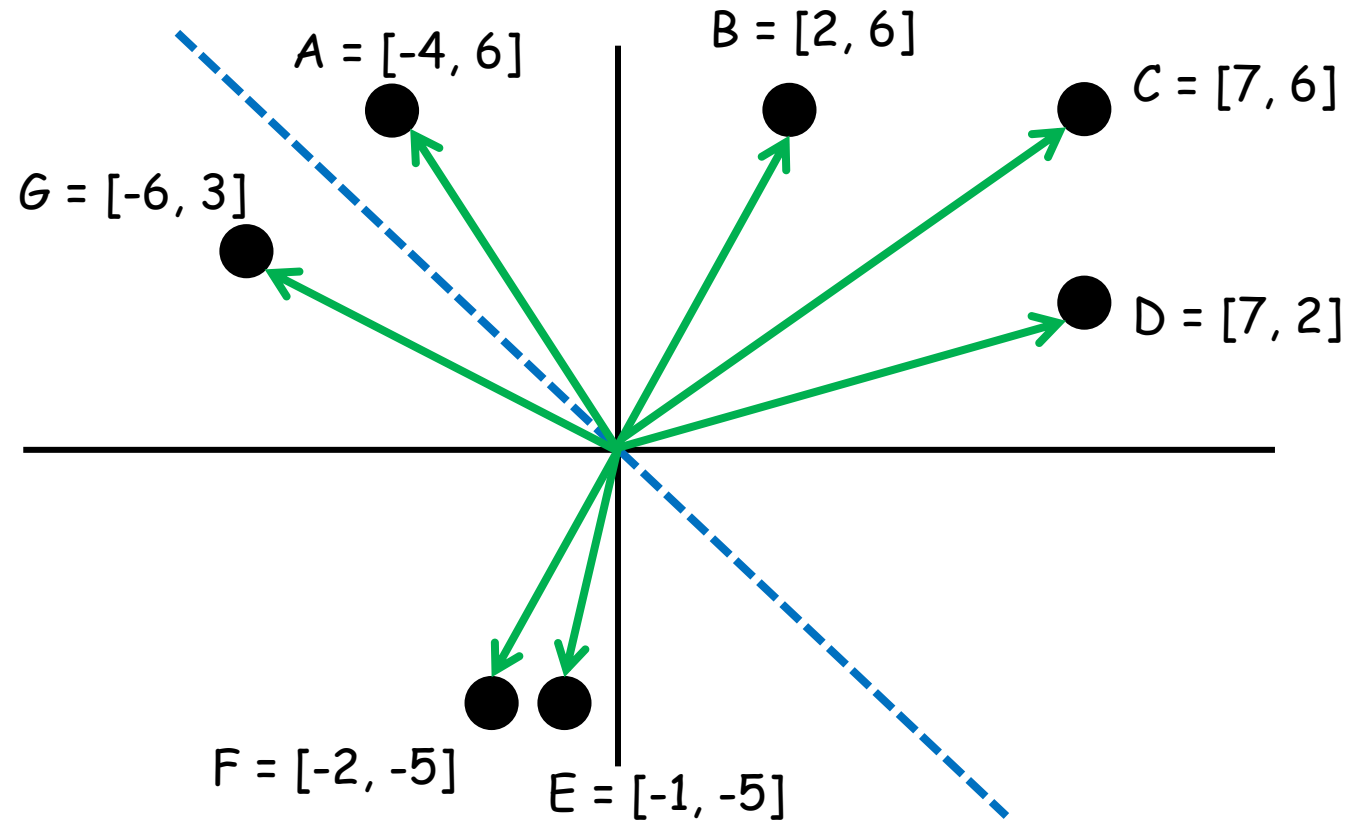hyperplane

both vectors
assigned a '0'

# Random Projections or Hyperplanes

Misal, ada beberapa vektor 2D

# Random Projections or Hyperplanes

Misal, kita hasilkan sebuah hyperplane **secara random**!



A = [-4, 6]
B = [2, 6]
C = [7, 6]
G = [-6, 3]
D = [7, 2]
F = [-2, -5]
E = [-1, -5]

area positif yang diatas garis biru

# Random Projections or Hyperplanes

Untuk identifikasi pada sisi hyperplane mana vektor-vektor berada, kita dapat menggunakan **normal vector (tegak lurus hyperplane)**, dan melakukan dot product dengannya.
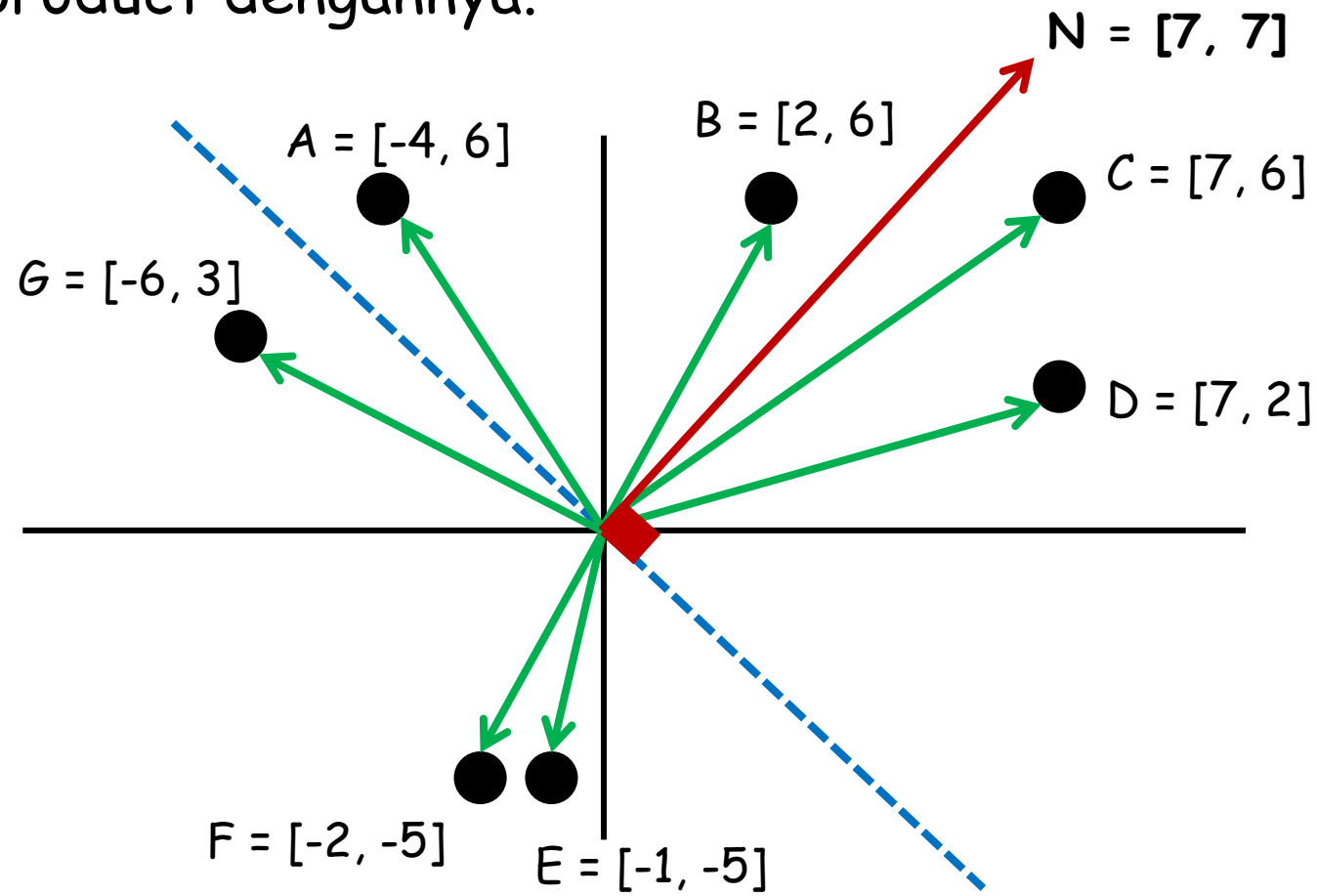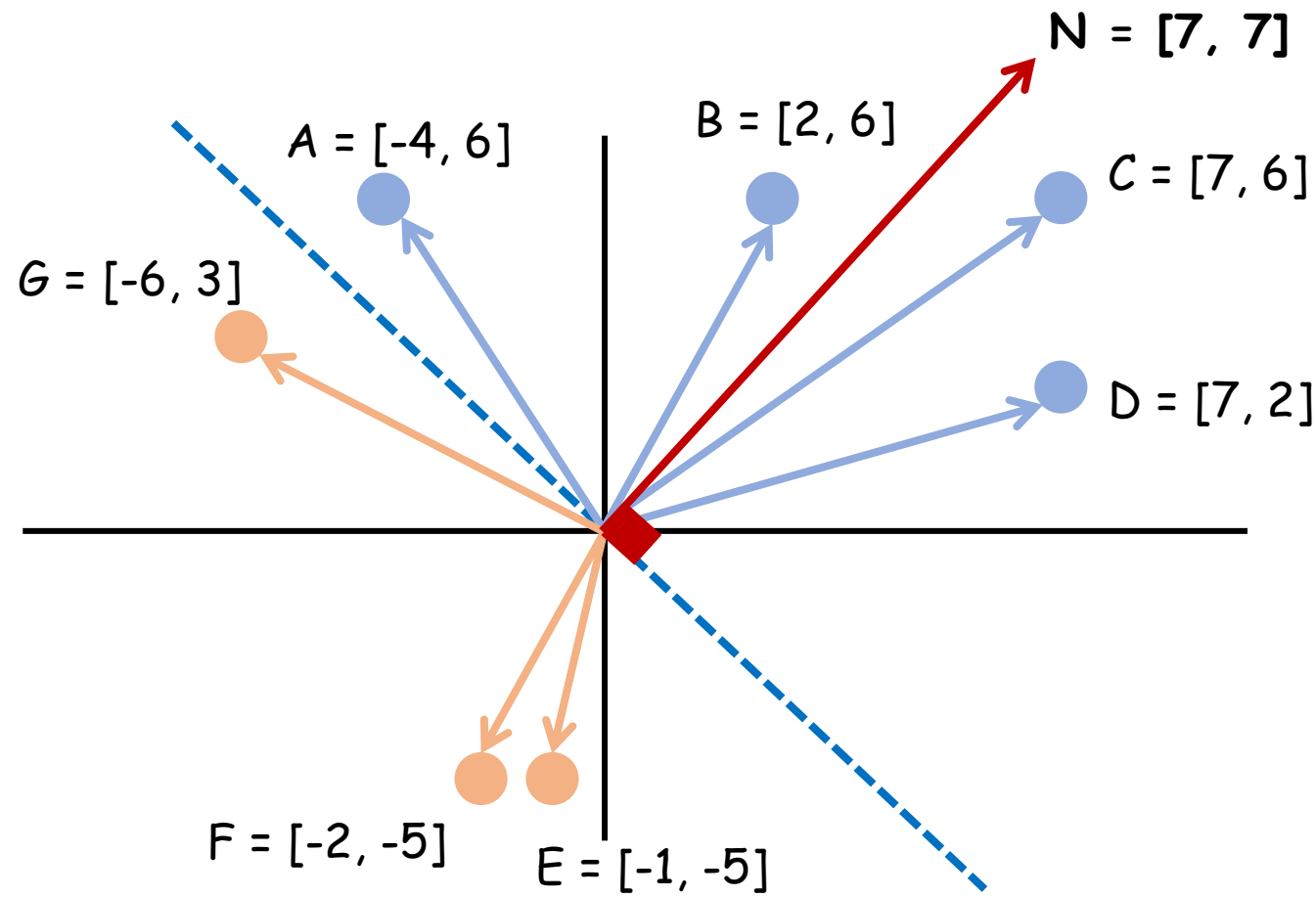
# Random Projections or Hyperplanes

Jika dua vector berada pada **arah yang sama**, dot product akan menghasilkan nilai positif; dan jika tidak, dot product akan negatif.

untuk tau positif/negative, di dot product dengan tegak lurus (yang garis merah)

N = **[7, 7]**

B = [2, 6]

A = [-4, 6]

C = [7, 6]

Kita encode vektor E, F, dan G dengan **[0]**

G = [-6, 3]

**dot(A,N) > 0**
**dot(B,N) > 0**
**dot(C,N) > 0**
**dot(D,N) > 0**

titik warna biru itu diatas,

D = [7, 2]

yang warna orange itu dibawah

**dot(G,N) < 0**
**dot(F,N) < 0**
**dot(E,N) < 0**

Kita encode vektor A, B, C, dan D dengan **[1]**

F = [-2, -5]

E = [-1, -5]

# Random Projections or Hyperplanes

Kita perlu tambah **random hyperplane** agar bisa menambah informasi. Contoh di bawah adalah dengan **nbits = 3**.
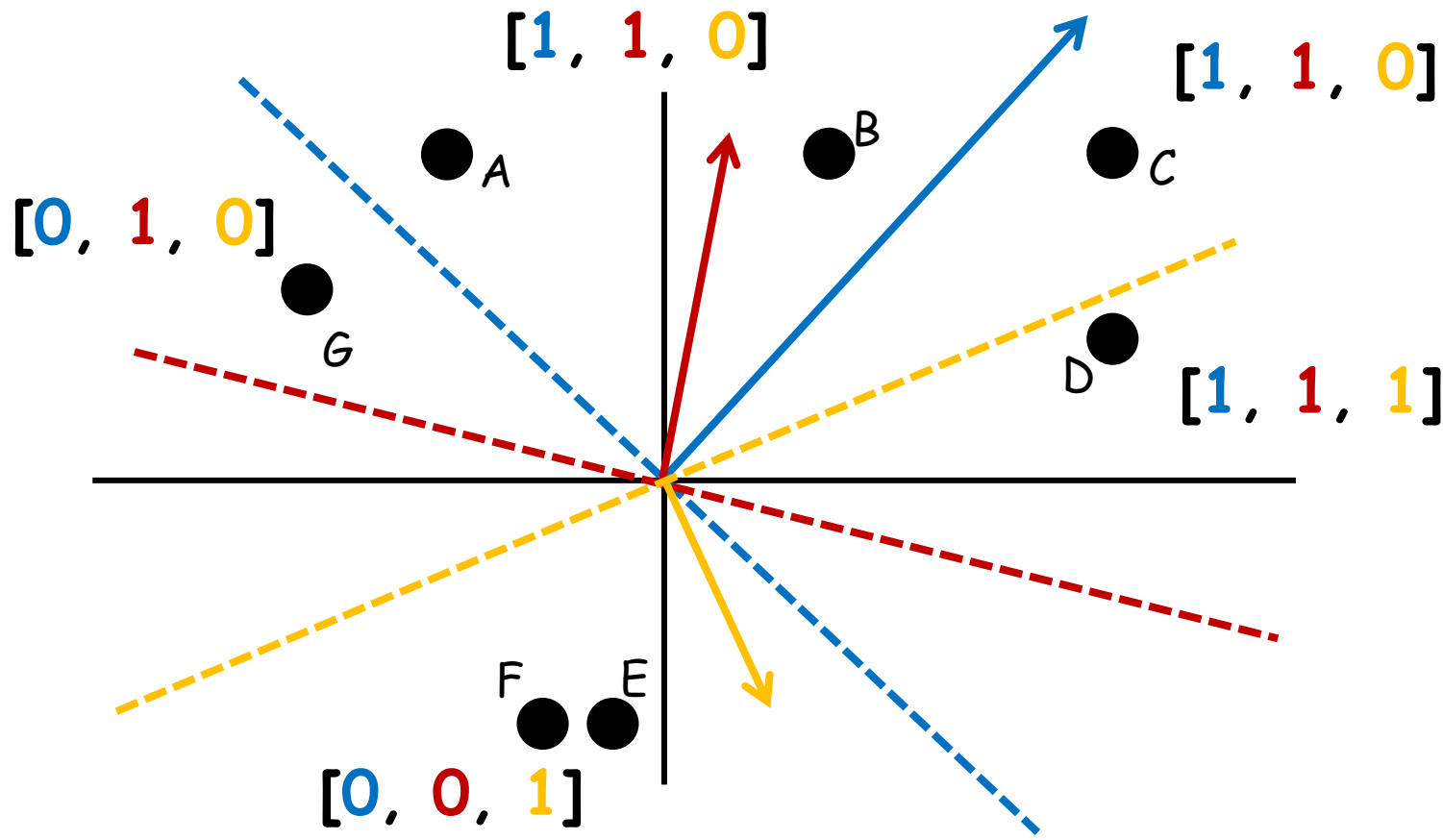


ada 3 random hyperplane

warna biru, merah, sama kuning

[1, 1, 0]

[1, 1, 0]

[0, 1, 0]

[1, 1, 1]

[0, 0, 1]
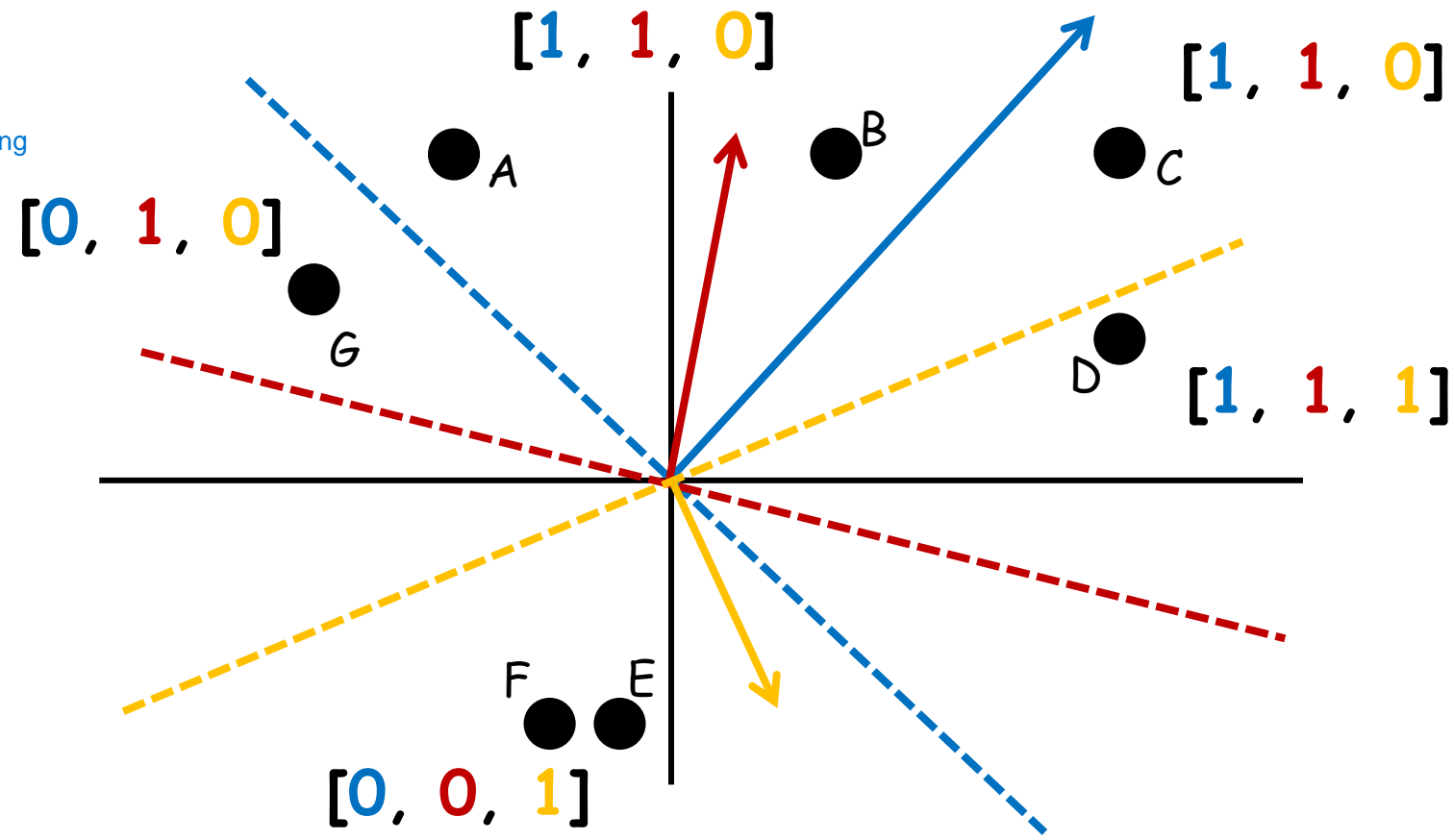
# Random Projections or Hyperplanes

Now, we have bucketed **eight** vectors!



{'110': [A, B, C],
 '111': [D],
 '001': [E, F],
 '010': [G]}

**This is our LSH index!**

# Random Projections or Hyperplanes

Misal, ada sebuah query vector **Q** yang di-hash ke [0, 0, 0]

karena ada 3 hyperplane setiap vektor mengandung array of positive/negative (terhadap garis tegak lurus

[1, 1, 0]

[1, 1, 0]

[0, 1, 0]

[1, 1, 1]

Q

A

B

C

G

D

F   E

[0, 0, 1]

```
{'110': [A, B, C],
 '111': [D],
 '001': [E, F],
 '010': [G]}
```
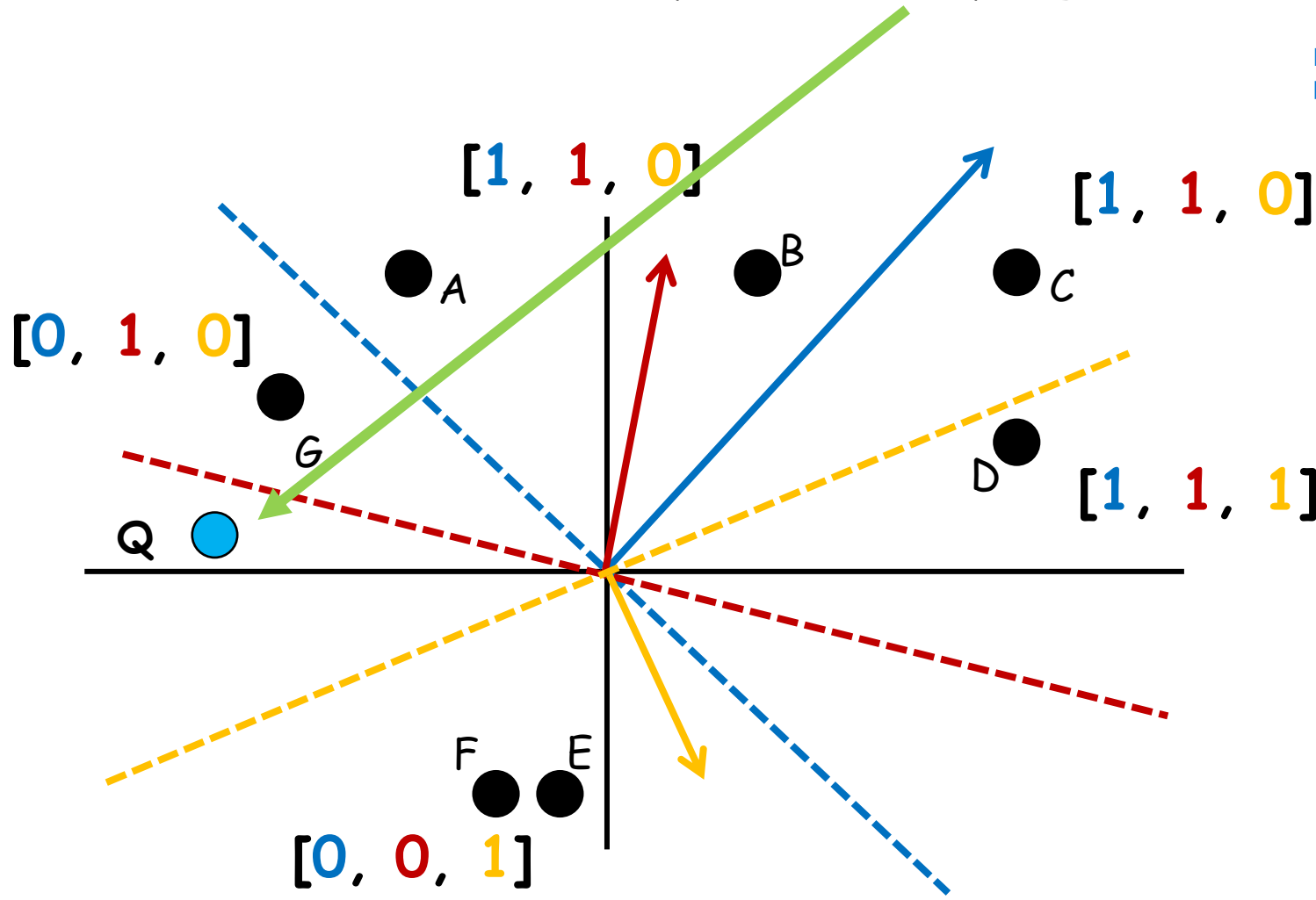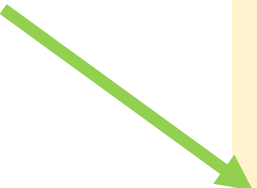
**This is our LSH index!**

# Random Projections or Hyperplanes

Misal, ada sebuah query vector **Q** yang di-hash ke **[0, 0, 0]**

Kita kemudian bandingkan query vector **Q** dengan semua **bucket** pada LSH index kita; dan ambil **Top-K vectors**.

Kita bisa gunakan **Hamming Distance**, yang menghitung "mismatch" antara dua binary vectors.

```
{'110': [A, B, C],
 '111': [D],
 '001': [E, F],
 '010': [G]}
```

**This is our LSH index!**

# Random Projections or Hyperplanes

Misal, ada sebuah query vector **Q** yang di-hash ke **[0, 0, 0]**

pokokny yang gak sama

kayak

Hamming Distance

000 vs 110 -> ada yang gak sama sebanyak
2 bit (2 MSB pertama)

```python
def hamming_distance(string1, string2) :

  if len(string1) != len(string2):
    raise ValueError("Panjang string harus sama.")

num_mismatch = 0
for n in range(len(string1)):
  if string1[n] != string2[n]:
    num_mismatch += 1
return num_mismatch
```

```
{'110': [A, B, C],
 '111': [D],
 '001': [E, F],
 '010': [G]}
```

**This is our LSH index!**

# Random Projections or Hyperplanes

Misal, ada sebuah query vector **Q** yang di-hash ke **[0, 0, 0]**

Hamming Distance (HD), misal **K = 3**

HD('000', '110') = 2
HD('000', '111') = 3
HD('000', '001') = 1
HD('000', '010') = 1

```
{'110': [A, B, C],
 '111': [D],
 '001': [E, F],
 '010': [G]}
```

This is our LSH index!

Top-3 = [E, F, G]

* Kita juga bisa re-ranking 3 top vectors ini berdasarkan nilai Cosine Similarity dengan Q

In reality, we use many more hyperplanes — more hyperplanes mean higher resolution binary vectors — producing much more precise representations of our vectors.

A higher **nbits** value improves search quality by increasing the resolution of hashed vectors.



nbits = 2

a = O 1
b = O 1

nbits = 8

a = O 1 1 1 O 1 O O
b = O 1 O 1 1 O O O
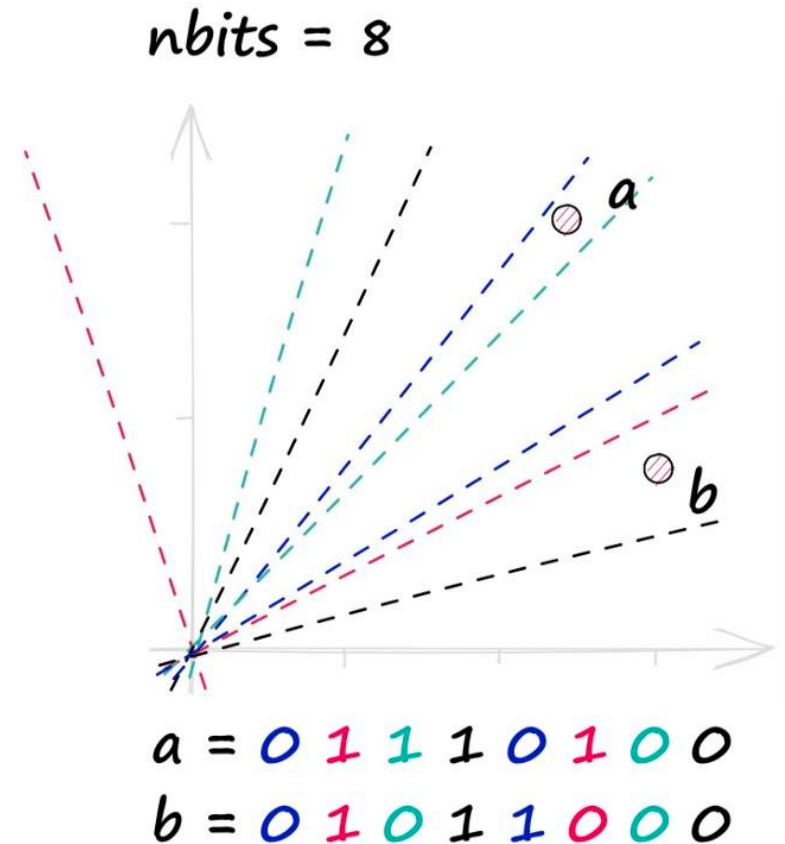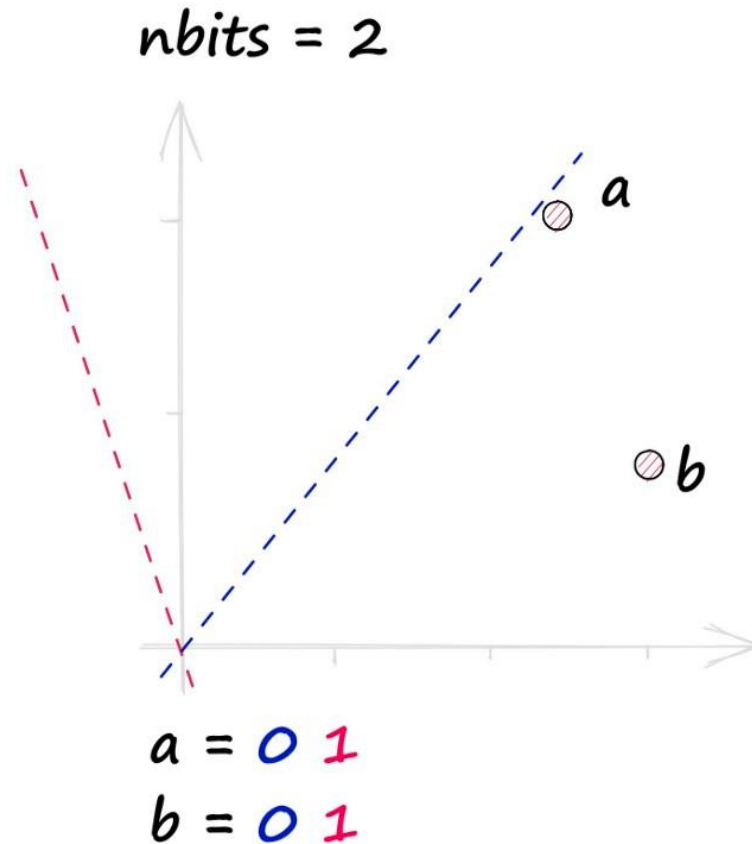
## Contoh Implementasi (Toy Implementation)

https://github.com/pinecone-io/examples/blob/master/learn/search/faiss-ebook/locality-sensitive-hashing-random-projection/random_projection.ipynb

```python
def all_binary(n):
    total = 1 << n
    print(f"{total} possible combinations")
    combinations = []
    for i in range(total):
        # get binary representation of integer
        b = bin(i)[2:]
        # pad zeros to start of binary representtion
        b = '0' * (n - len(b)) + b
        b = [int(i) for i in b]
        combinations.append(b)
    return combinations
```

```python
class RandomProjection:
    # initialize what will be the buckets
    buckets = {}
    # initialize counter
    counter = 0

    def __init__(self, nbits, d):
        self.nbits = nbits
        self.d = d
        # create our hyperplane normal vecs for splitting data
        self.plane_norms = np.random.rand(d, nbits) - .5
        print(f"Initialized {self.plane_norms.shape[1]} hyperplane normal vectors.")
        # add every possible combination to hashes attribute as numpy array
        self.hashes = all_binary(nbits)
        # and add each as a key to the buckets dictionary
        for hash_code in self.hashes:
            # convert to string
            hash_code = ''.join([str(i) for i in hash_code])
            self.buckets[hash_code] = []
        # convert self.hashes to numpy array
        self.hashes = np.stack(self.hashes)
```

```python
def get_binary(self, vec):
    # calculate nbits dot product values
    direction = np.dot(vec, projection.plane_norms)
    # find positive direction (>0) and negative direction (<=0)
    direction = direction > 0
    # convert boolean array to integer strings
    binary_hash = direction.astype(int)
    return binary_hash

def hash_vec(self, vec, show=False):
    # generate hash
    binary_hash = self.get_binary(vec)
    # convert to string format for dictionary
    binary_hash = ''.join(binary_hash.astype(str))
    # add ID to buckets dictionary
    self.buckets[binary_hash].append(self.counter)
    if show:
        print(f"{self.counter}: {''.join(binary_hash)}")
    # increment counter
    self.counter += 1
```

https://github.com/pinecone-io/examples/blob/master/learn/search/faiss-ebook/locality-sensitive-hashing-random-projection/random_projection.ipynb

```python
def hamming(self, hashed_vec):
    # get hamming distance between query vec and all buckets in self.hashes
    hamming_dist = \
        np.count_nonzero(hashed_vec != projection.hashes, axis=1).reshape(-1, 1)
    # add hash values to each row
    hamming_dist = np.concatenate((projection.hashes, hamming_dist), axis=1)
    # sort based on distance
    hamming_dist = hamming_dist[hamming_dist[:, -1].argsort()]
    return hamming_dist
```

https://github.com/pinecone-io/examples/blob/master/learn/search/faiss-ebook/locality-sensitive-hashing-random-projection/random_projection.ipynb

```python
def top_k(self, vec, k=5):
    # generate hash
    binary_hash = self.get_binary(vec)
    # calculate hamming distance between all vectors
    hamming_dist = self.hamming(binary_hash)
    # loop through each bucket until we have k or more vector IDs
    vec_ids = []
    for row in hamming_dist:
        str_hash = ''.join(row[:-1].astype(str))
        bucket_ids = self.buckets[str_hash]
        vec_ids.extend(bucket_ids)
        if len(vec_ids) >= k:
            vec_ids = vec_ids[:k]
            break
    # return top k IDs
    return vec_ids
```

https://github.com/pinecone-io/examples/blob/master/learn/search/faiss-ebook/locality-sensitive-hashing-random-projection/random_projection.ipynb

# Hierarchical Navigable Small Worlds
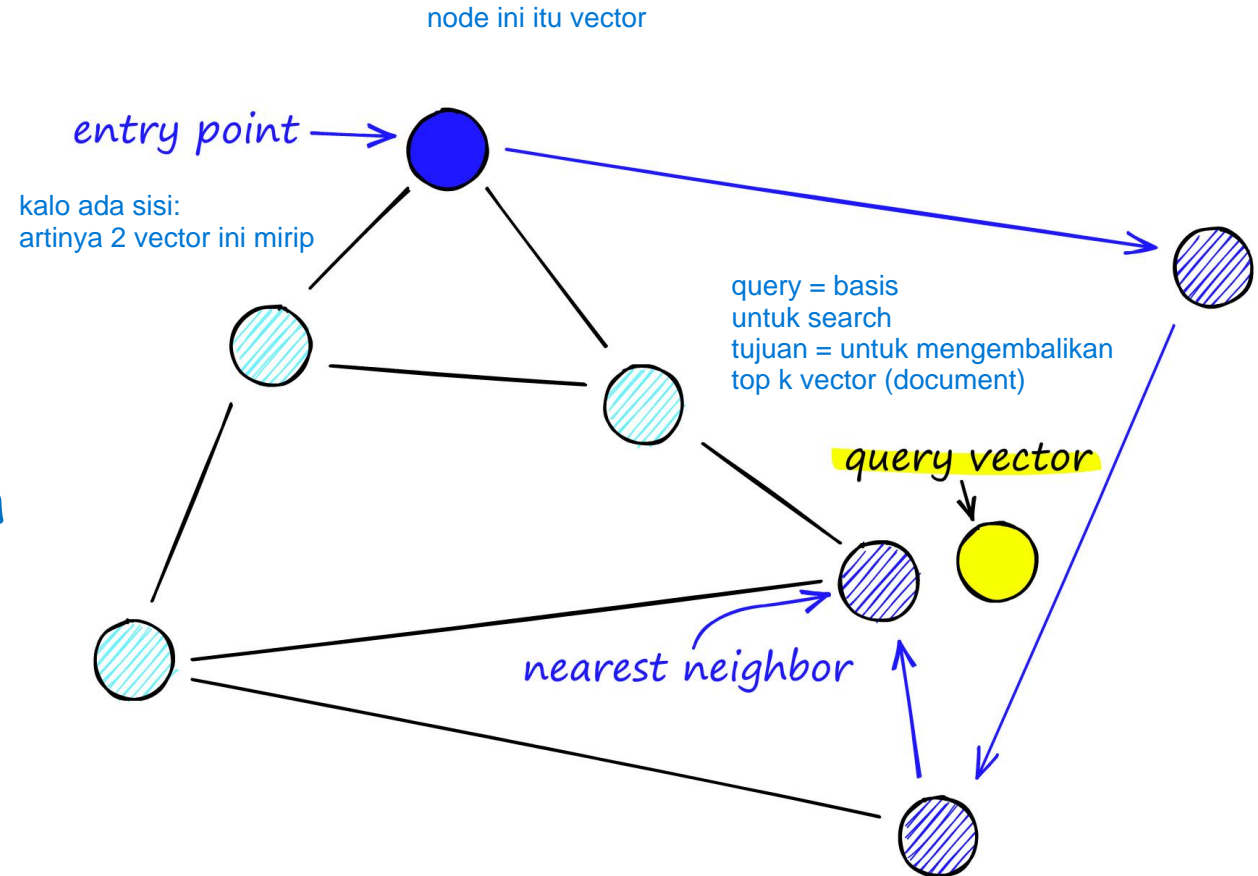
Semua materi diambil tanpa malu dari:

https://www.pinecone.io/learn/series/faiss/hnsw/

https://towardsdatascience.com/similarity-search-part-4-hierarchical-navigable-small-world-hnsw-2aad4fe87d37

# Navigable Small World Graphs

When searching an NSW graph, we begin at a pre-defined **entry-point**. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there.

We repeat the **greedy-routing search** process of moving from vertex to vertex by identifying the nearest neighboring vertices in each friend list. **Eventually, we will find no nearer vertices than our current vertex** — this is a local minimum and acts as our **stopping condition**.

https://www.pinecone.io/learn/series/faiss/hnsw/

node ini itu vector

entry point

kalo ada sisi:
artinya 2 vector ini mirip

query = basis
untuk search
tujuan = untuk mengembalikan
top k vector (document)

query vector

nearest neighbor

# Navigable Small World Graphs

When searching an NSW graph, we begin at a pre-defined **entry-point**. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there.
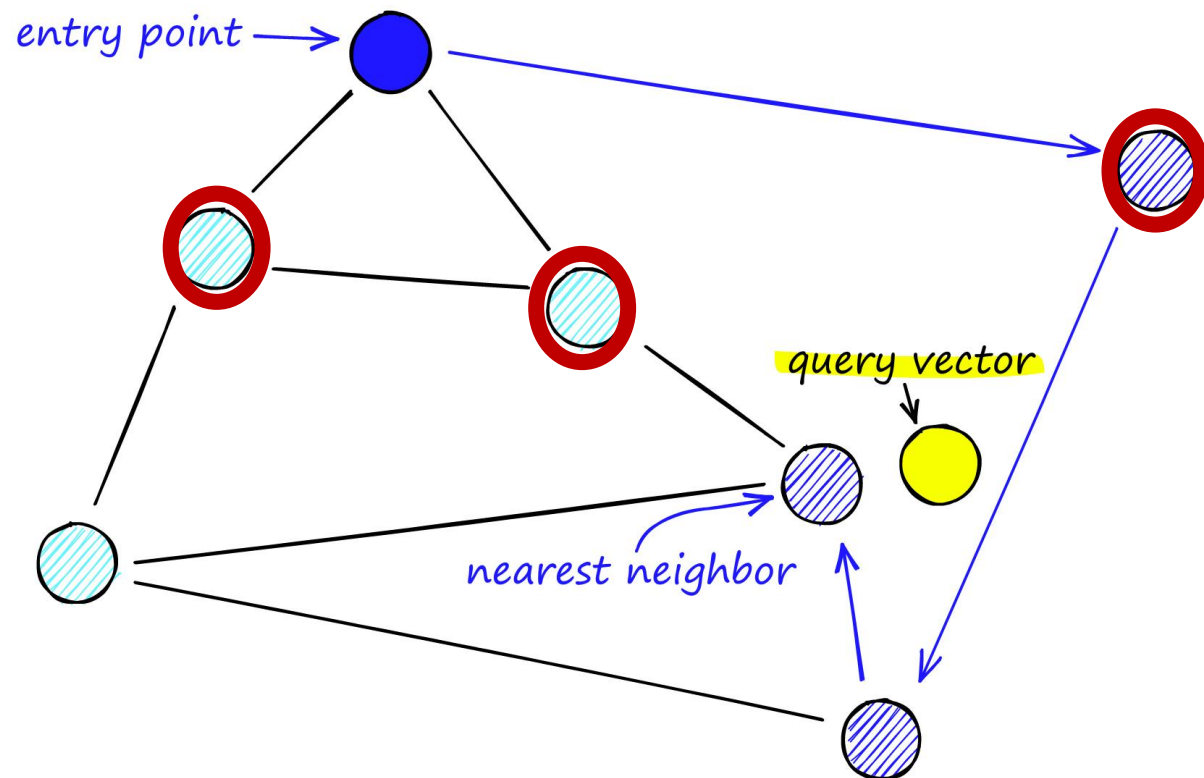
We repeat the **greedy-routing search** process of moving from vertex to vertex by identifying the nearest neighboring vertices in each friend list. **Eventually, we will find no nearer vertices than our current vertex** — this is a local minimum and acts as our **stopping condition**.



entry point

query vector

nearest neighbor

https://www.pinecone.io/learn/series/faiss/hnsw/

# Navigable Small World Graphs

When searching an NSW graph, we begin at a pre-defined **entry-point**. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there.
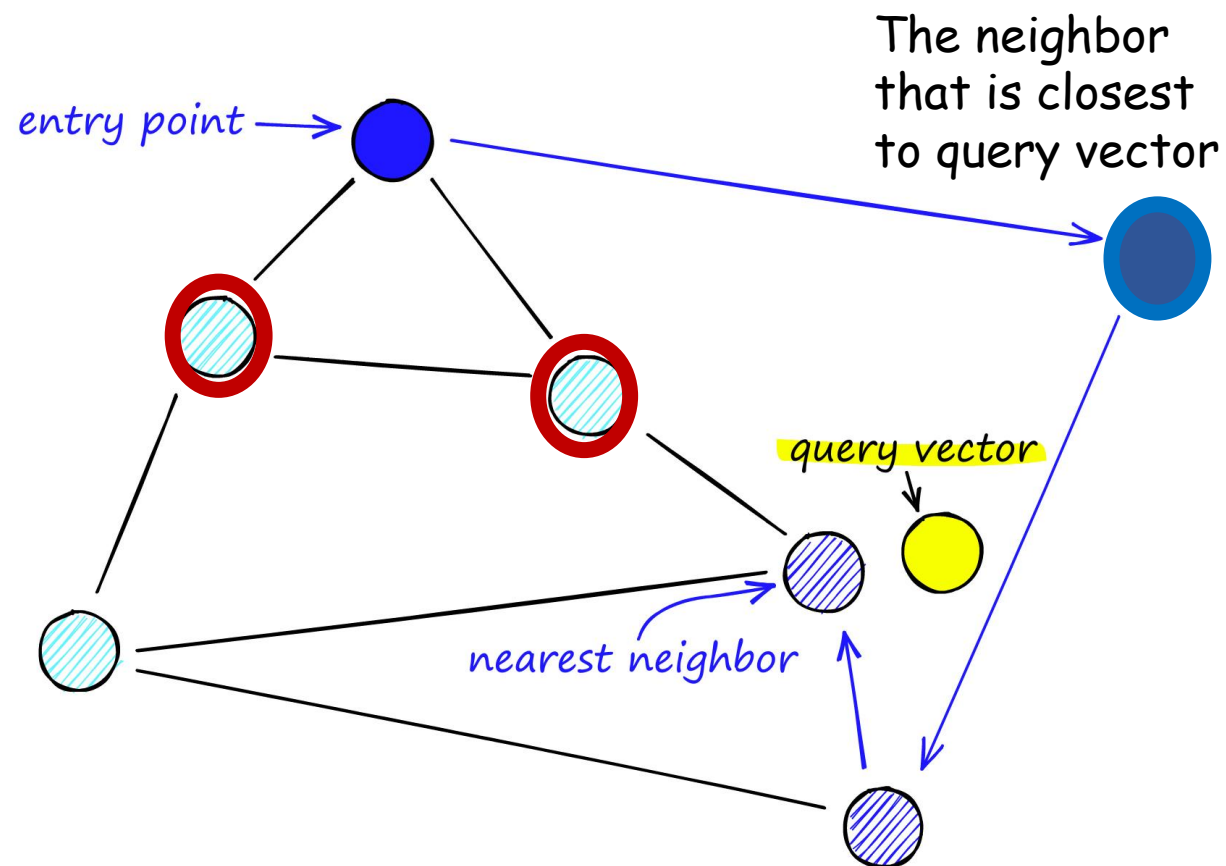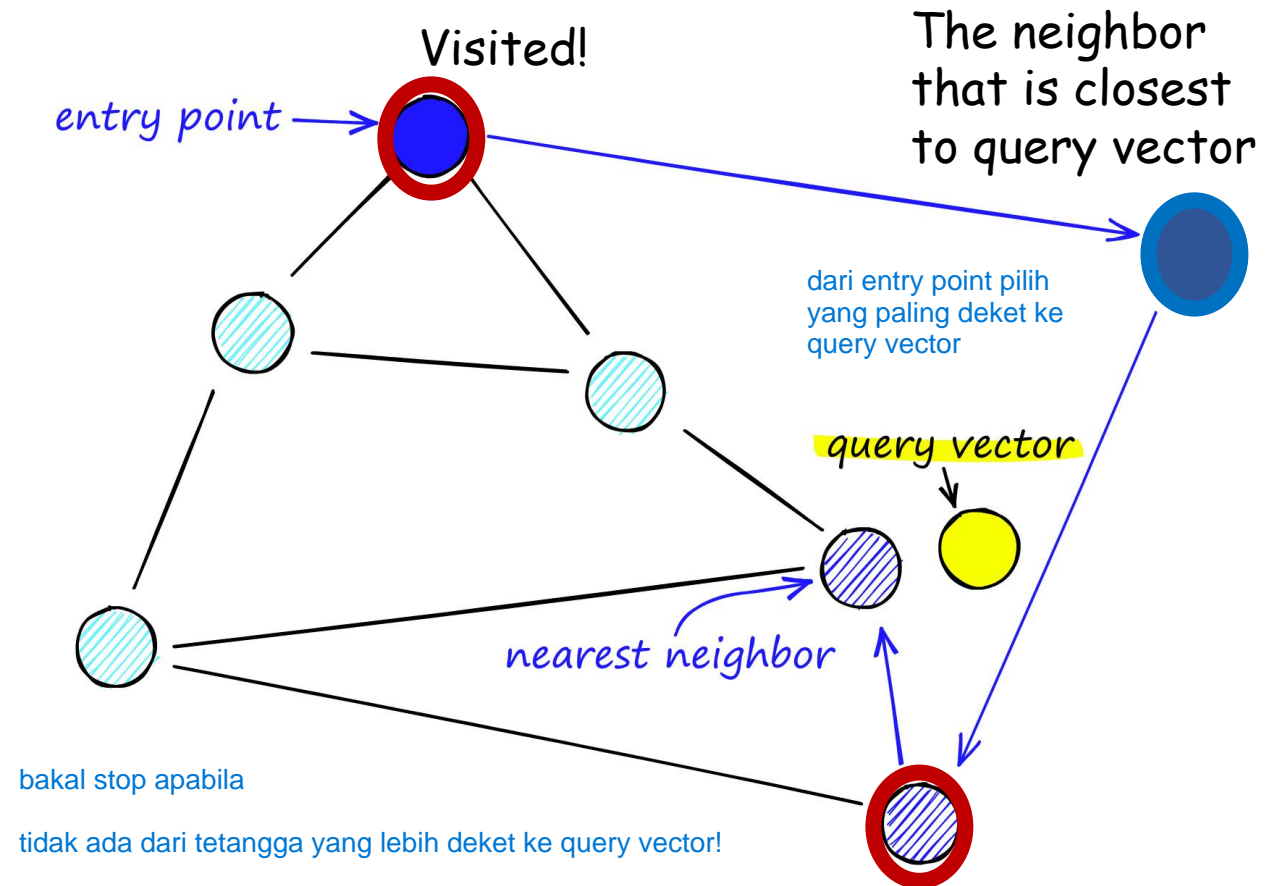
We repeat the **greedy-routing search** process of moving from vertex to vertex by identifying the nearest neighboring vertices in each friend list. **Eventually, we will find no nearer vertices than our current vertex** — this is a local minimum and acts as our **stopping condition**.

https://www.pinecone.io/learn/series/faiss/hnsw/



The neighbor that is closest to query vector

entry point

query vector

nearest neighbor

# Navigable Small World Graphs

When searching an NSW graph, we begin at a pre-defined **entry-point**. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there.
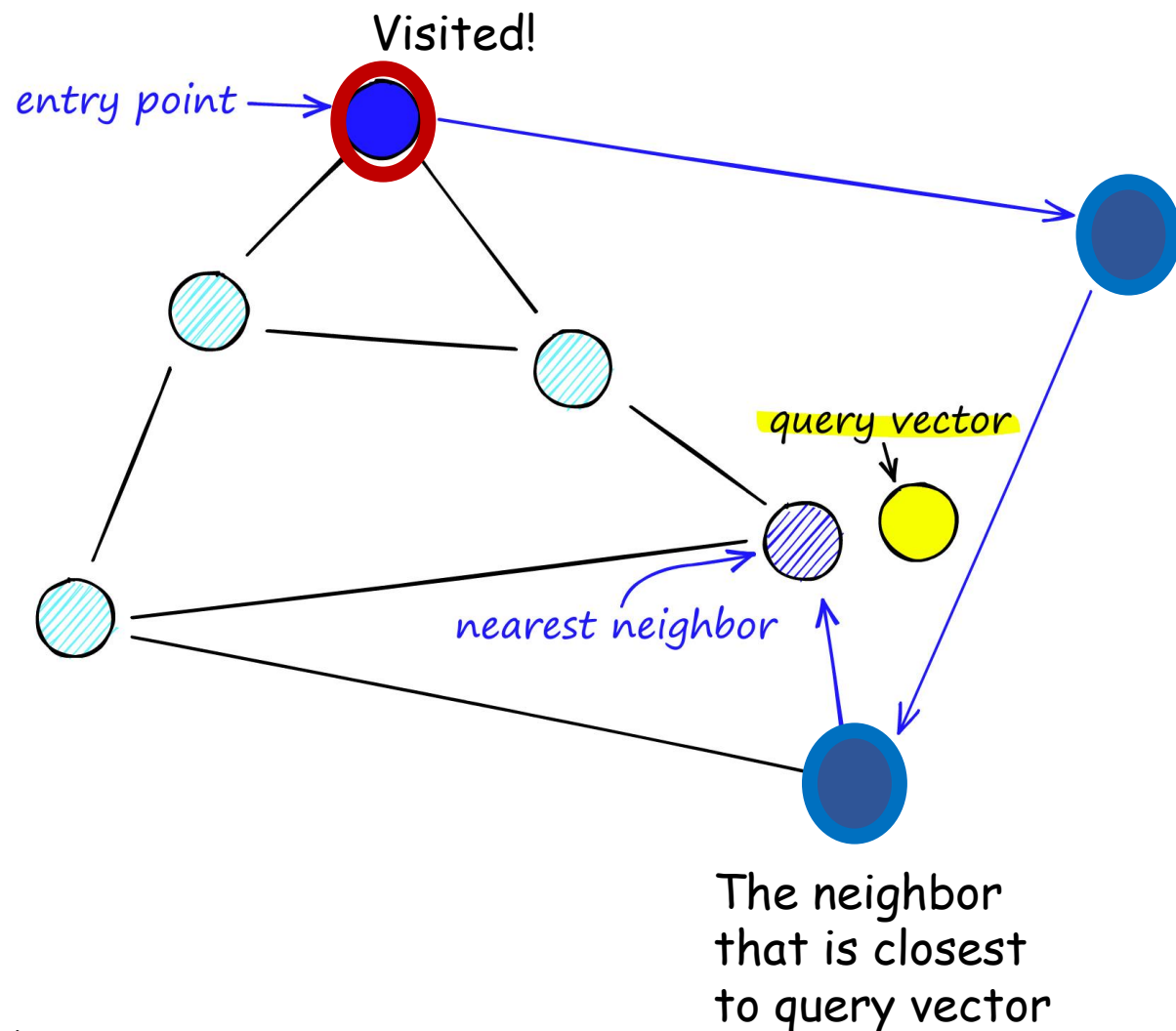
We repeat the **greedy-routing search** process of moving from vertex to vertex by identifying the nearest neighboring vertices in each friend list. **Eventually, we will find no nearer vertices than our current vertex** — this is a local minimum and acts as our **stopping condition**.

Visited!

entry point

The neighbor that is closest to query vector

dari entry point pilih yang paling deket ke query vector

query vector

nearest neighbor

bakal stop apabila

tidak ada dari tetangga yang lebih deket ke query vector!

https://www.pinecone.io/learn/series/faiss/hnsw/

# Navigable Small World Graphs

When searching an NSW graph, we begin at a pre-defined **entry-point**. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there.

We repeat the **greedy-routing search** process of moving from vertex to vertex by identifying the nearest neighboring vertices in each friend list. **Eventually, we will find no nearer vertices than our current vertex** — this is a local minimum and acts as our **stopping condition**.

https://www.pinecone.io/learn/series/faiss/hnsw/

Visited!

entry point

query vector

nearest neighbor

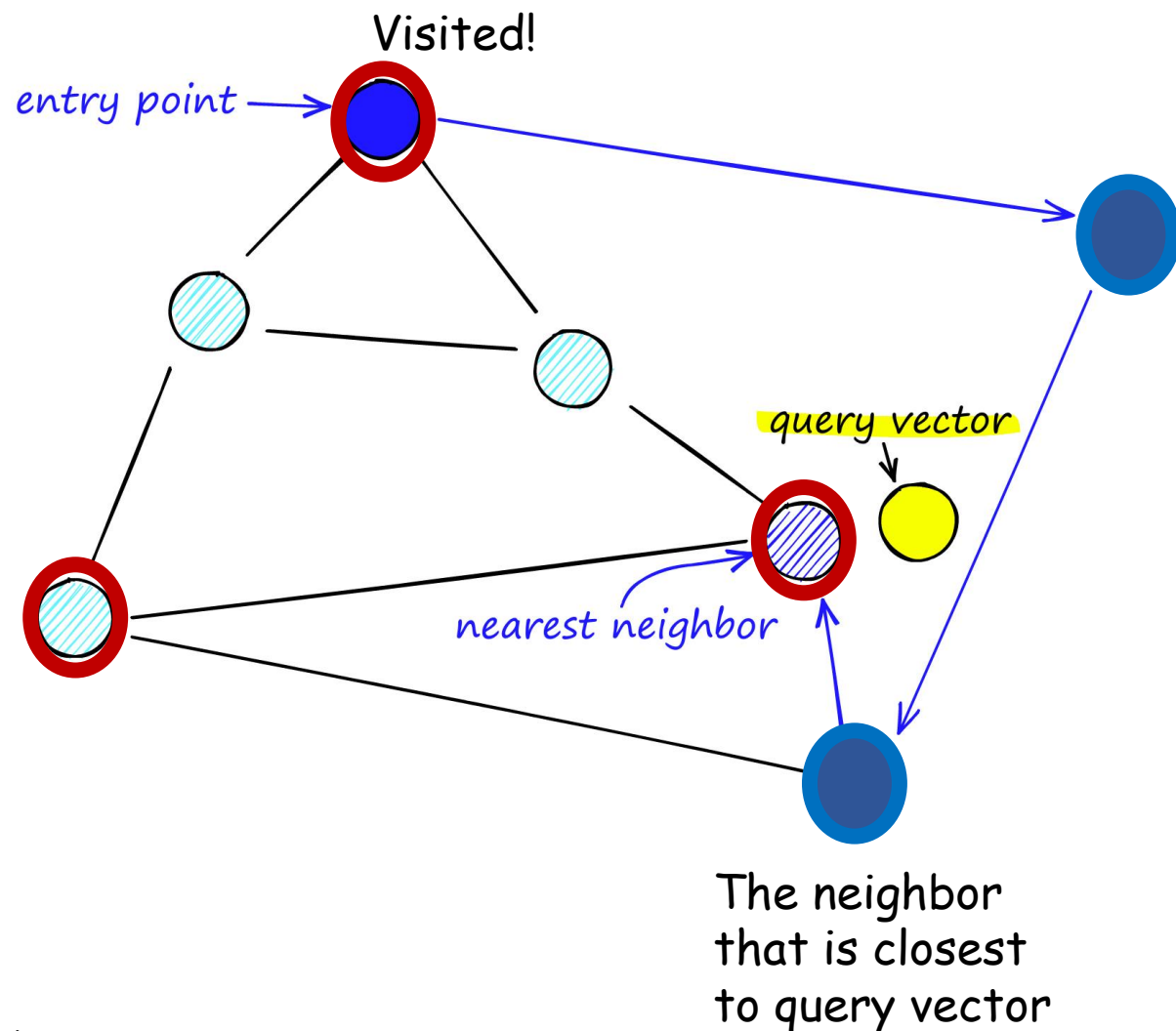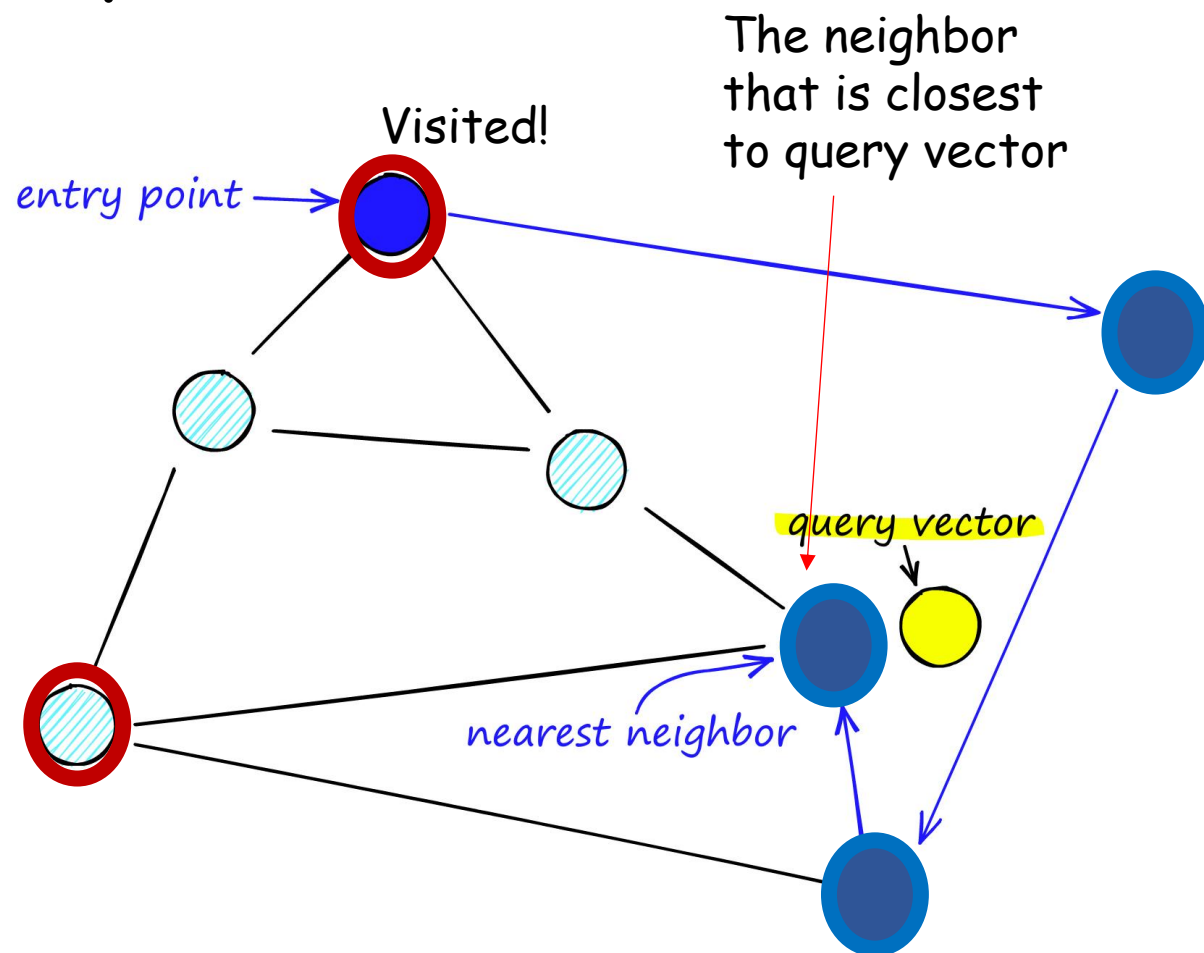The neighbor that is closest to query vector

# Navigable Small World Graphs

When searching an NSW graph, we begin at a pre-defined **entry-point**. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there.
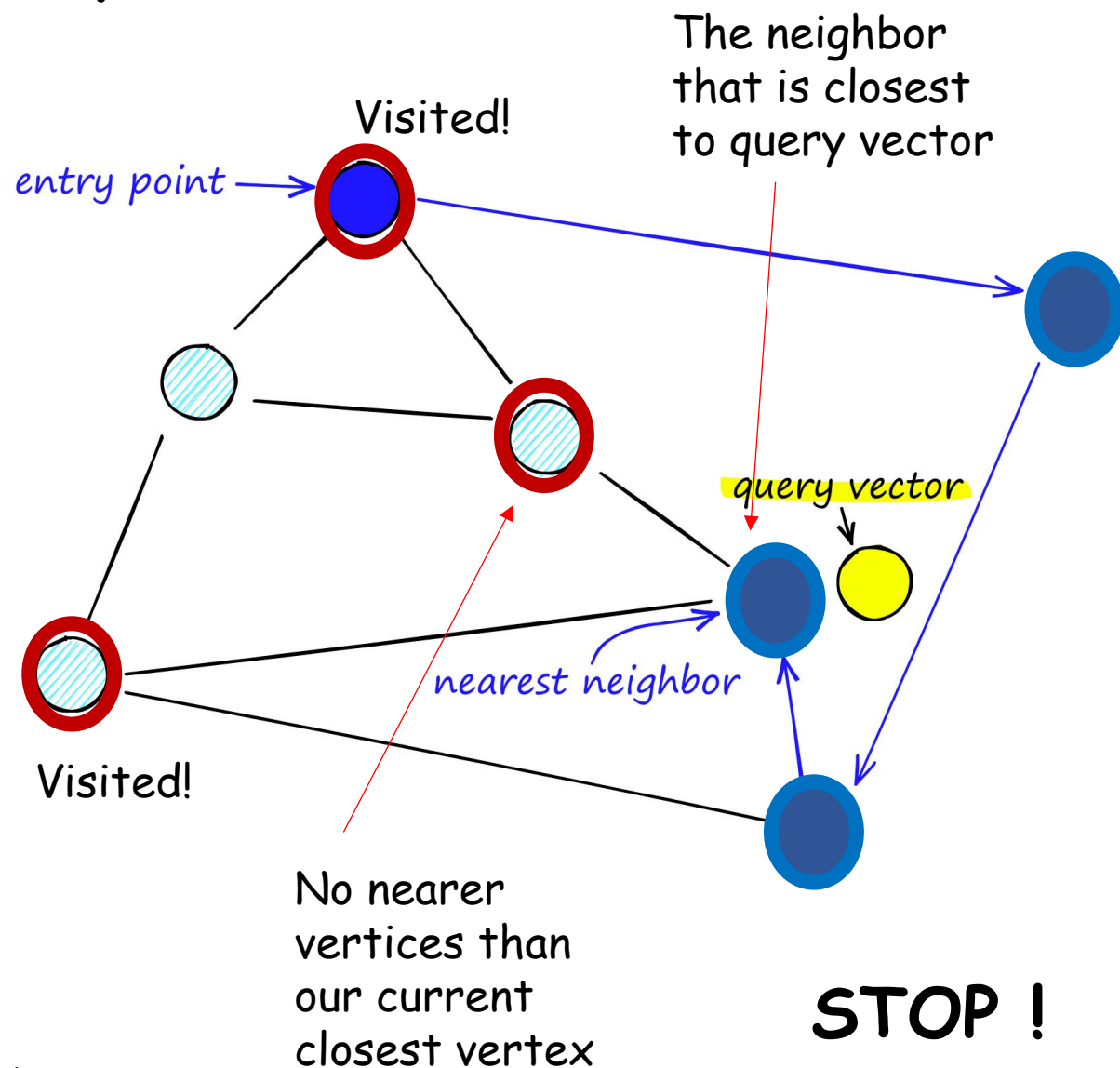
We repeat the **greedy-routing search** process of moving from vertex to vertex by identifying the nearest neighboring vertices in each friend list. **Eventually, we will find no nearer vertices than our current vertex** — this is a local minimum and acts as our **stopping condition**.

https://www.pinecone.io/learn/series/faiss/hnsw/

Visited!

entry point

query vector

nearest neighbor

The neighbor that is closest to query vector

# Navigable Small World Graphs

When searching an NSW graph, we begin at a pre-defined **entry-point**. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there.
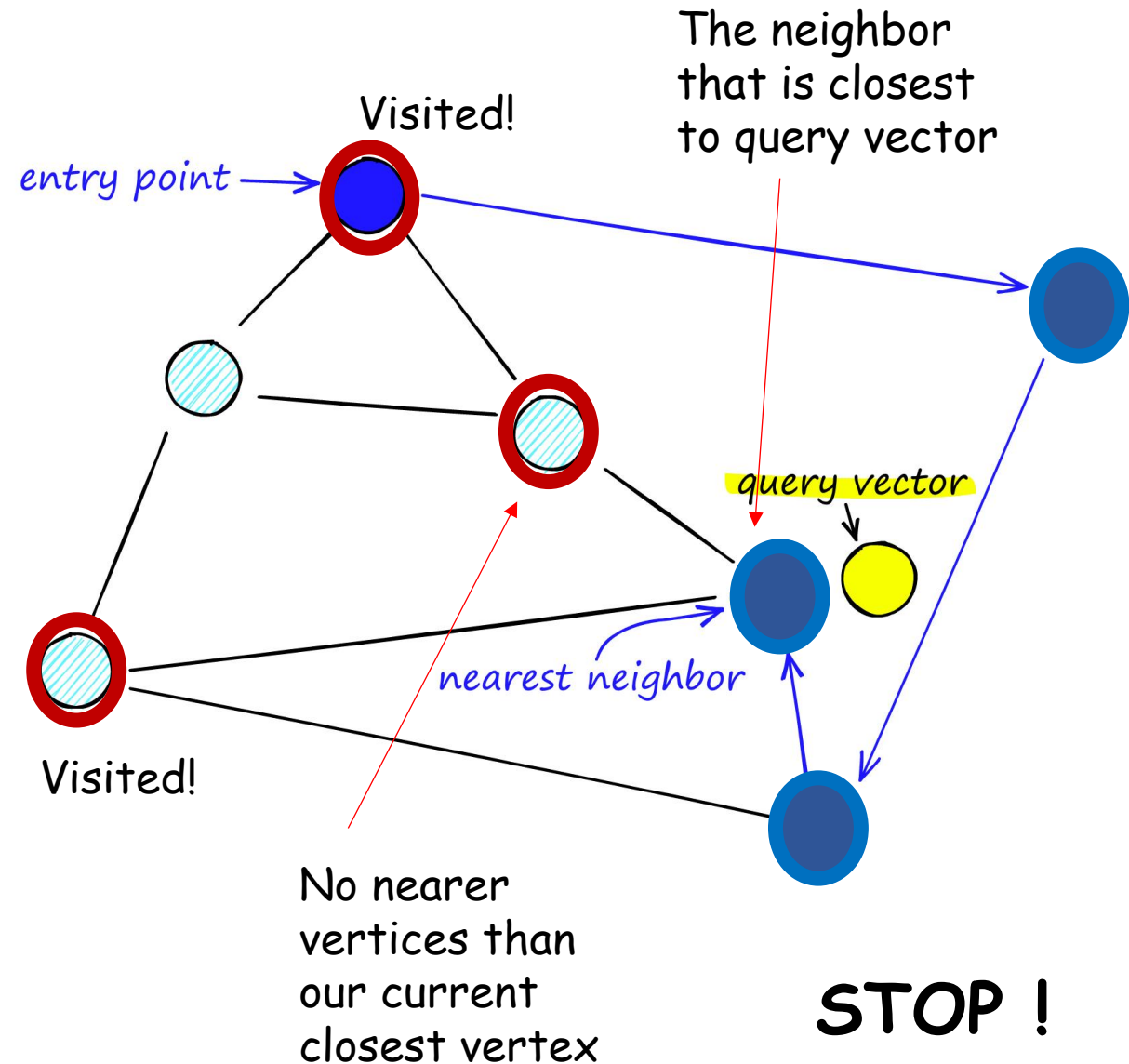
We repeat the **greedy-routing search** process of moving from vertex to vertex by identifying the nearest neighboring vertices in each friend list. **Eventually, we will find no nearer vertices than our current vertex** — this is a local minimum and acts as our **stopping condition**.

https://www.pinecone.io/learn/series/faiss/hnsw/

The neighbor that is closest to query vector

Visited!

entry point

query vector

nearest neighbor

# Navigable Small World Graphs

When searching an NSW graph, we begin at a pre-defined **entry-point**. This entry point connects to several nearby vertices. We identify which of these vertices is the closest to our query vector and move there.

We repeat the **greedy-routing search** process of moving from vertex to vertex by identifying the nearest neighboring vertices in each friend list. **Eventually, we will find no nearer vertices than our current vertex** — this is a local minimum and acts as our **stopping condition**.

https://www.pinecone.io/learn/series/faiss/hnsw/



The neighbor that is closest to query vector

Visited!

entry point

query vector

nearest neighbor

Visited!

No nearer vertices than our current closest vertex

STOP !

# Navigable Small World Graphs

Secara praktis, proses seperti ini dilakukan secara berulang-ulang untuk mendapatkan **K buah nearest vectors**.

Setiap run, kita "entry point" bisa diacak.



The neighbor that is closest to query vector

Visited!

entry point

query vector

nearest neighbor

Visited!

No nearer vertices than our current closest vertex

STOP !

# Navigable Small World Graphs

Contoh lain:

# Navigable Small World Graphs



**Entry point**
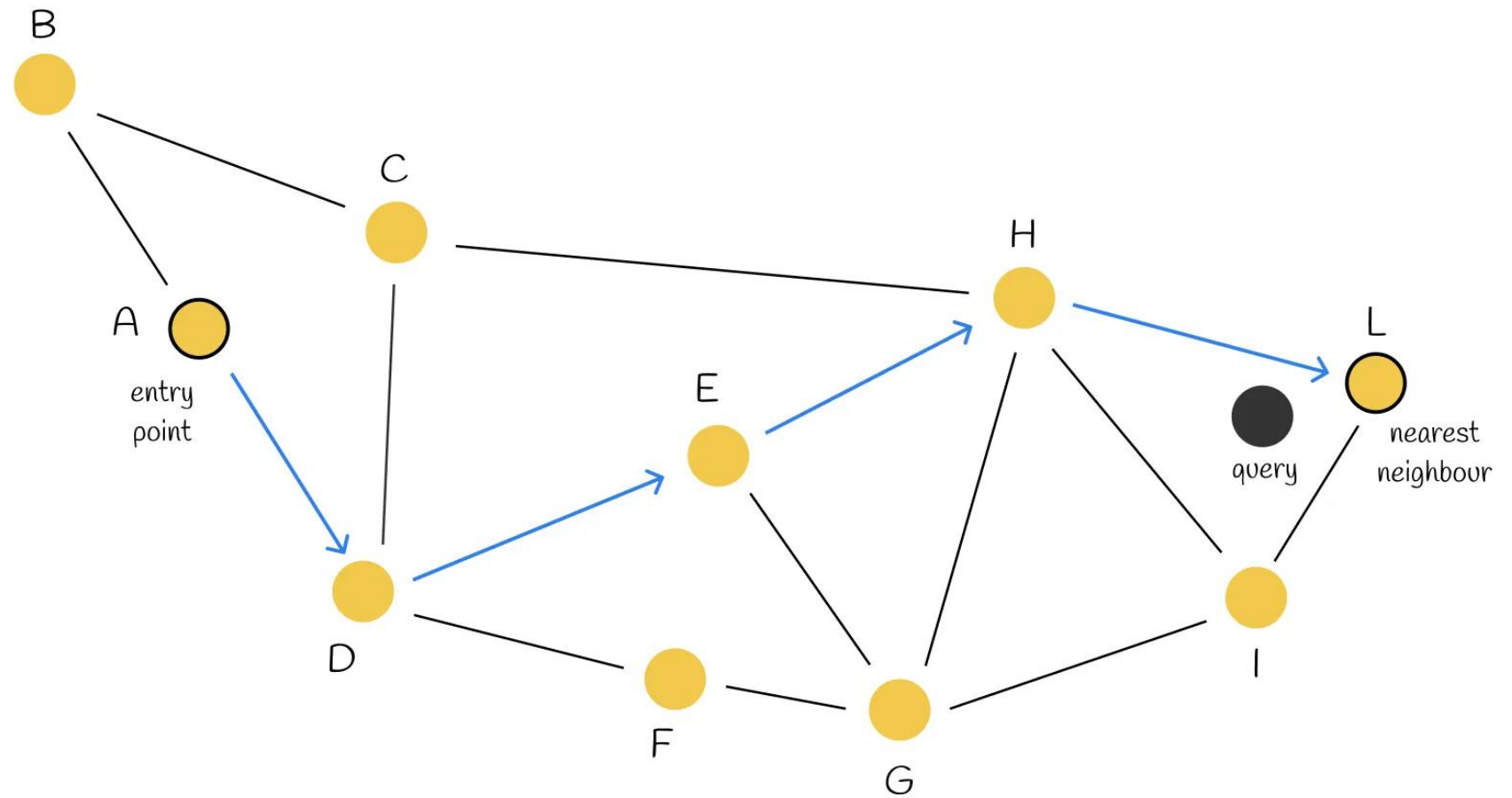
**query**

This greedy strategy **does not guarantee** that it will find the exact nearest neighbor as the method uses only local information at the current step to take decisions.
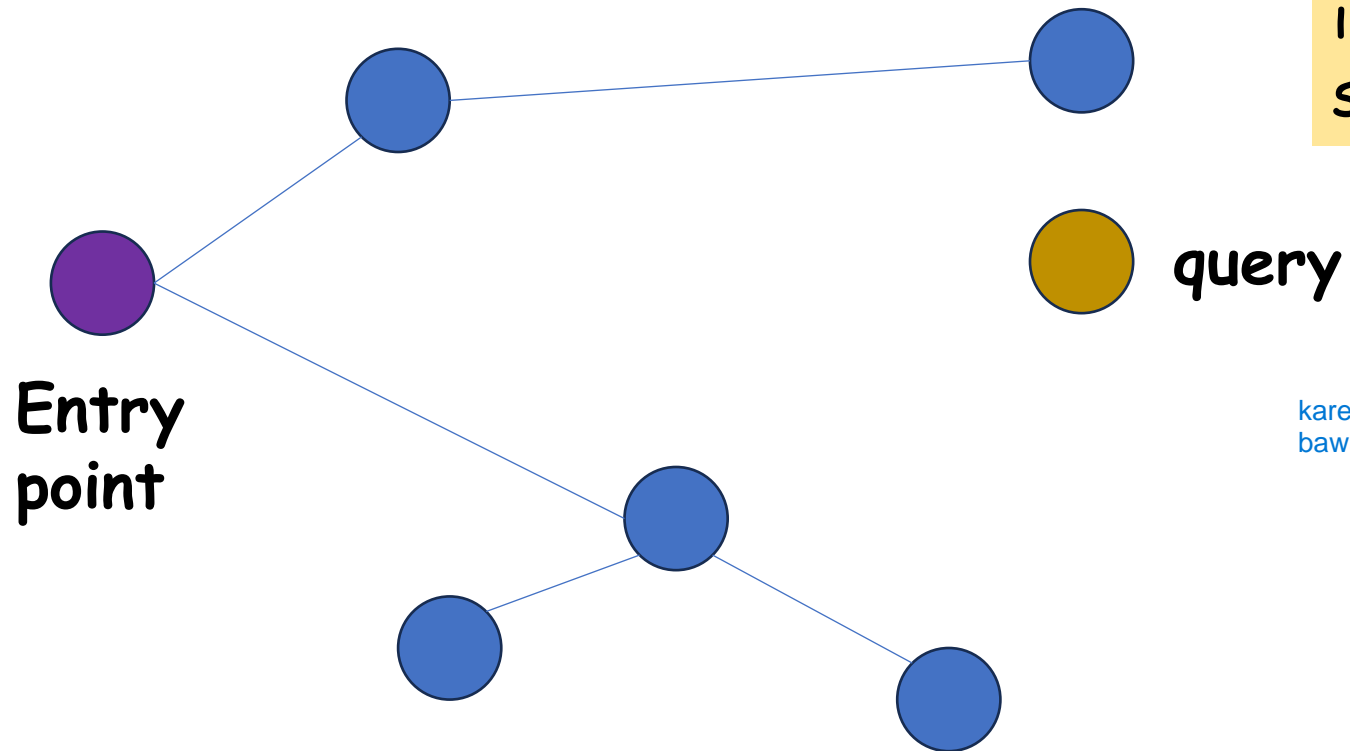
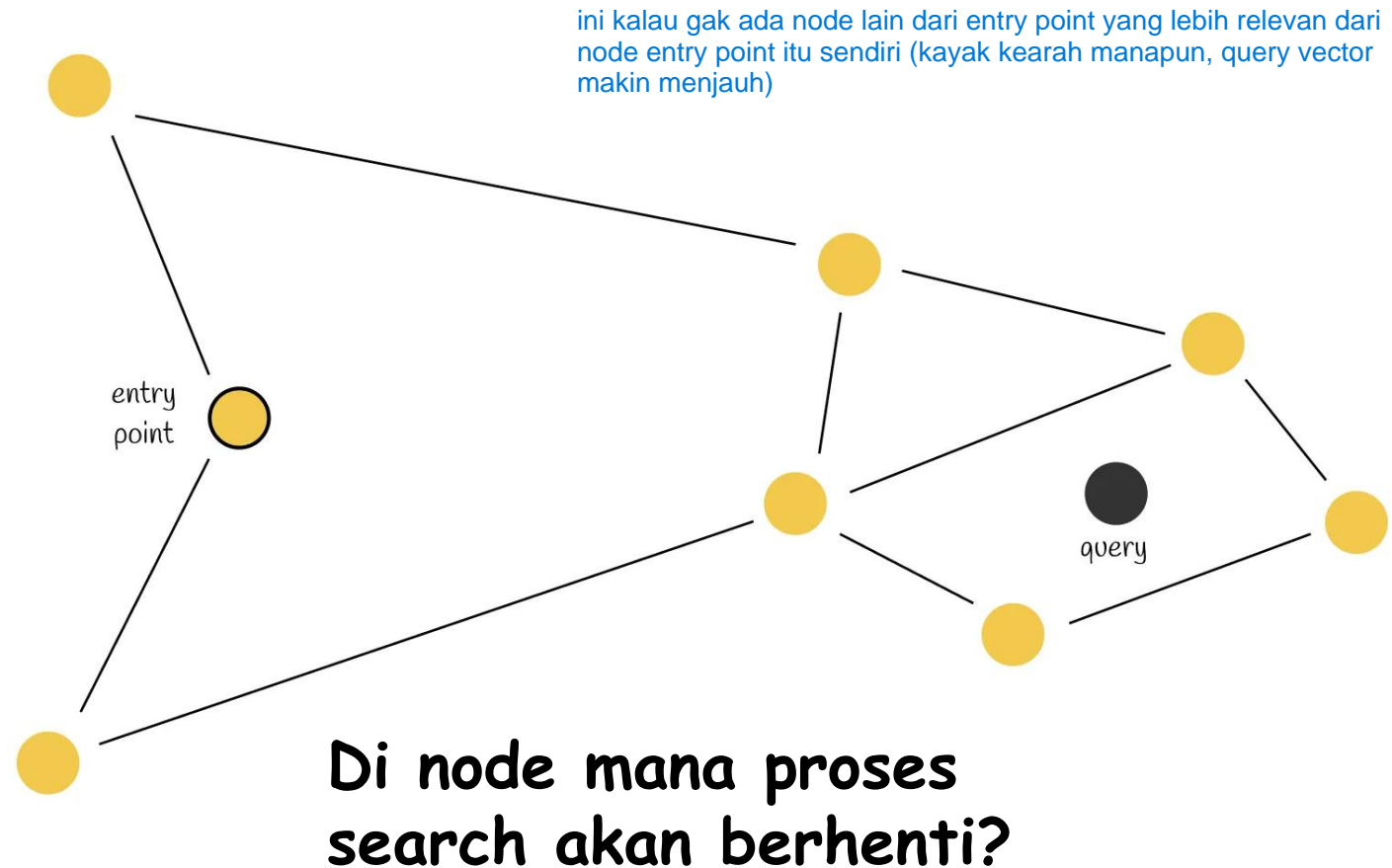karena greedy, yang terdekat itu dari query vector itu yang entry point kearah bawah

# Navigable Small World Graphs

Problem:

**Early stopping** is one of the problems of the algorithm.

For the most part, this might happen when the starting region has too many **low-degree vertices**.

ini kalau gak ada node lain dari entry point yang lebih relevan dari node entry point itu sendiri (kayak kearah manapun, query vector makin menjauh)

entry point

query

Di node mana proses search akan berhenti?

# Indexing
## (Constructing Navigable Small World Graphs)

# How to construct NSW? (indexing)

sbelumnya itu search, sekarang itu indexing

- **Sebuah node = sebuah vektor**

- **Node A dekat dengan Node B = vektor A dan vektor B "similar" (cosine sim., Euclidean dist., dsb.)**

M itu tetangga terdekat

- Misal, **M** adalah banyaknya tetangga terdekat untuk node yang baru "masuk".

- Pada setiap iterasi, sebuah node baru ditambahkan ke dalam index dan dihubungkan dengan **M** buah tetangga terdekat.

# Constructing Navigable Small World Graphs

**M = 2**

# Constructing Navigable Small World Graphs

**M = 2**

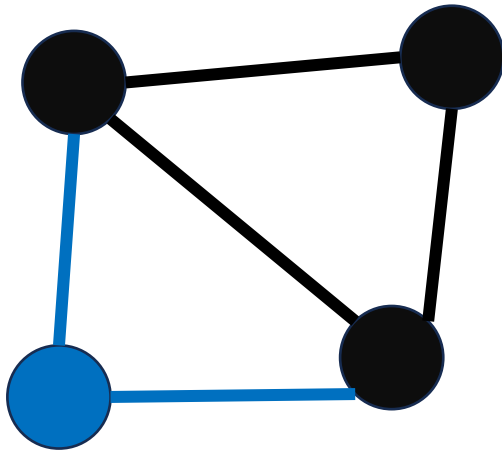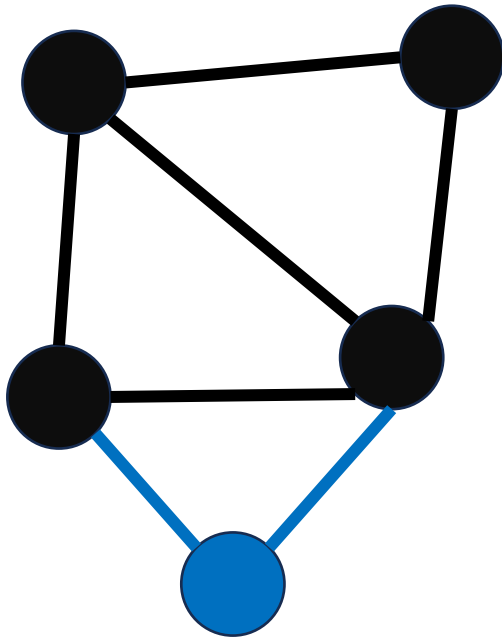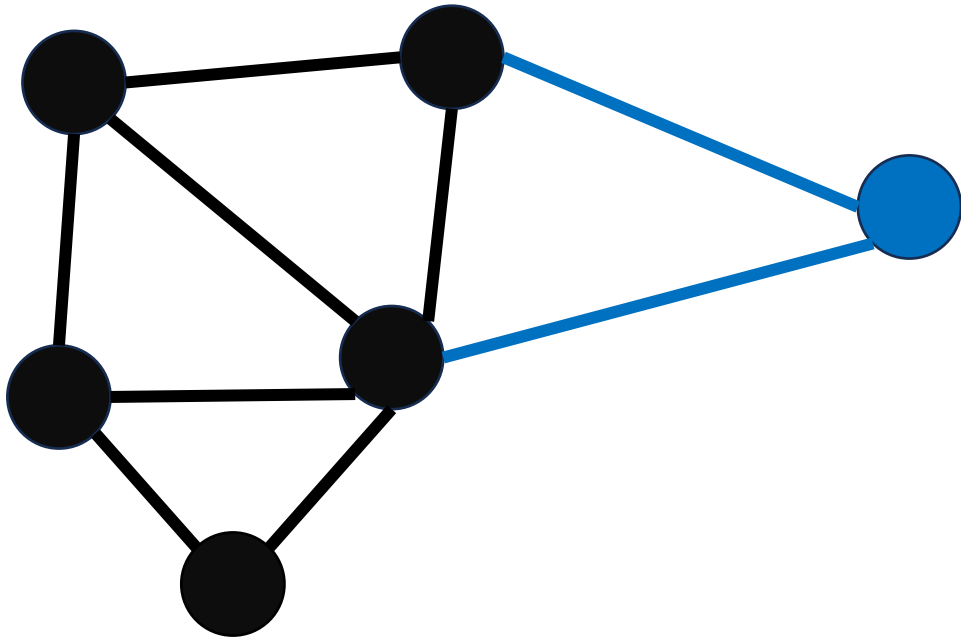# Constructing Navigable Small World Graphs

**M = 2**



**Node biru:** vektor yang baru dimasukkan ke index
**Sisi biru:** sisi yang baru terbentuk

# Constructing Navigable Small World Graphs

M = 2



Node biru: vektor yang baru dimasukkan ke index
Sisi biru: sisi yang baru terbentuk

# Constructing Navigable Small World Graphs

M = 2



**Node biru:** vektor yang baru dimasukkan ke index
**Sisi biru:** sisi yang baru terbentuk
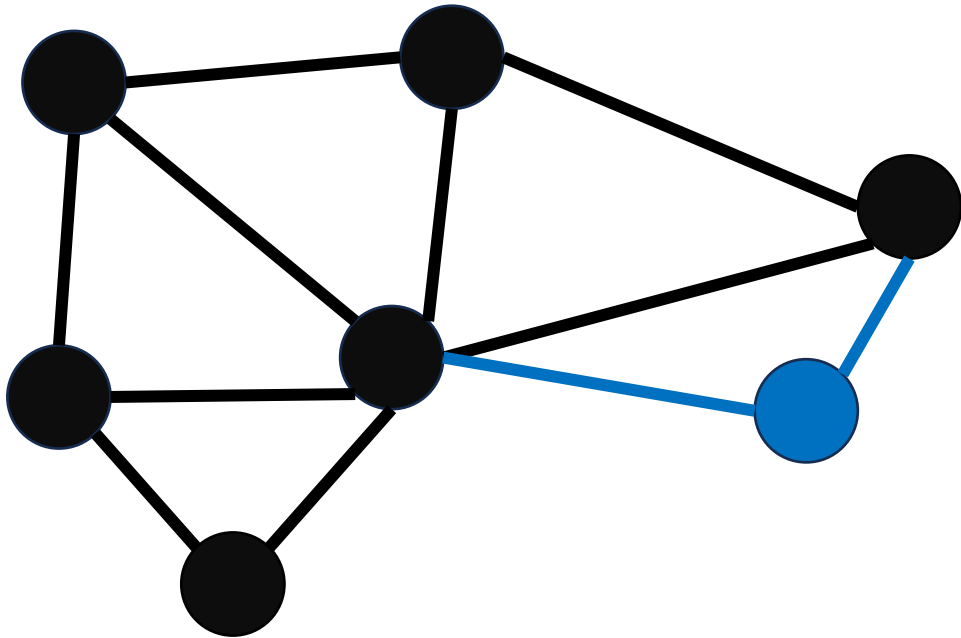
# Constructing Navigable Small World Graphs

**M = 2**



**Node biru:** vektor yang baru dimasukkan ke index
**Sisi biru:** sisi yang baru terbentuk

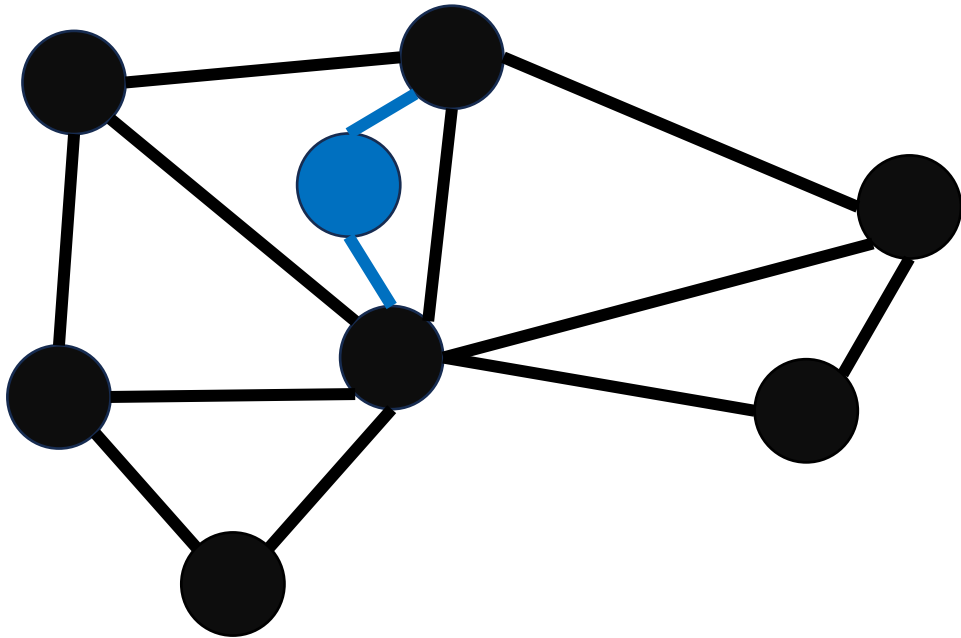# Constructing Navigable Small World Graphs

**M = 2**



**Node biru:** vektor yang baru dimasukkan ke index
**Sisi biru:** sisi yang baru terbentuk

# Constructing Navigable Small World Graphs

**M = 2**



**Node biru:** vektor yang baru dimasukkan ke index
**Sisi biru:** sisi yang baru terbentuk
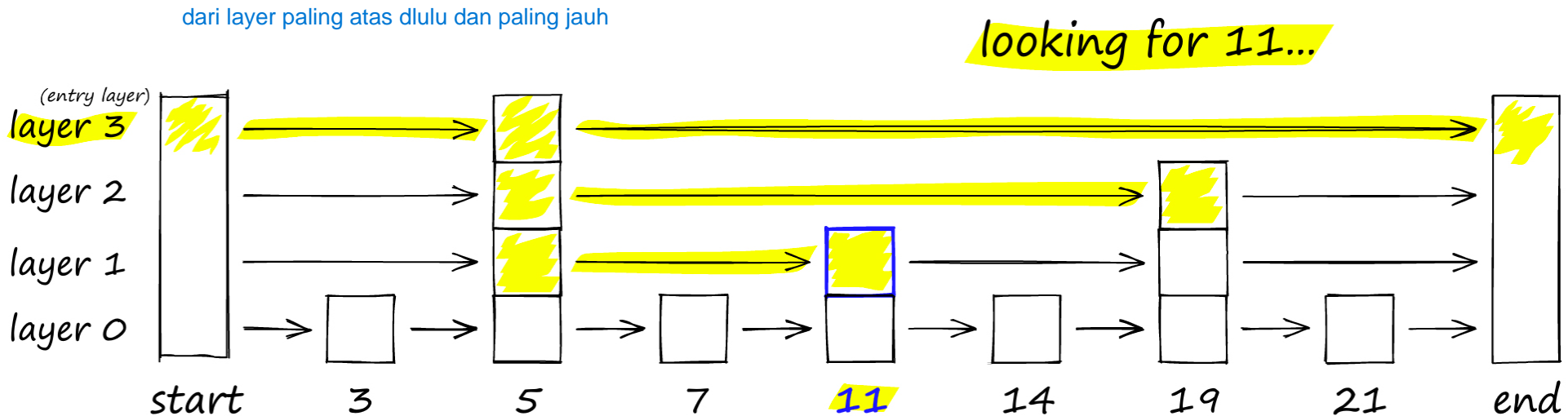
# Make them "Hierarchical"

# Hierarchical NSW = HNSW

## Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

Yu. A. Malkov, D. A. Yashunin

**Abstract** — We present a new approach for the approximate K-nearest neighbor search based on navigable small world graphs with controllable hierarchy (Hierarchical NSW, HNSW). The proposed solution is fully graph-based, without any need for additional search structures, which are typically used at the coarse search stage of the most proximity graph techniques. Hierarchical NSW incrementally builds a multi-layer structure consisting from hierarchical set of proximity graphs (layers) for nested subsets of the stored elements. The maximum layer in which an element is present is selected randomly with an

# Probability Skip List – Fast Search



dari layer paling atas dlulu dan paling jauh

looking for 11...

(entry layer)
layer 3
layer 2
layer 1
layer 0

start   3   5   7   11   14   19   21   end

To search a skip list, we start at the highest layer with the longest 'skips' and move along the edges towards the right (below). If we find that the current node 'key' is greater than the key we are searching for — we know we have overshot our target, so we move down to previous node in the next level.
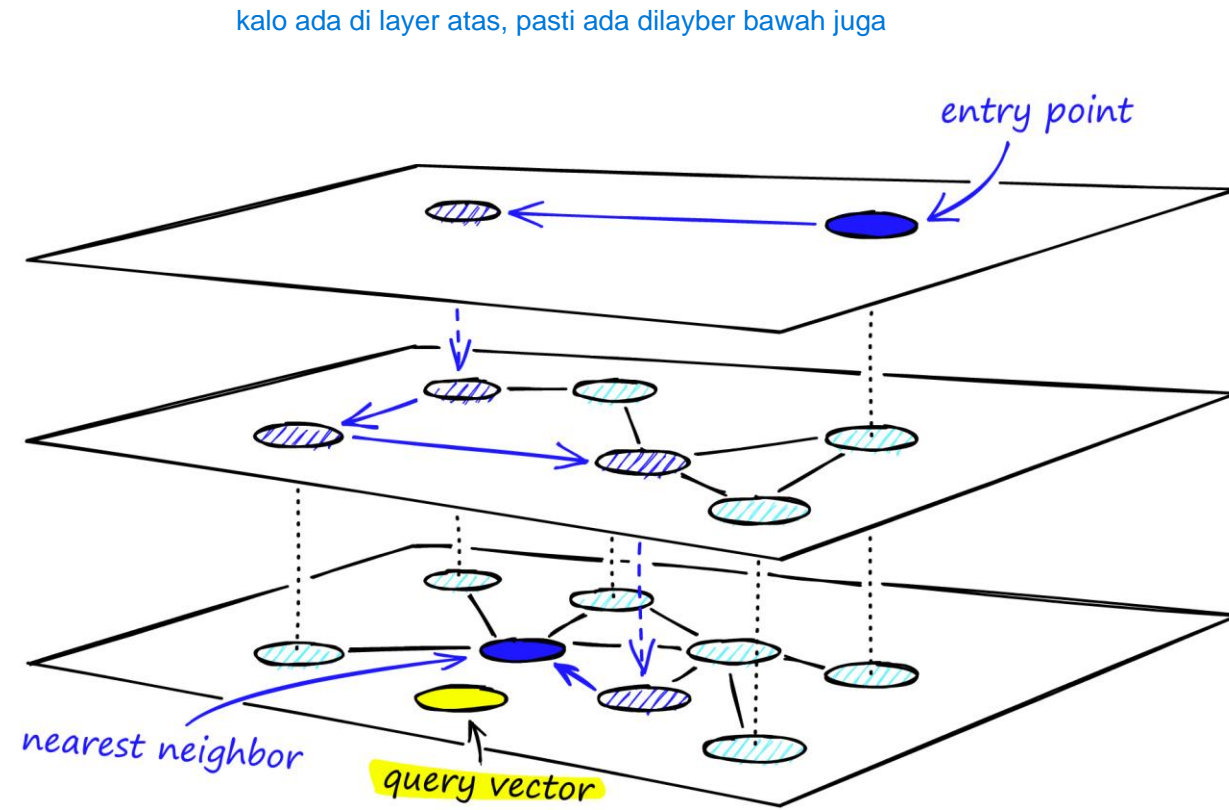
A skip list has the main parameter $p$ which defines **the probability of an element appearing in several lists**. If an element appears in **layer** $i$, then the probability that it will appear in **layer** $i$ + 1 is equal to $p$.

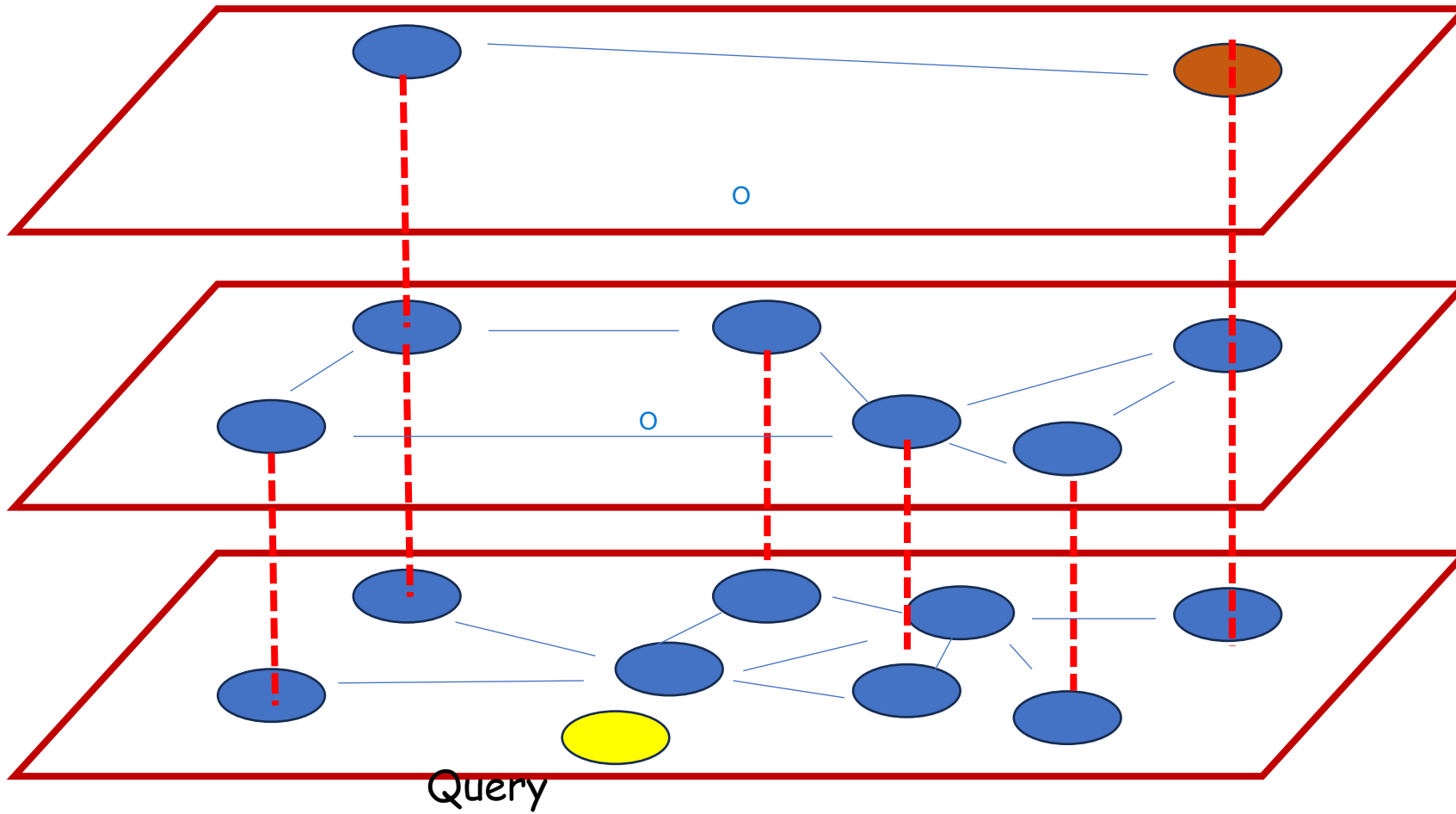Insert & search: **On average O(Log N)**
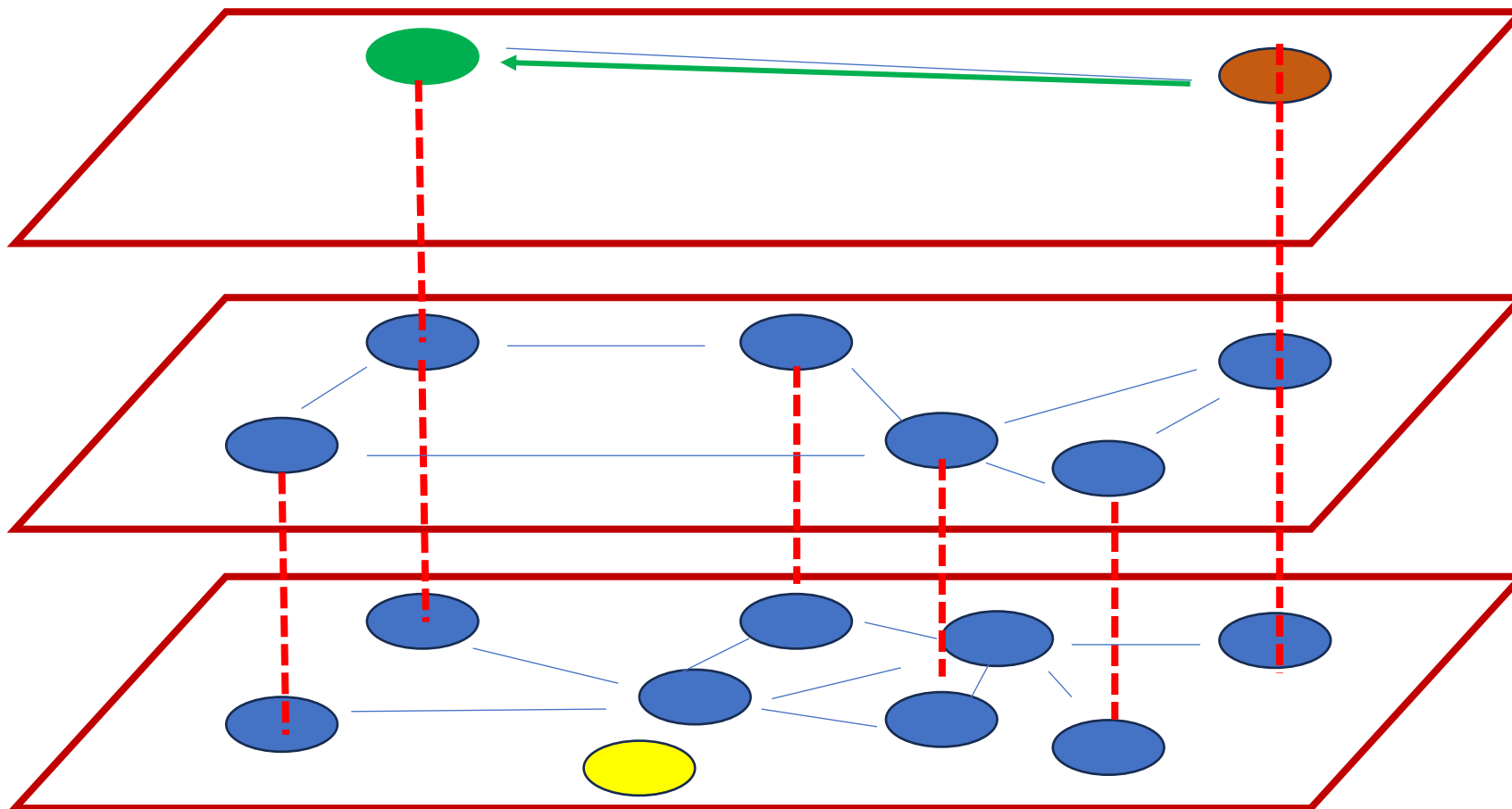
# Hierarchical Navigable Small World Graphs

During the search, we enter the top layer. These vertices will tend to be **higher-degree vertices** (with links separated across multiple layers), meaning that we, by default, start in the **zoom-in phase.**
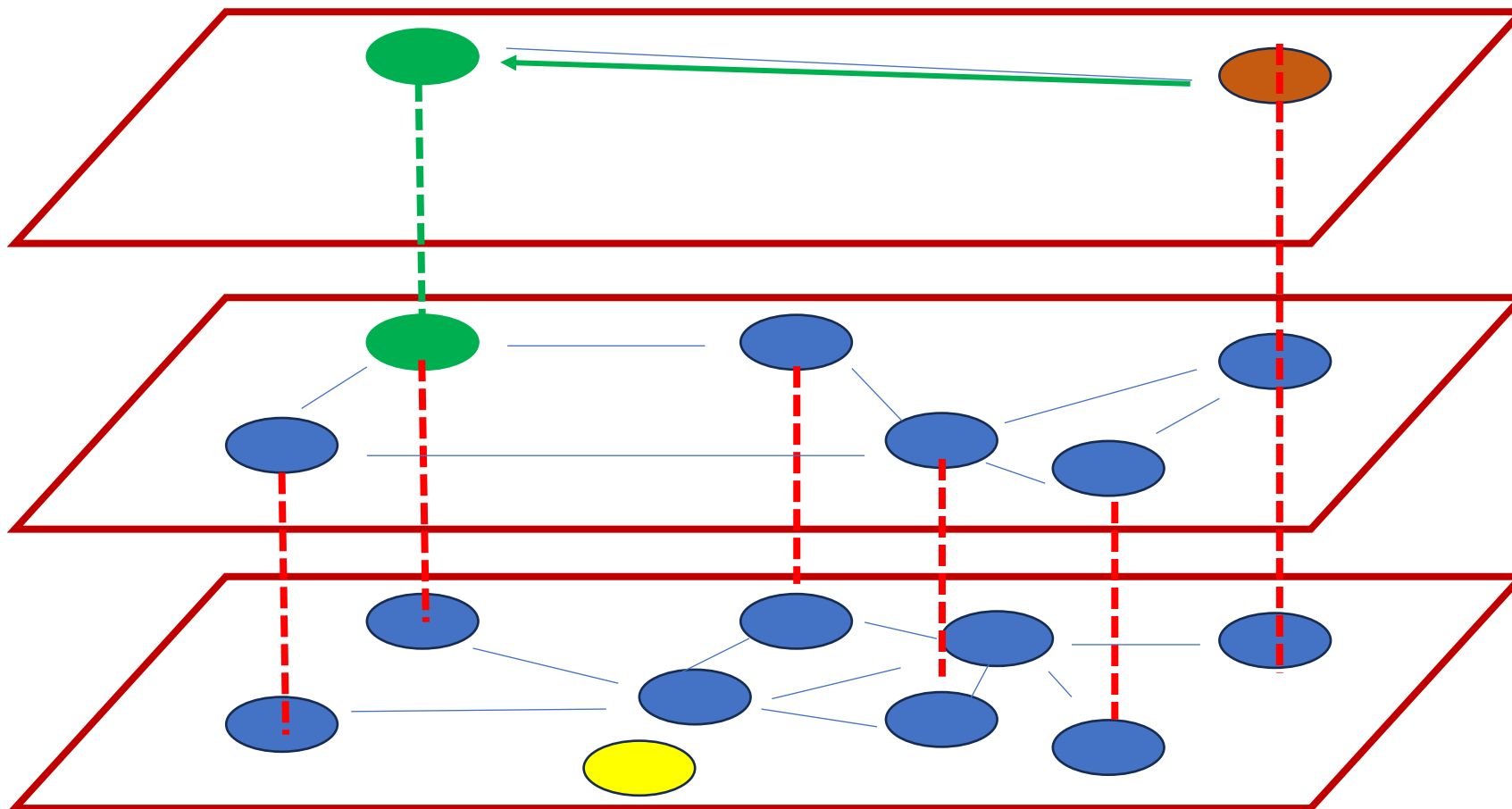
We move to the nearest vertex until we find a local minimum. Unlike NSW, **at this point, we shift to the current vertex in a lower layer and begin searching again**. We repeat this process until finding the local minimum of our bottom layer — layer 0.



kalo ada di layer atas, pasti ada dilayber bawah juga

entry point

nearest neighbor

query vector

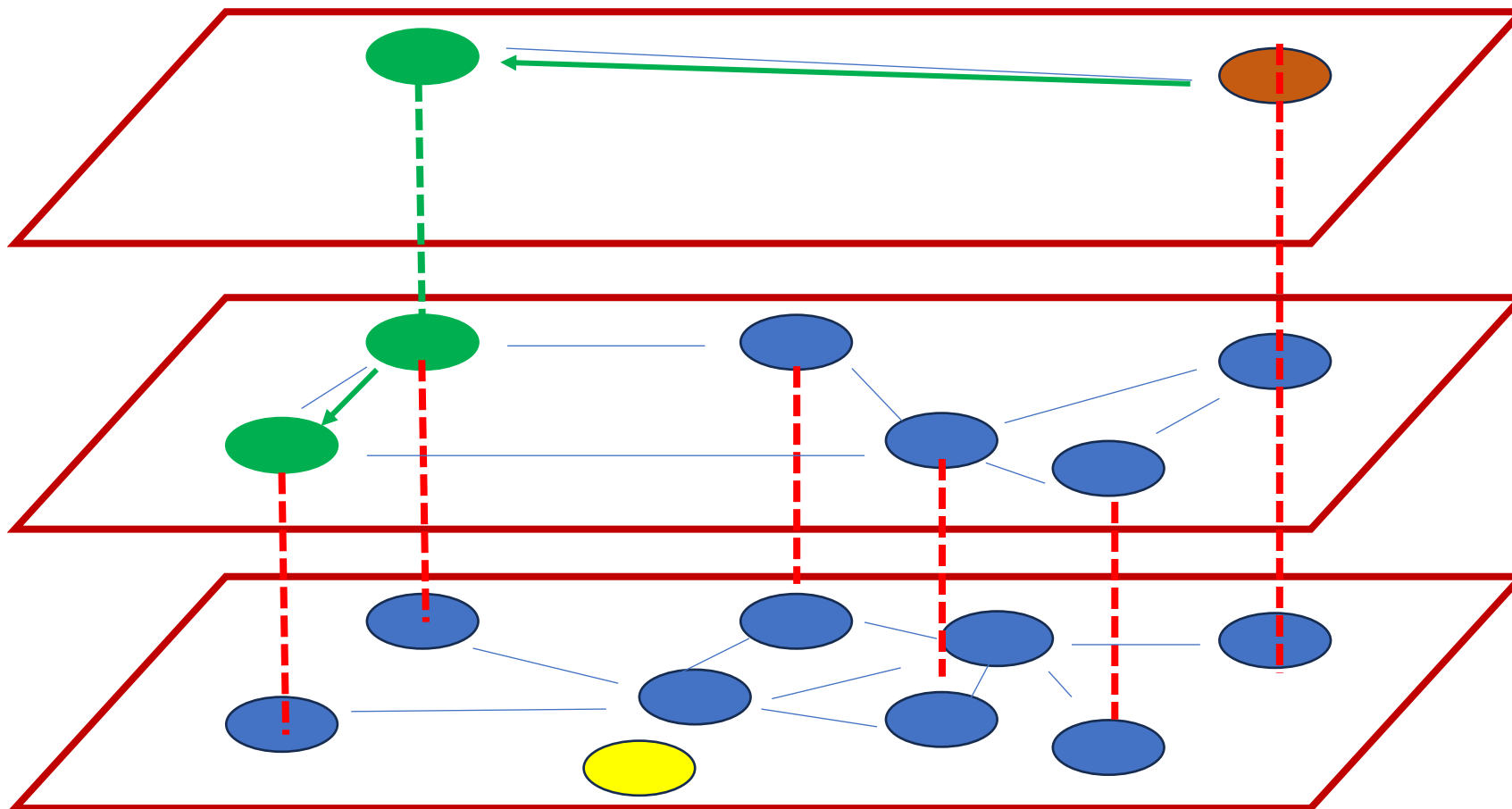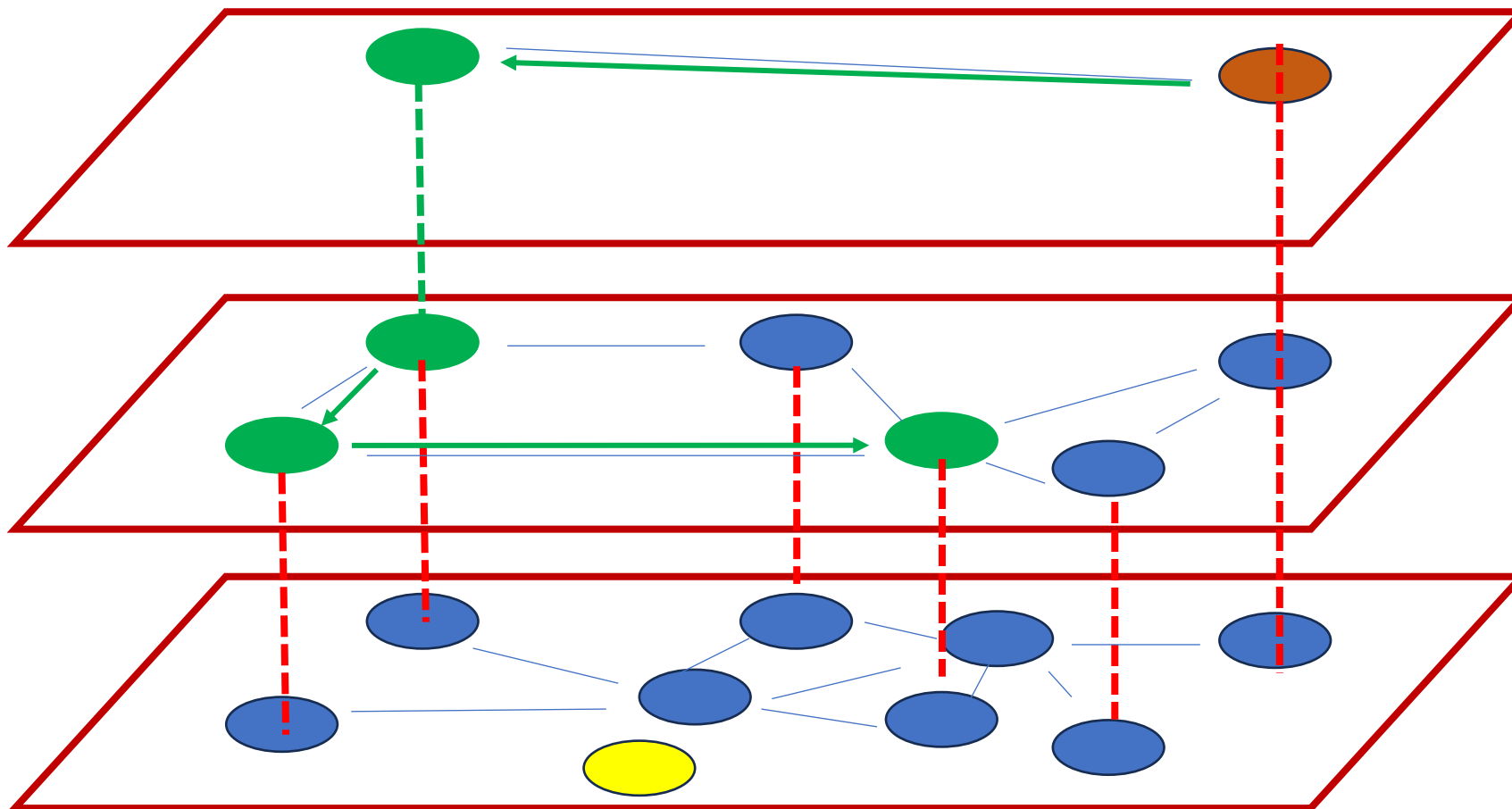sisi di setiap layer bisa beda-beda

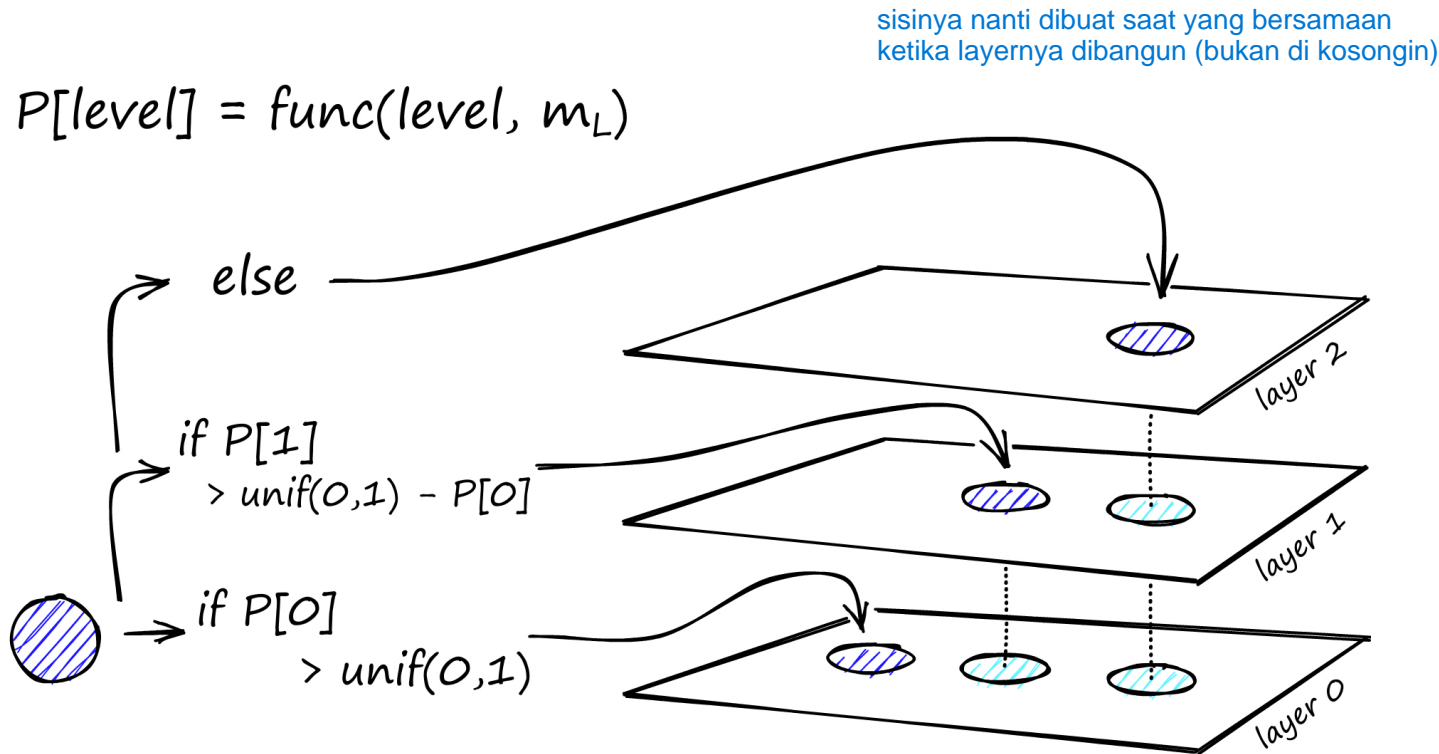https://www.pinecone.io/learn/series/faiss/hnsw/

ngitung jaraknya itu query vector di copy per layer

Query
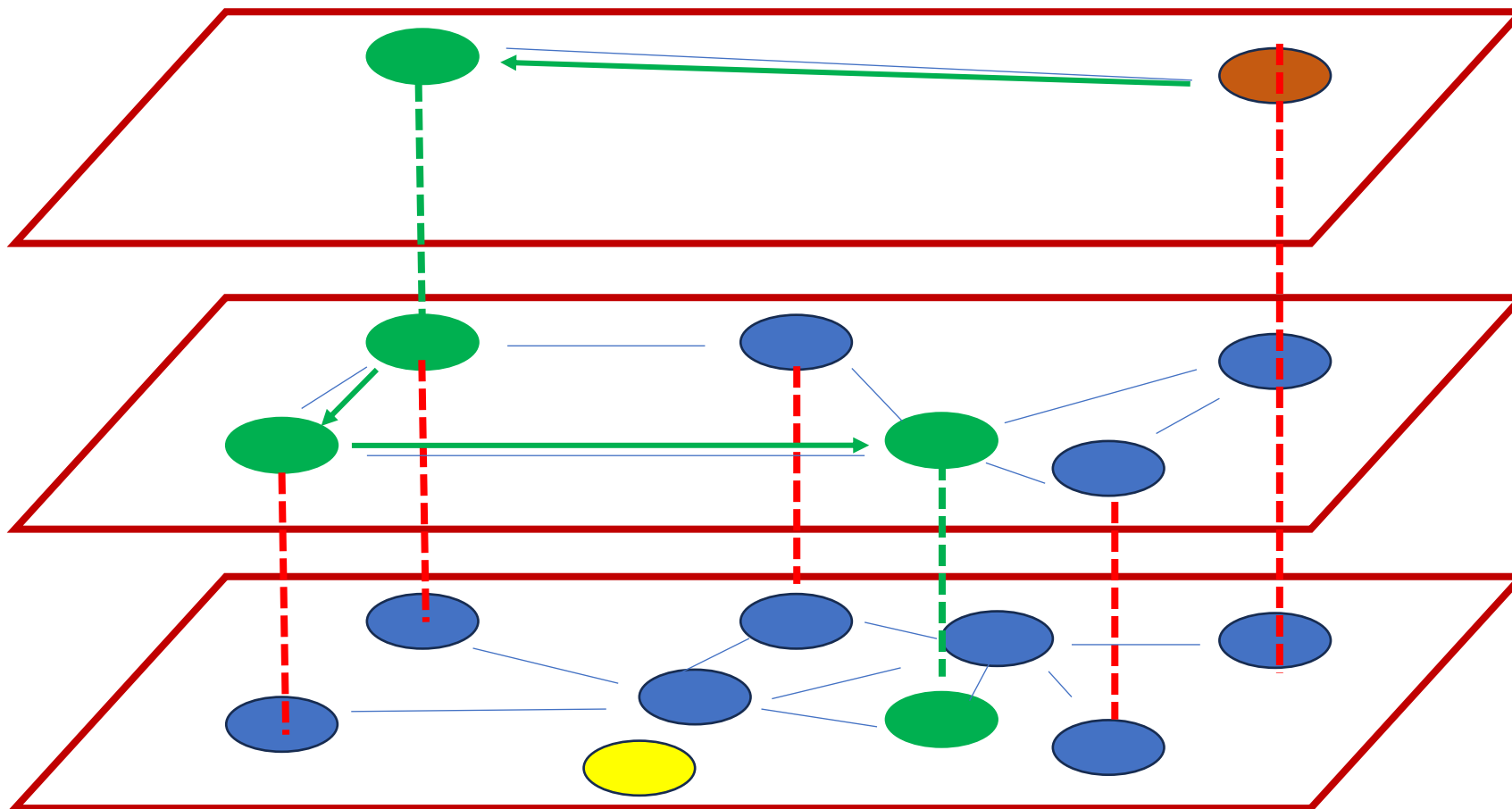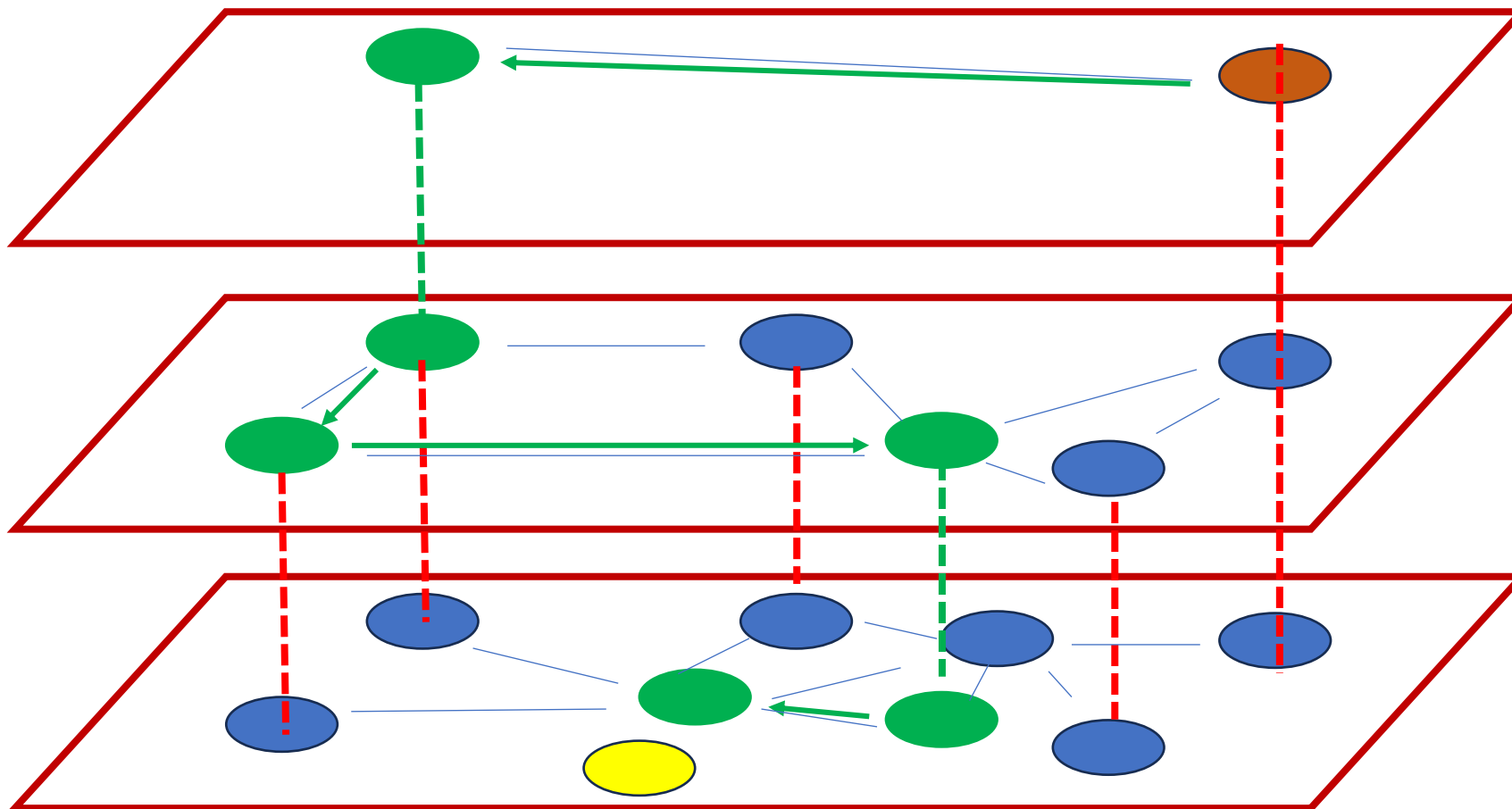
# How to construct HNSW graph?

During graph construction, vectors are iteratively inserted one-by-one. The number of layers is represented by **parameter L**.

The probability of a vector insertion at a given layer is given by a **exponentially decaying probability** function normalized by the *'level multiplier'* $m_L$

sisinya nanti dibuat saat yang bersamaan
ketika layernya dibangun (bukan di kosongin)

The probability function is repeated for each layer (other than layer 0).

The vector is added to its insertion layer and every layer below it.

P[level] = func(level, $m_L$)

else

if P[1]
> unif(0,1) – P[0]

if P[0]
> unif(0,1)

layer 2

layer 1

layer 0

# Constructing HNSW

After a node is assigned the level number (**value 1**), the algorithm starts from the upper layer and greedily finds the nearest node.

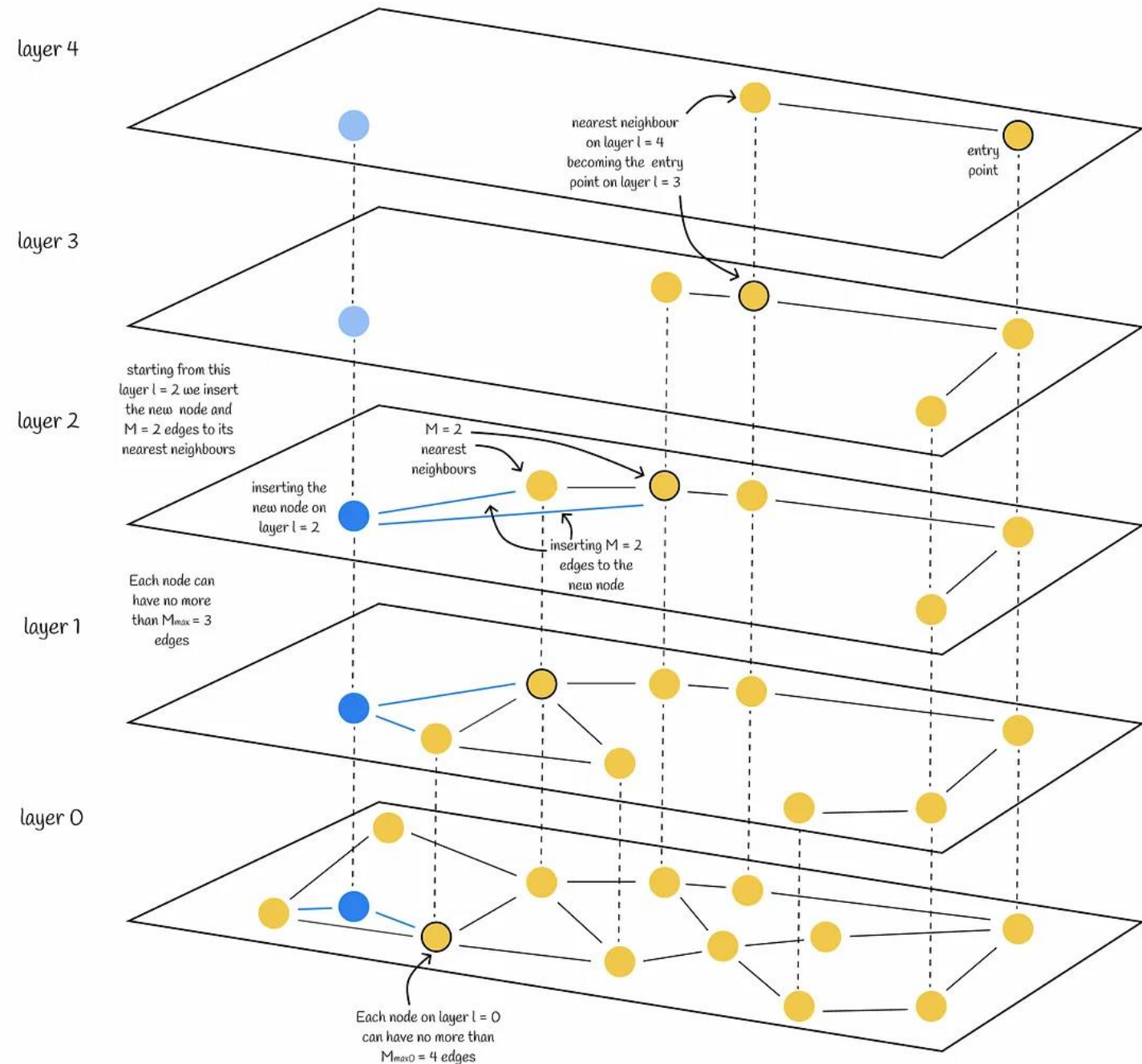The found node is then used as an entry point to the next layer and the search process continues.

Once the **layer l** is reached, the insertion proceeds to the second step.
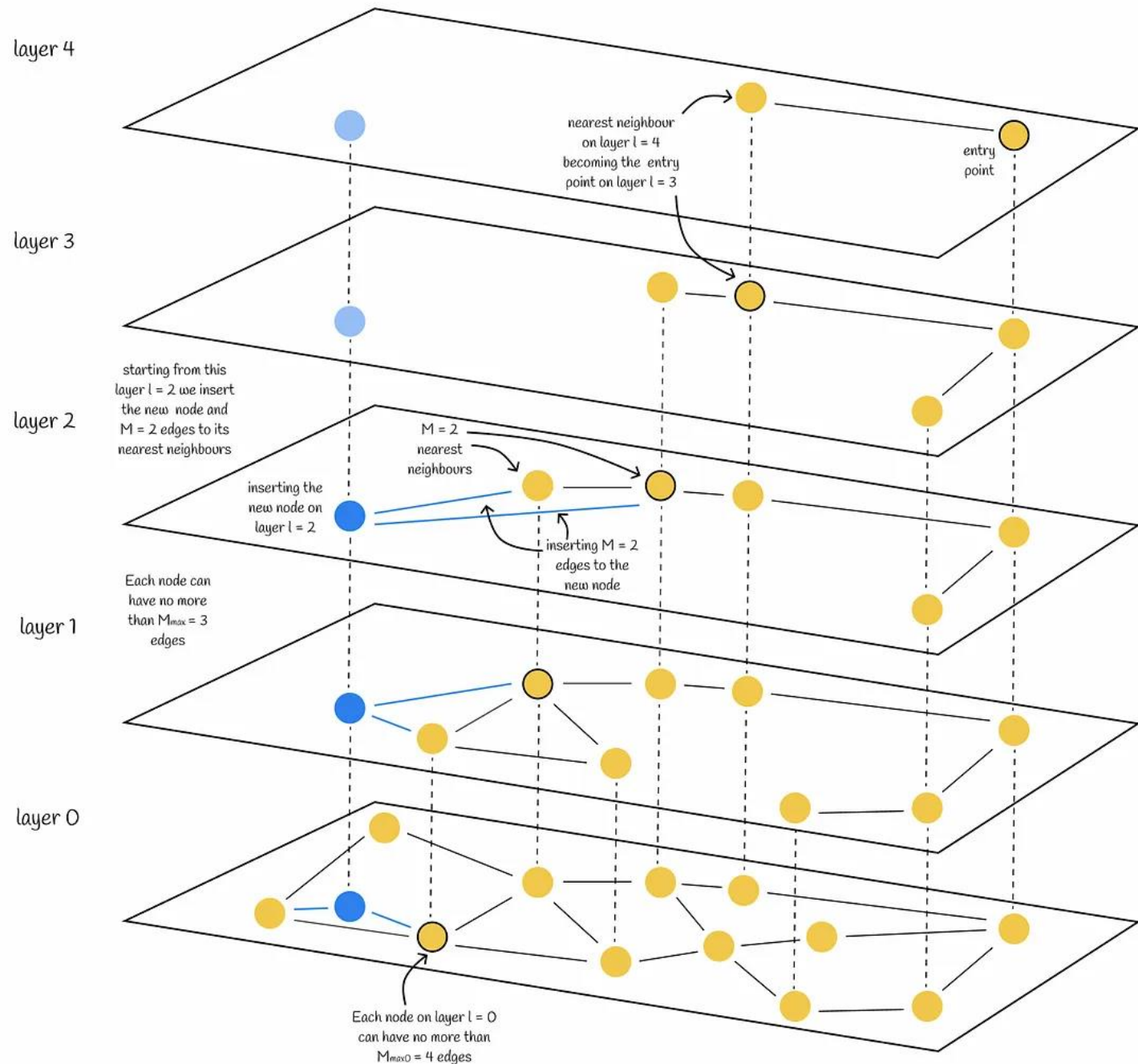
# Constructing HNSW

Starting from **layer l** the algorithm inserts the new node at the current layer.

Then it acts the same as before at step 1 but instead of finding only one nearest neighbour, it greedily searches for **efConstruction** (hyperparameter) nearest neighbours.



layer 4

layer 3

layer 2

layer 1

layer 0

nearest neighbour on layer l = 4 becoming the entry point on layer l = 3

entry point

starting from this layer l = 2 we insert the new node and M = 2 edges to its nearest neighbours

M = 2 nearest neighbours

inserting the new node on layer l = 2

inserting M = 2 edges to the new node

Each node can have no more than $M_{max} = 3$ edges

Each node on layer l = 0 can have no more than $M_{max0} = 4$ edges

# Constructing HNSW

Then **M** out of **efConstruction** neighbours are chosen and edges from the inserted node to them are built.

After that, the algorithm descends to the next layer and each of found **efConstruction** nodes acts as an entry point. The algorithm terminates after the new node and its edges are inserted on the lowest layer 0.