Indexing in Information Retrieval

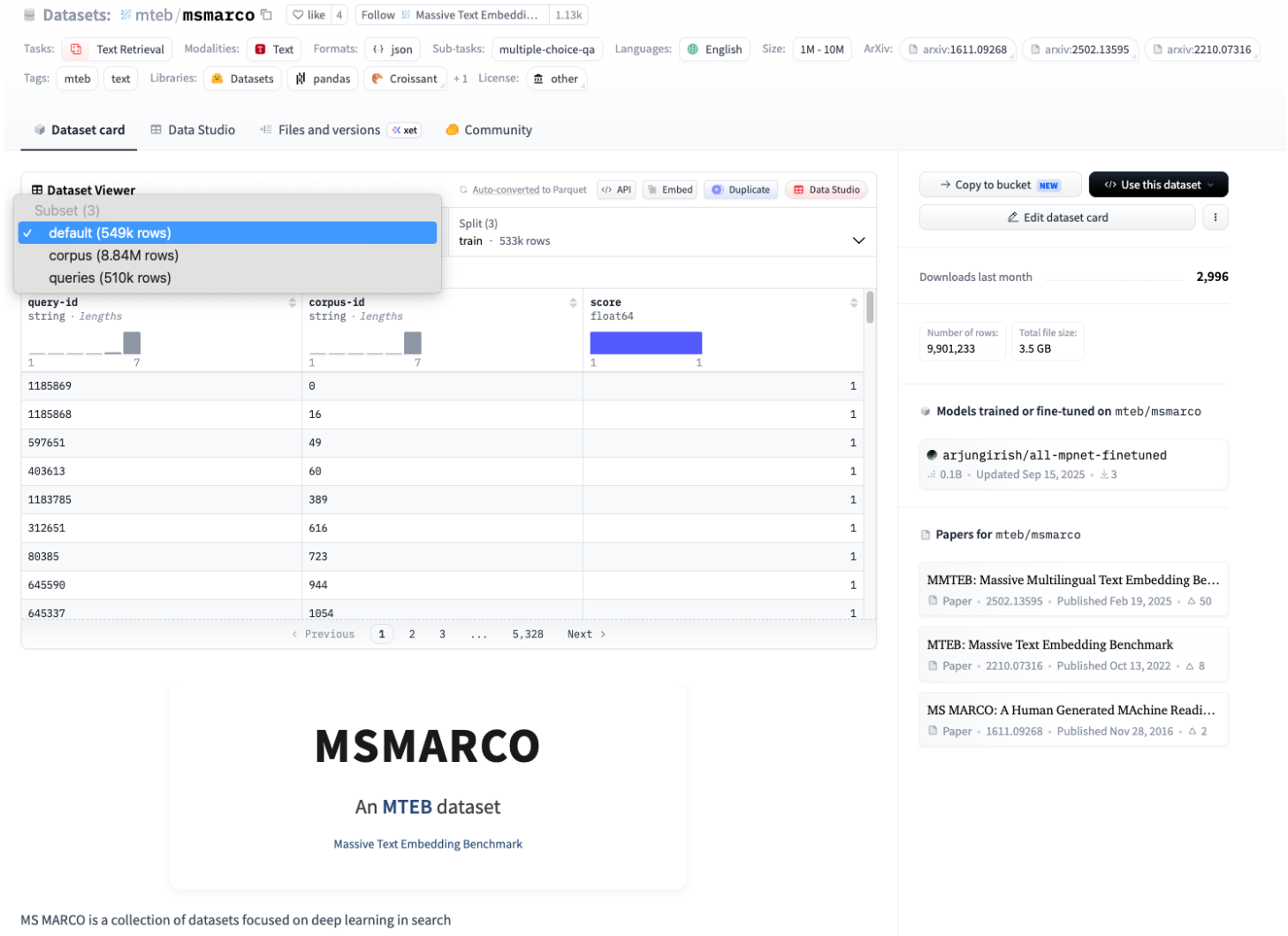
2026 하계세미나

장영준

Table Of Contents

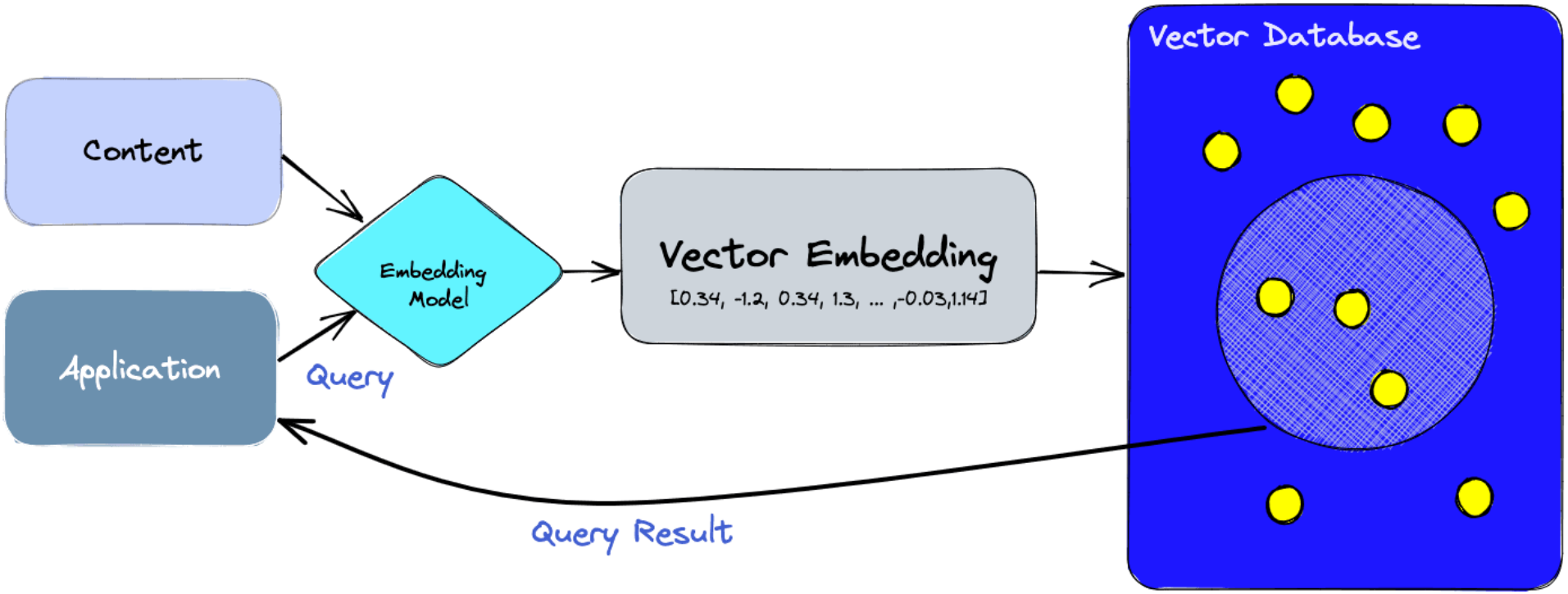
Preliminaries

Evaluation in IR



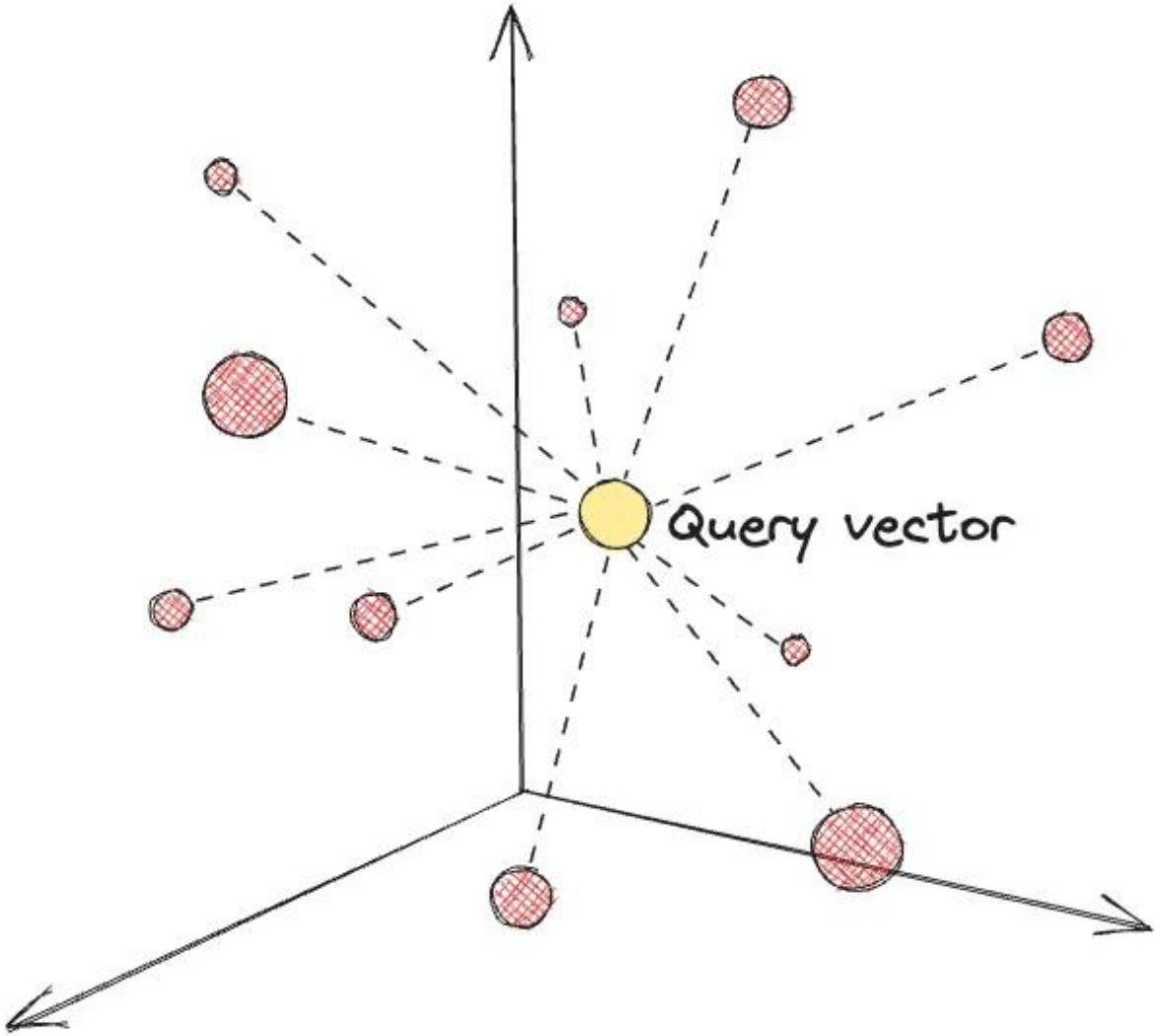
Preliminaries

Evaluation in IR



Preliminaries

Evaluation in IR

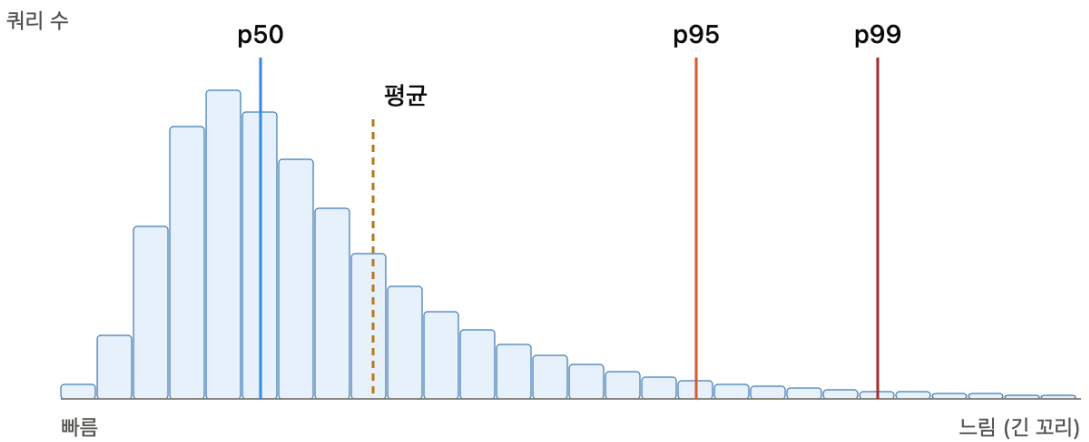


Preliminaries

Evaluation in IR

Metrics

- 성능: nDCG@k, Recall@k
- 용량: RAM (GB)
- Latency: p50, p95, p99



평균은 꼬리에 끌려가 p50보다 오른쪽에 위치한다

Preliminaries

본 세미나의 목적

- 자주 사용되는 Indexing 기법들의 개념을 익힌다.
- nlpai-lab/KURE-v1 모델로, miracl (query: 213, corpus: 1.5M) 데이터에 대해 각 indexing 기법을 실험해보며, 실제 어떤 tradeoffs가 있는지 살펴본다.

	공략	대가	대표 지표
PQ	메모리	정확도 손실	RAM
IVF	속도	정확도 손실	p95
HNSW	속도	정확도 손실 / 메모리 증가	P95 + RAM

Exhaustive Search (FlatIP)

Exhaustive Search (FlatIP)

FlatIP

- Approximate Nearest Neighbor (ANN)을 사용하지 않고, 검색 시 모든 벡터와 비교를 하는 방법
- 장점: 매우 정확 (상한) / 단점: 매우 느림
- Cosine Similarity = 벡터를 **Flat** (normalize) + **I**nner **P**roduct (**IP**) ➔ **FlatIP**

nDCG@10	Recall@10	p50	p95	p99	RAM
0.682	0.779	526.7ms	532.8ms	538.4ms	5.7GB

Product Quantization

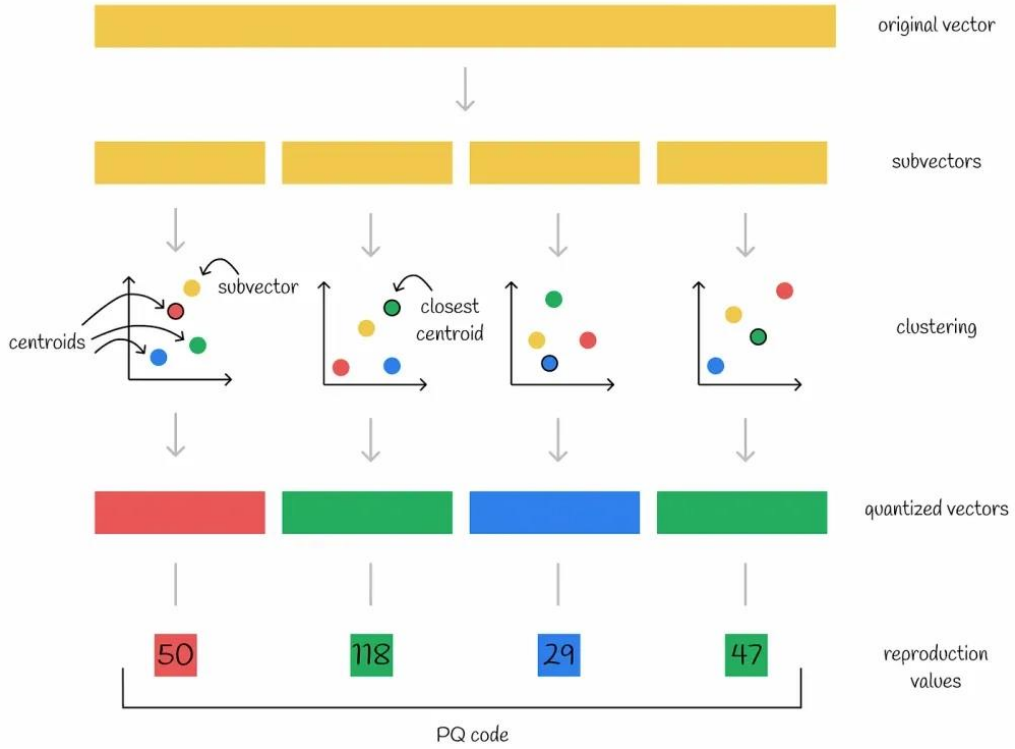
Product Quantization (PQ)

PQ

벡터의 개수가 많아지면, 메모리 부하가 커짐 ➔ 데이터 압축도 하고, 거리 계산도 손실 없이 할 수 있는 알고리즘이 필요

[색인 과정 (Indexing Documents)]

1. 각 벡터를 동등한 n개의 subvector로 나눔
   (partition: 1, 2, ..., n)
2. 각 벡터의 i번째 subvector끼리 K-Means clustering 수행
   ➔ 각 subvector마다 k개의 cluster centroid를 가짐
3. 한 벡터의 n개의 subvector는 각각 가장 가까운
   cluster의 번호를 값으로 가짐.



[RAM 변화] (fp32, dim=1024, n_subvectors=4, K=256)

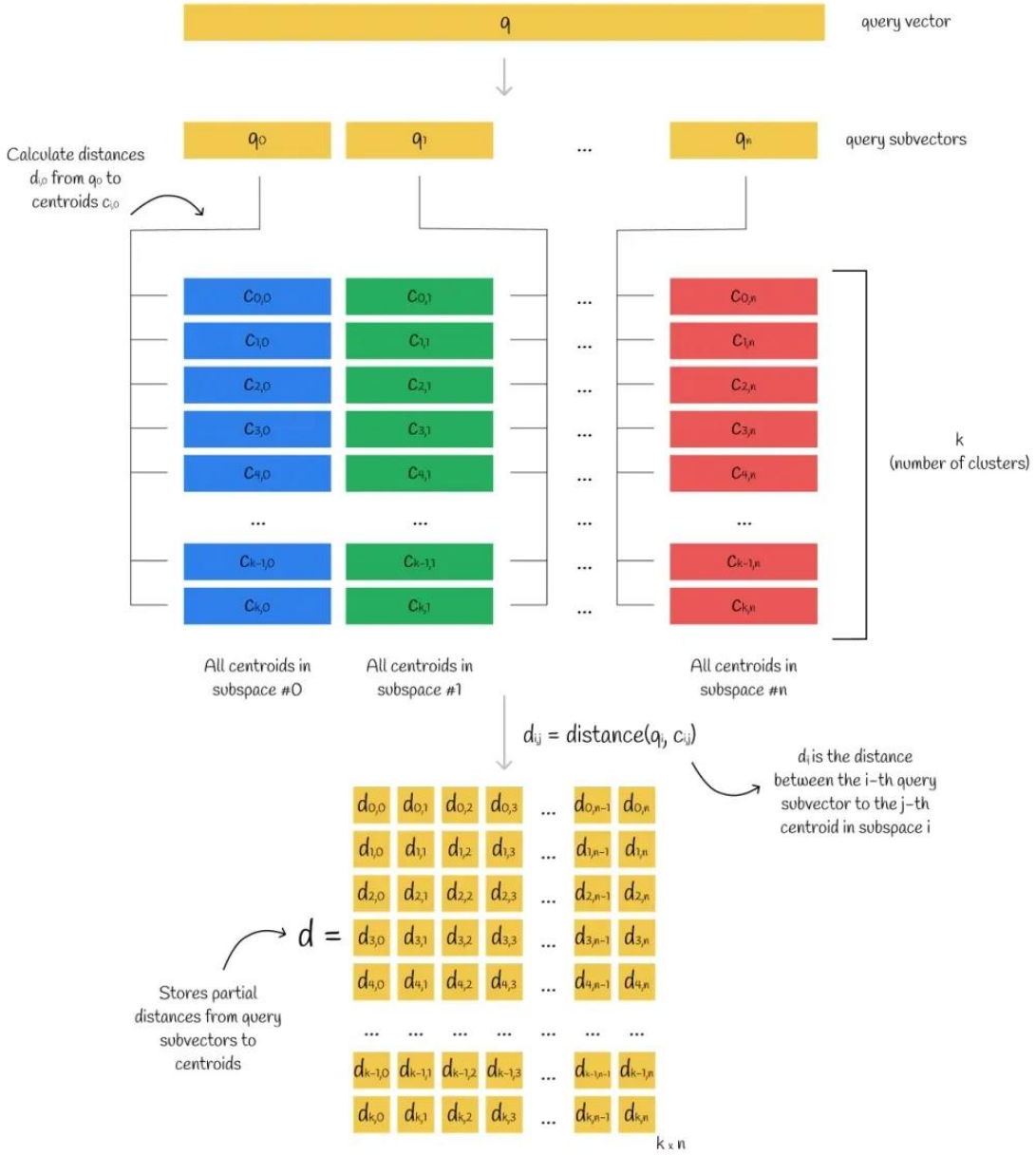
- 기존: **1024** x **4** (fp32 ➔ 32bits=4byte) = **4096 bytes**
- PQ 적용 후: **4** (n_subvectors) x **1** (0~255=1byte) = **4bytes**

Product Quantization (PQ)

PQ

[추론 과정 (Inference)]

1. query 또한 동일한 n개의 subvectors로 분할
2. query의 각 subvector와 document subevector로 만든
   k개의 cluster centroid들과 유사도 계산해서
   유사도 matrix 작성 (LookUpTable, LUT)



Product Quantization (PQ)

PQ

[추론 과정 (Inference)]

3. 각 document 벡터마다 cluster index가 매핑되어 있으니,
   LUT를 조회하면서 query에 대한 각 doc의 유사도를 구함
   (Exhaustive Search 예시)

PQ Experiment Setup

[Hyperparams]

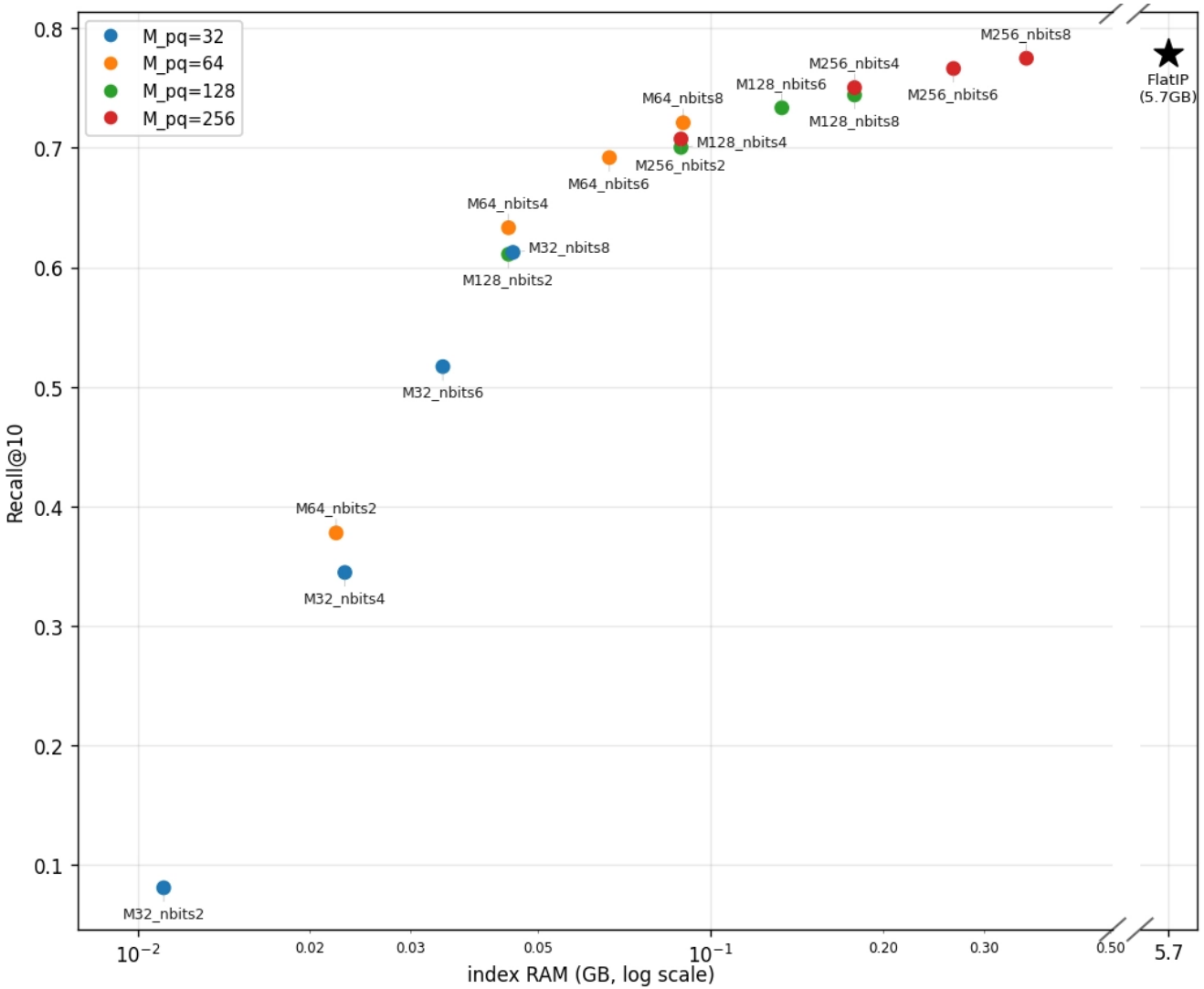
- **M**: subvector 개수 (몇 개의 subvectors로 쪼개지는지?) – 32, 64, 128, 256
- **nbits**: code 비트 수 (# of centroid = 2^{nbits}) - 2, 4, 6, 8
  (앞선 예시에서는 nbits=8 -> 256개의 centroids)

Product Quantization (PQ)

PQ Experiment Results

- **M**: subvector 개수
- **nbits**: code 비트 수

  (# of centroid = 2^{nbits})

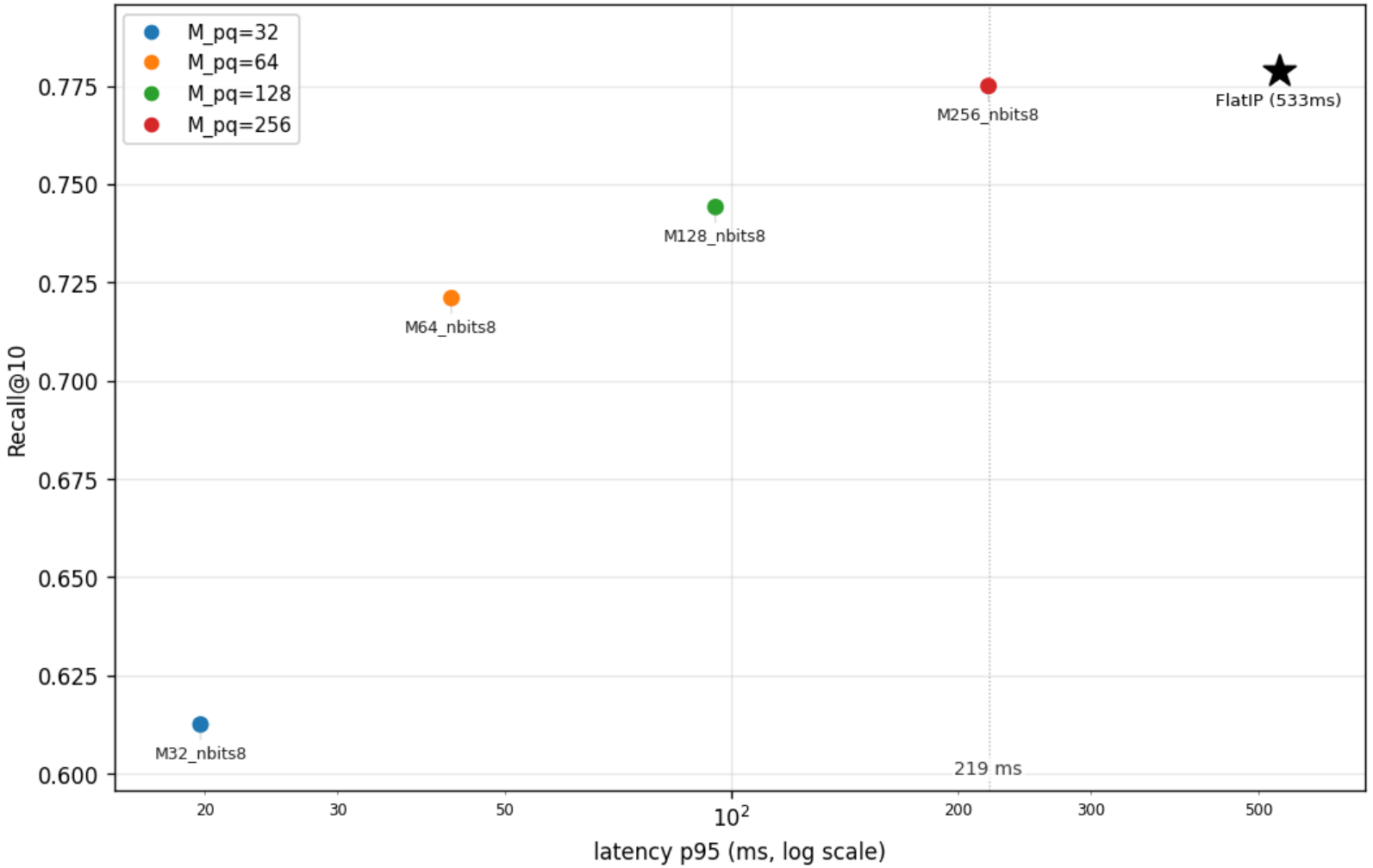


Product Quantization (PQ)

PQ Experiment Results

- **M**: subvector 개수
- **nbits**: code 비트 수

  (# of centroid = 2^{nbits})

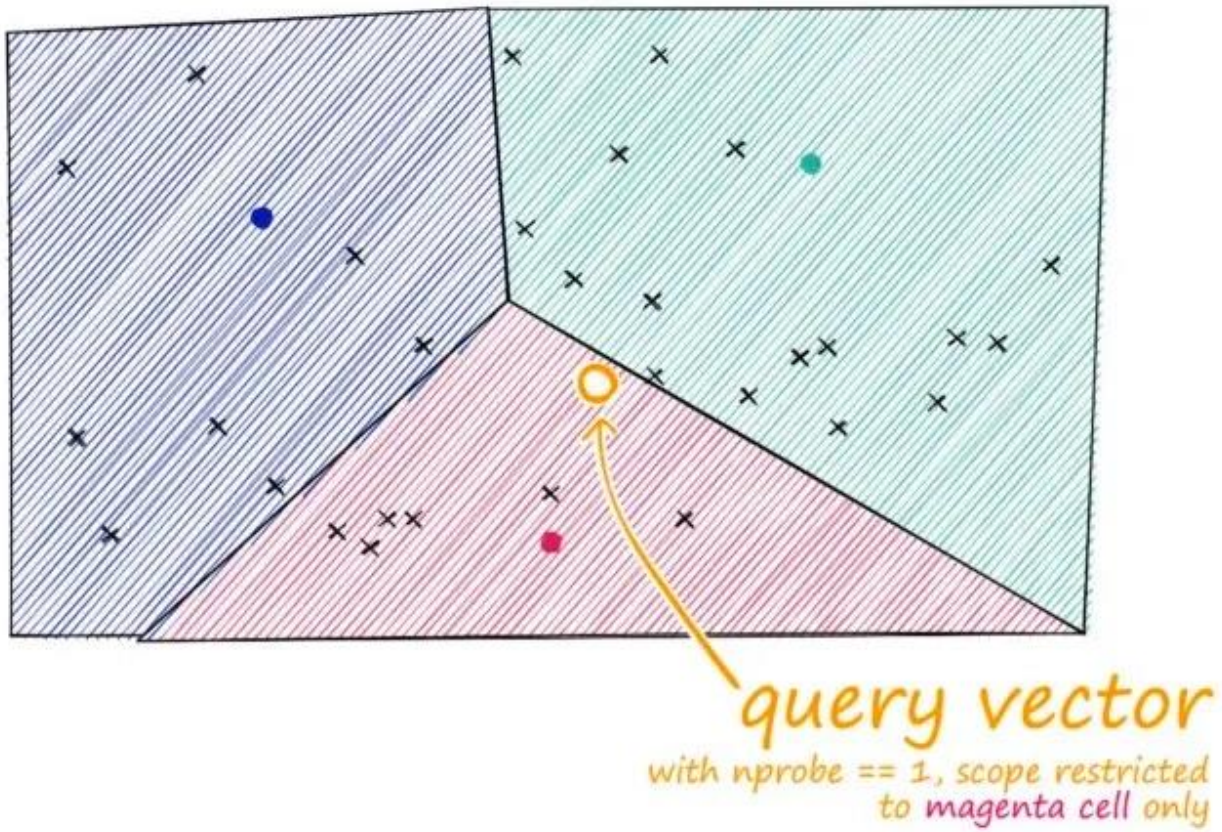


Inverted File Index

Inverted File Index (IVF)

IVF

- Document vector들을 clustering 해두고, query time에 각 cluster centroid랑 kNN search 한 뒤, 가까운 cluster에 있는 vector들과만 kNN search 하는 방식

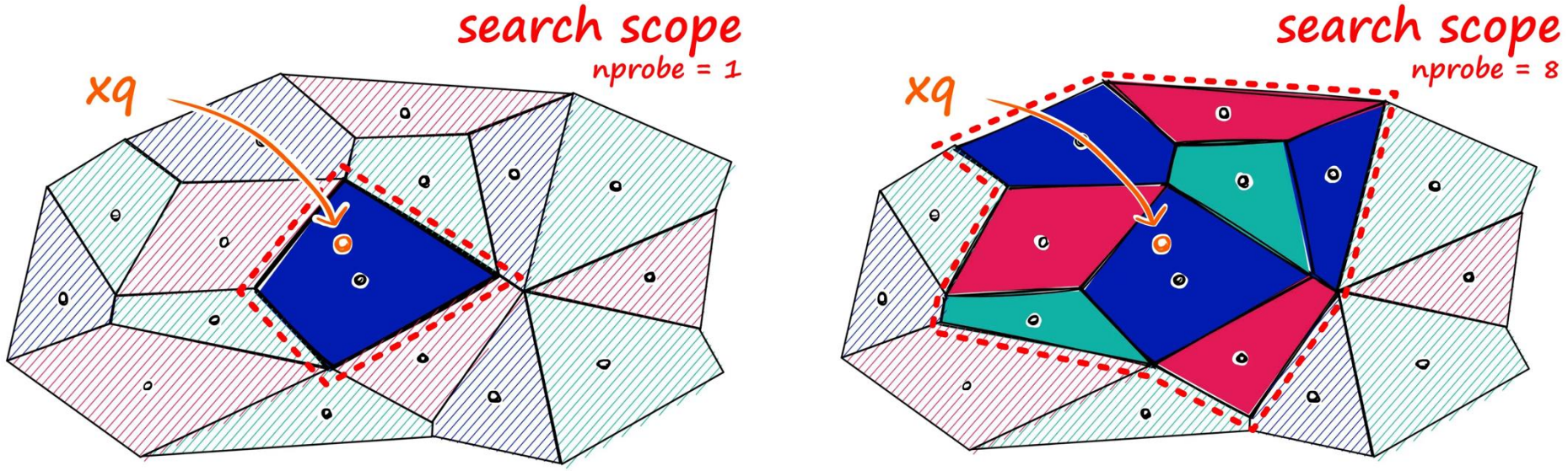


Inverted File Index (IVF)

IVF

[Hyperparams]

- **nlist**: corpus vector 들의 partition 개수 – 1024, 4096, 16384
- **nprobe**: query와 가장 가까운 몇 개의 cluster를 볼 것인지 – 1, 4, 16, 64, 256

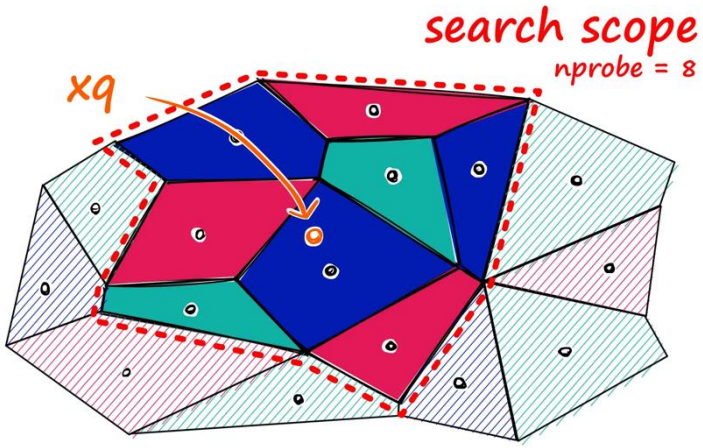


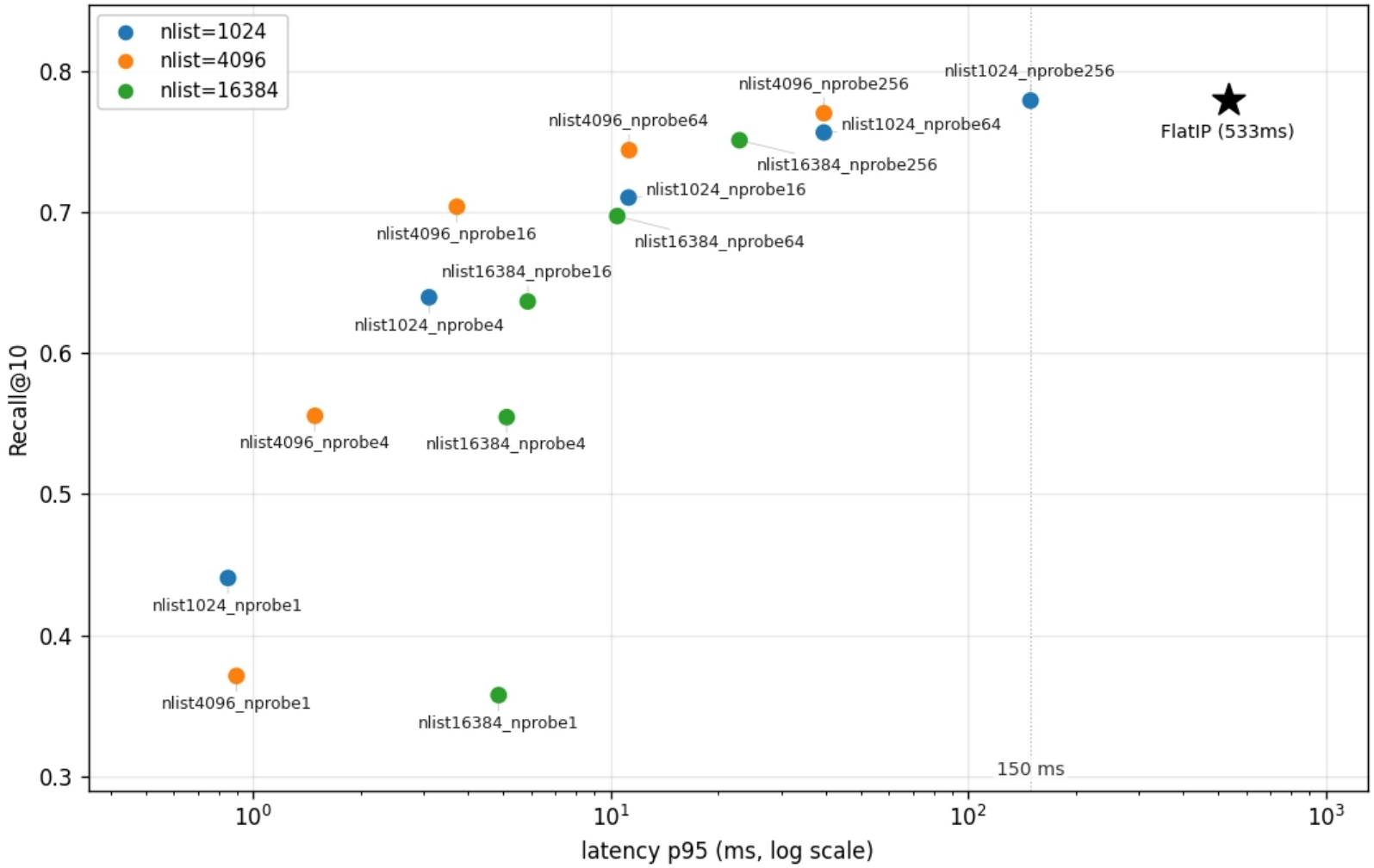
Inverted File Index (IVF)

IVF

[Hyperparams]

- **nlist**: partition 개수
- **nprobe**: 검색 시 몇 개의 cluster





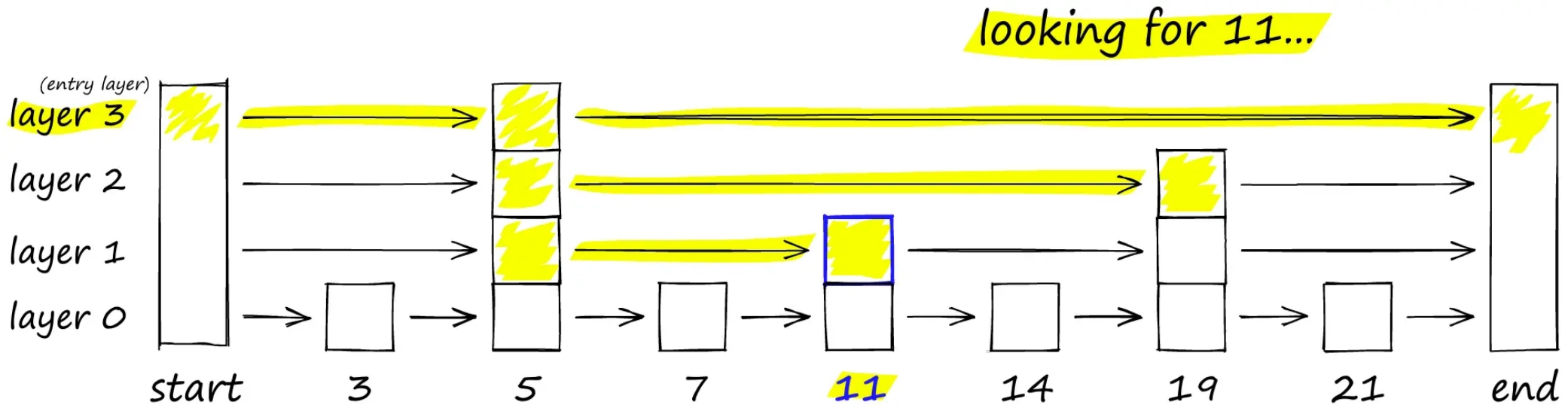
Hierarchical Navigable Small Worlds (HNSW)

Preliminaries

HNSW

1. Skip List

- 상위 -> 하위 레이어로 이동할수록 각 노드간의 연결이 늘어나는 list
- O(logN) 시간복잡도

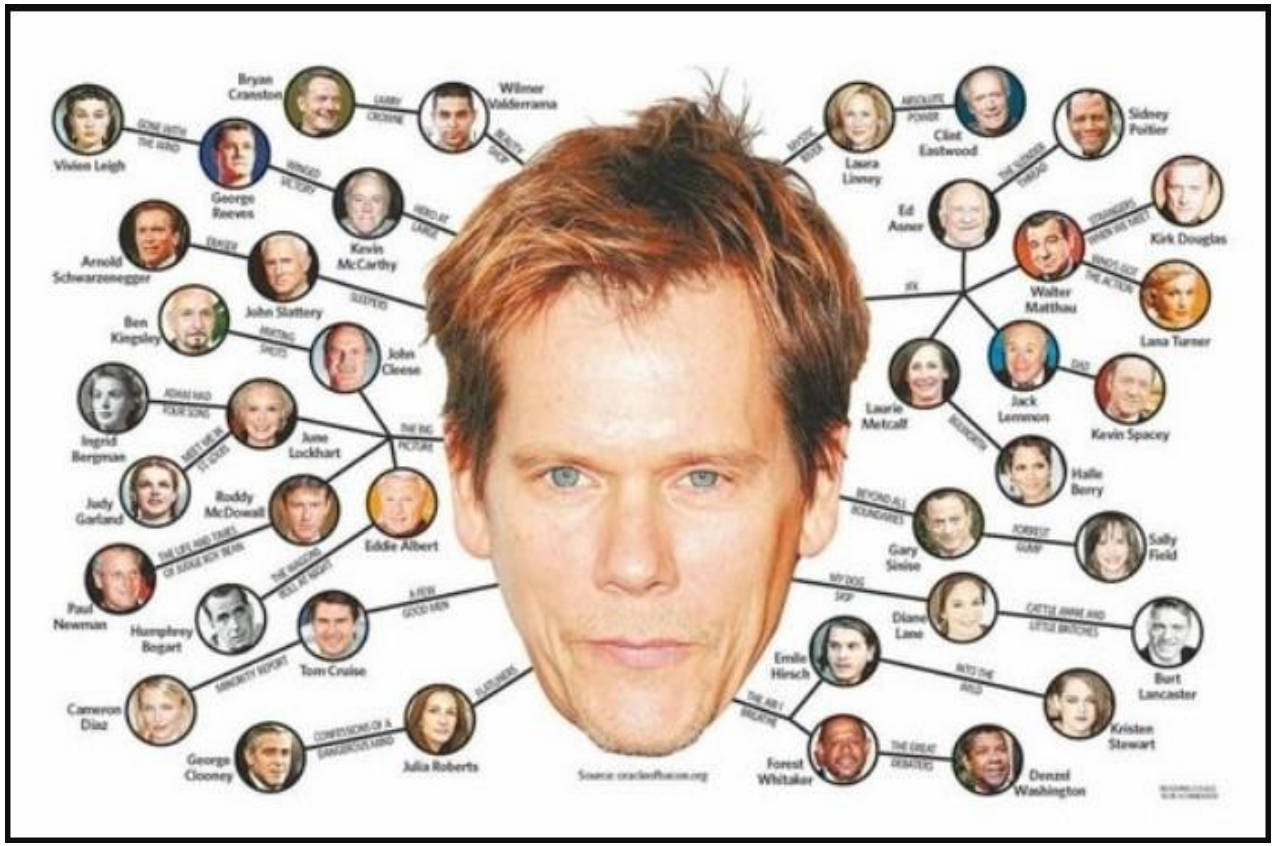


Preliminaries

HNSW

2. Navigable Small Worlds (NSW)

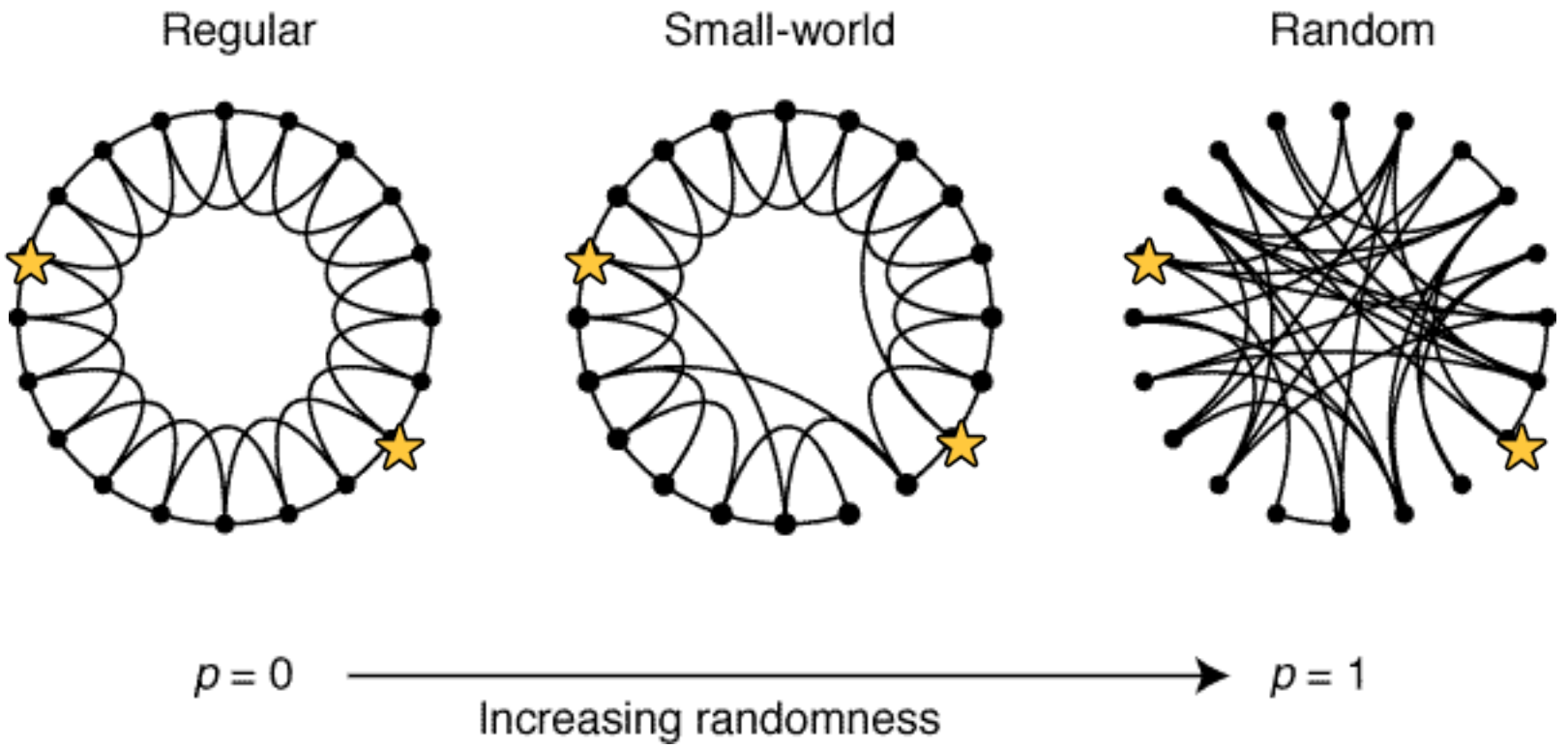
- 케빈 베이컨의 6단계 법칙



Preliminaries

HNSW

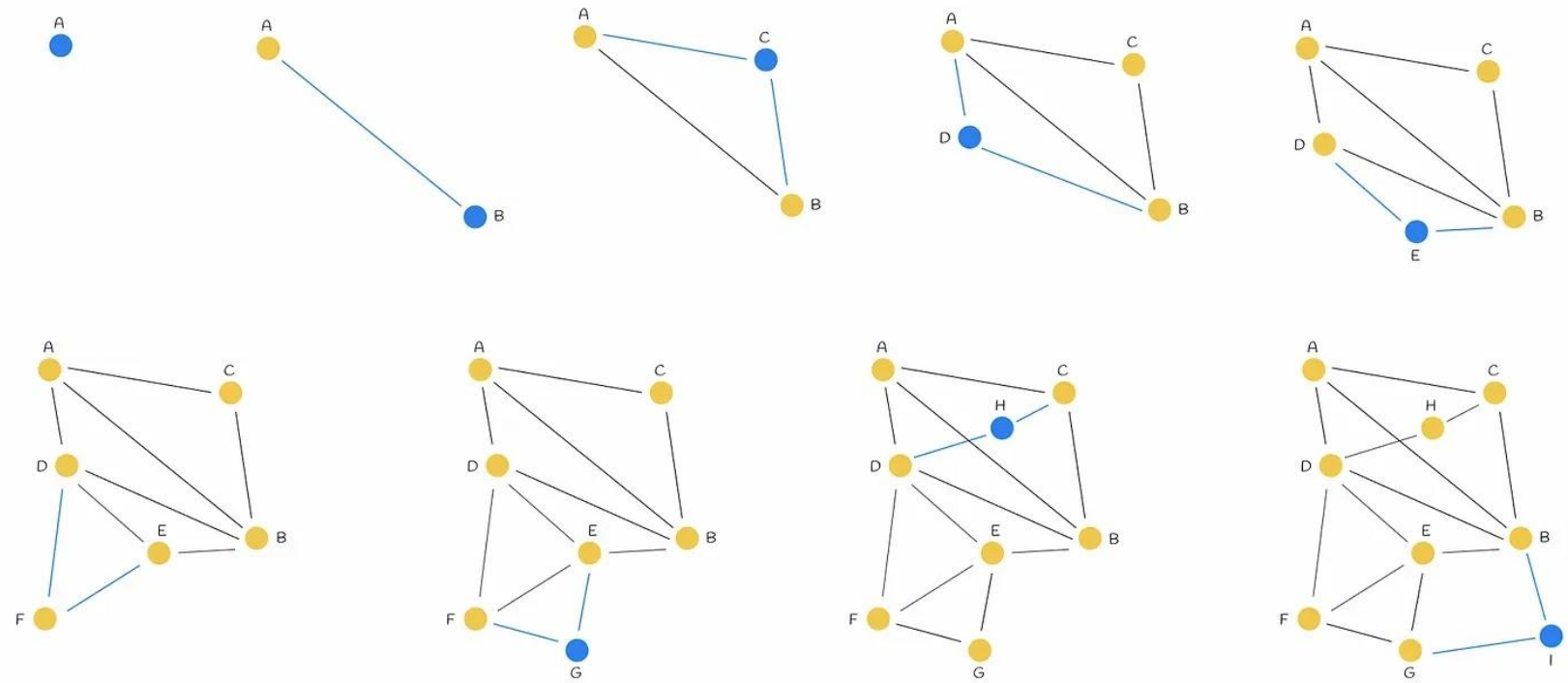
2. Navigable Small Worlds (NSW)



Preliminaries

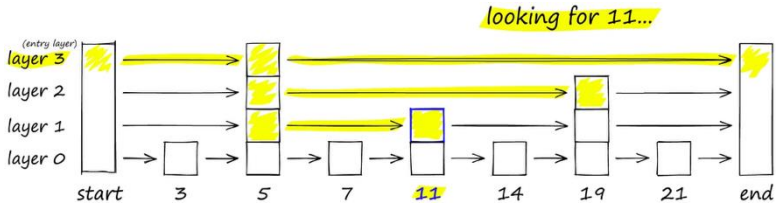
HNSW

2. Navigable Small Worlds (NSW)

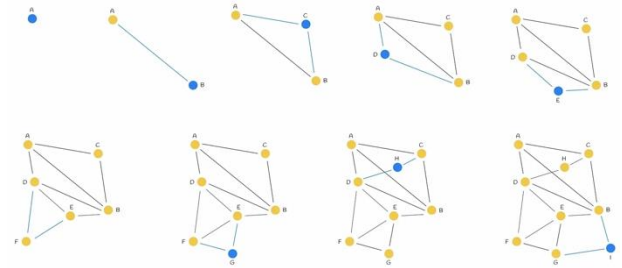


Preliminaries

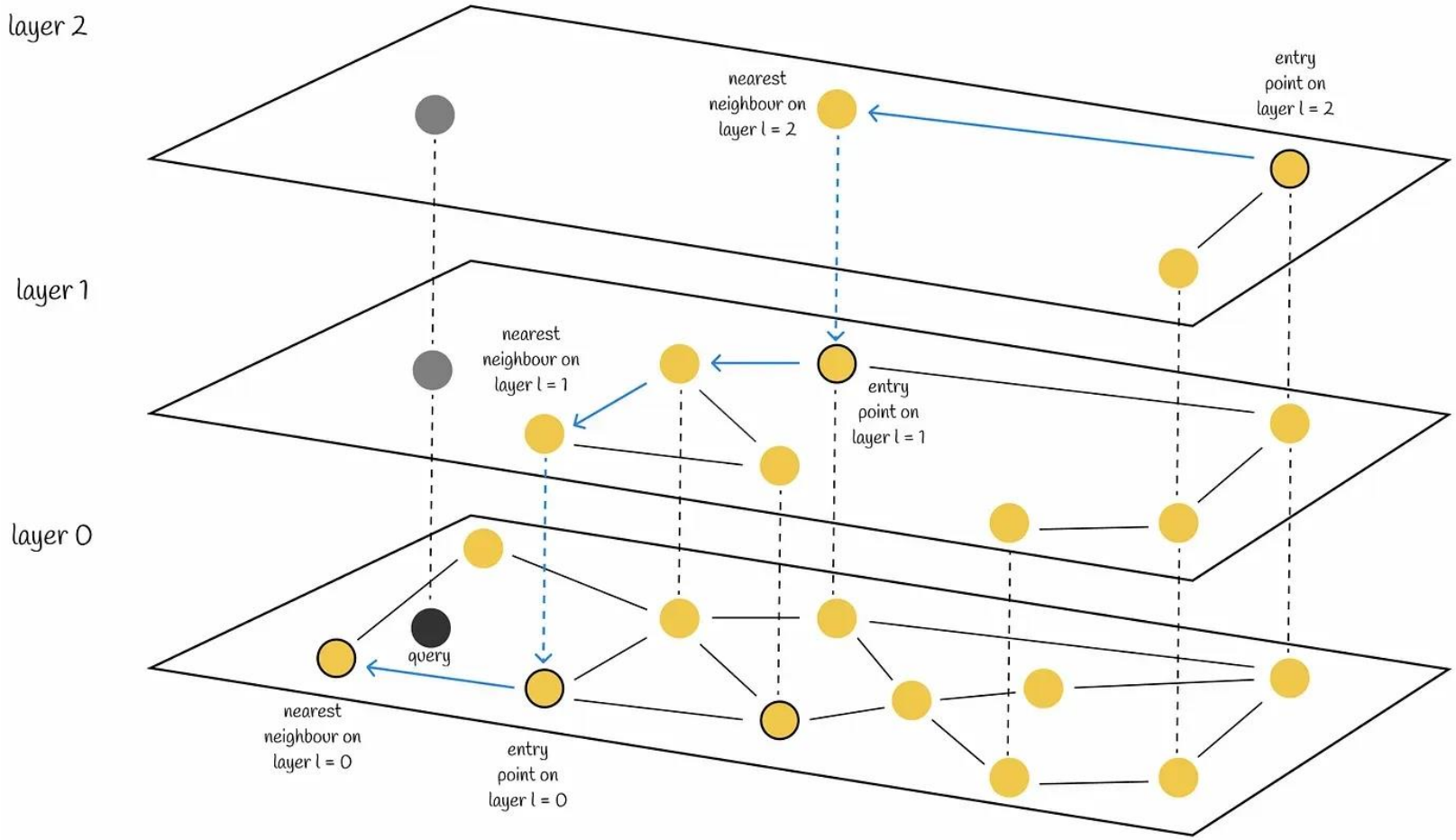
HNSW



+



HNSW = Skip-List + NSW

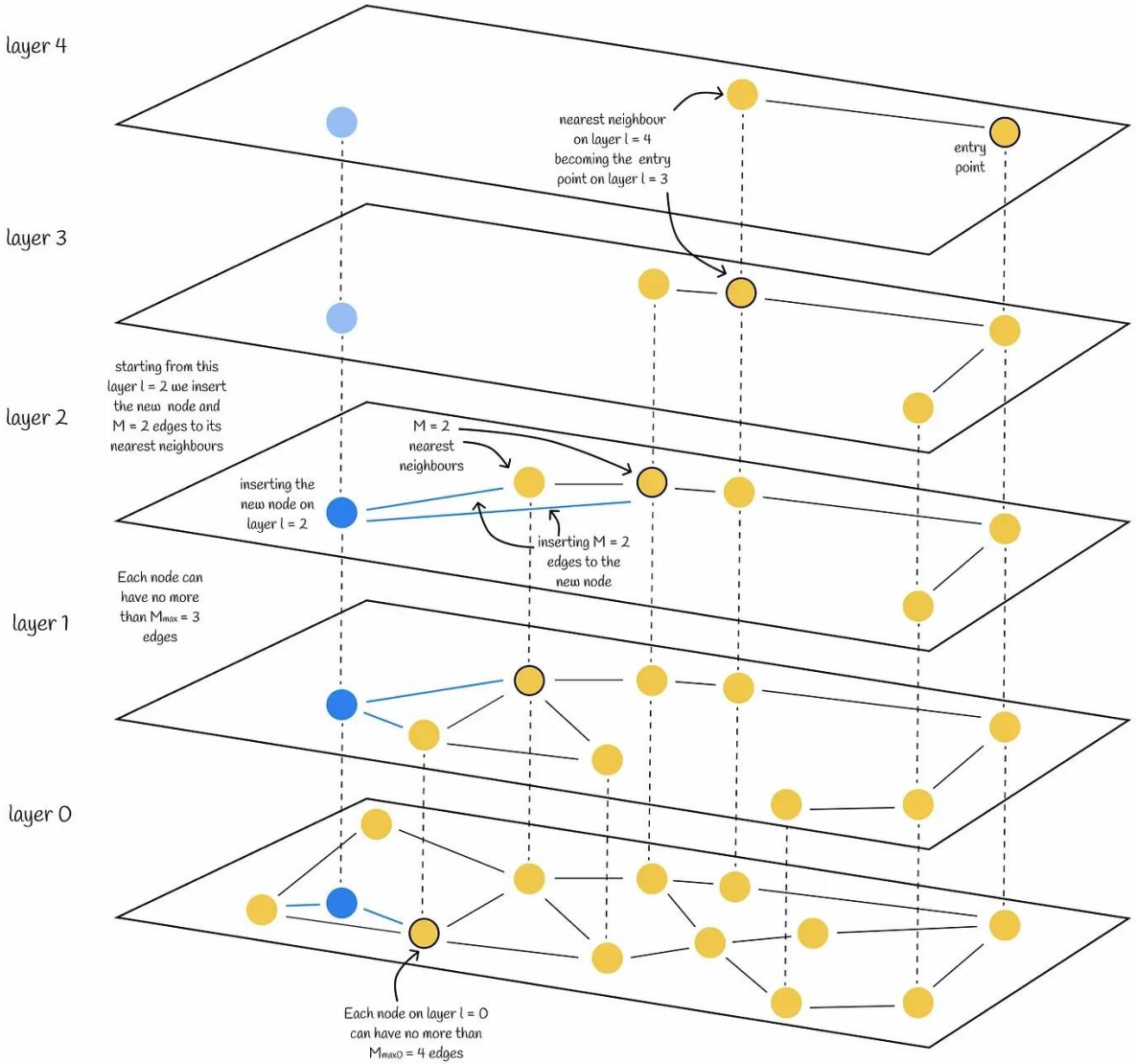


Preliminaries

HNSW

HNSW 그래프 Build (Explanation Ver.)

- $\boldsymbol{m_L}$: 새 노드 입력시 입력되는 가장 높은 레이어를 결정하는 파라미터
- $\boldsymbol{M}$: 노드가 새로 레이어에 입력될 때 연결할 edge 개수
- $\boldsymbol{M_{\{max\}}}$: 노드가 가질 수 있는 최대 edge 개수. 만약 새로운 노드가 연결되어 $M_{\{max\}}$ 보다 edge 개수가 많아지면 다른 edge를 버림
- $\boldsymbol{efConstruction}$: 입력시 이웃 노드들을 선발할 때 사용하는 후보 개수 (heap의 크기)
- 어떤 노드가 어떤 layer에 할당되는지: $l = \lfloor -m_L * \ln(u) \rfloor$
  - u: 0~1 사이의 uniform 확률
  - $m_L = \frac{1}{\ln(M)}$ (M: edge 개수)
- 파란색 노드가 L=2, M=2 파라미터로 삽입되었을 때의 빌드

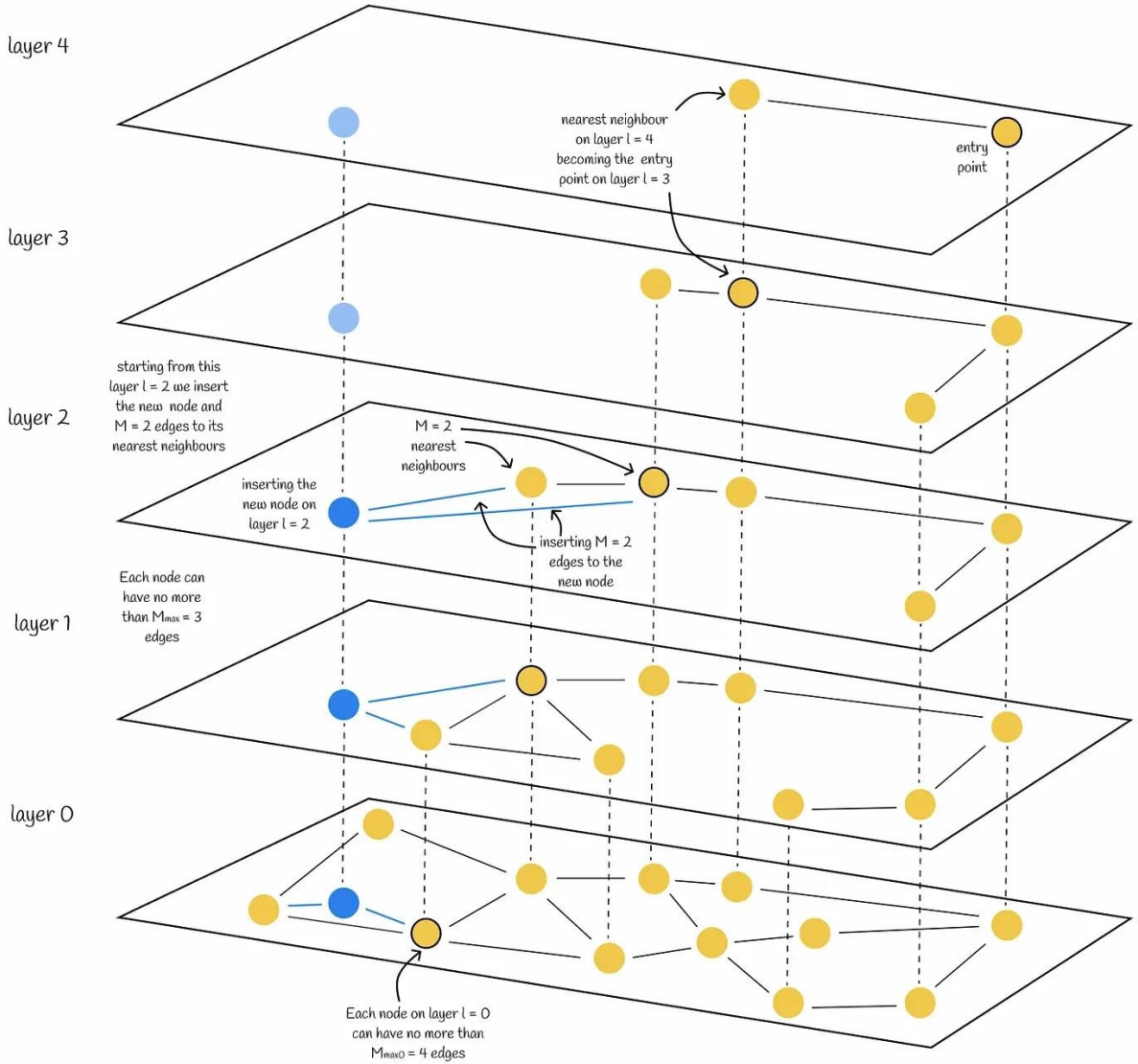


Preliminaries

HNSW

HNSW 그래프 Build (Explanation Ver.)

- **[L=4]** 삽입될 예정인 파란색 노드가 entry point에서 시작.
  레이어 2에 삽입될 예정이므로 레이어 4에는 edge를 만들지 않음.
  다만 entry point부터 탐색을 시작해서, (삽입될 예정인) 파란색 노드와 최대한 가까운 노드를 Greedy하게 탐색.
  해당 레이어에서 최대한 가까이 이동했으면 아래 레이어로 이동
- **[L=3]** 위 과정 반복
- **[L=2]** 파란색 노드가 실제로 삽입됨. 먼저 레이어 2에서 파란색 노드와 가장 가까운 노드를 찾고, 파란색 노드 기준 가장 가까운 노드 2개(M=2)와 edge를 연결
- **[L=1]** 위 과정 반복
- **[L=0]** 위 과정 반복하는데, M_{max0} (주로 $max0 = 2 * M$)을 넘지 않는 선에서 edge 연결

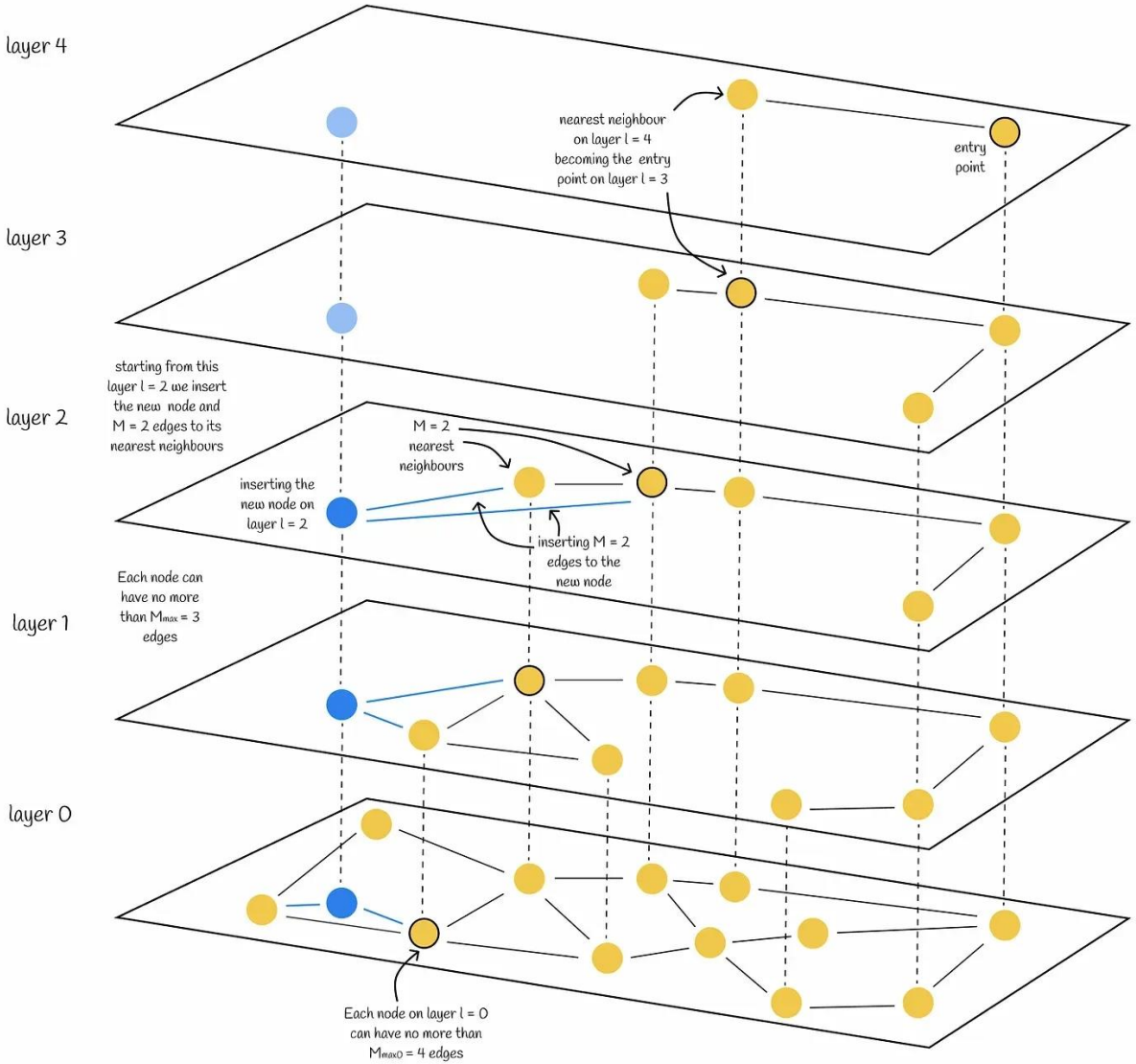


Preliminaries

HNSW

HNSW 그래프 Build (Practical Ver.)

- m_L: 새 노드 입력시 입력되는 가장 높은 레이어를 결정하는 파라미터
- M: 노드가 새로 레이어에 입력될 때 연결할 edge 개수
- $M_{\{max\}}$: 노드가 가질 수 있는 최대 edge 개수. 만약 새로운 노드가 연결되어 $M_{\{max\}}$ 보다 edge 개수가 많아지면 다른 edge를 버림
- $efConstruction$: 입력시 이웃 노드들을 선발할 때 사용하는 후보 개수 (**heap**의 크기)
  1. **후보 queue (min heap):** "앞으로 탐색해볼 노드들"
  2. **결과 queue (max heap):** "지금까지 발견한 가장 좋은 노드들"
     ← $efConstruction$



Preliminaries

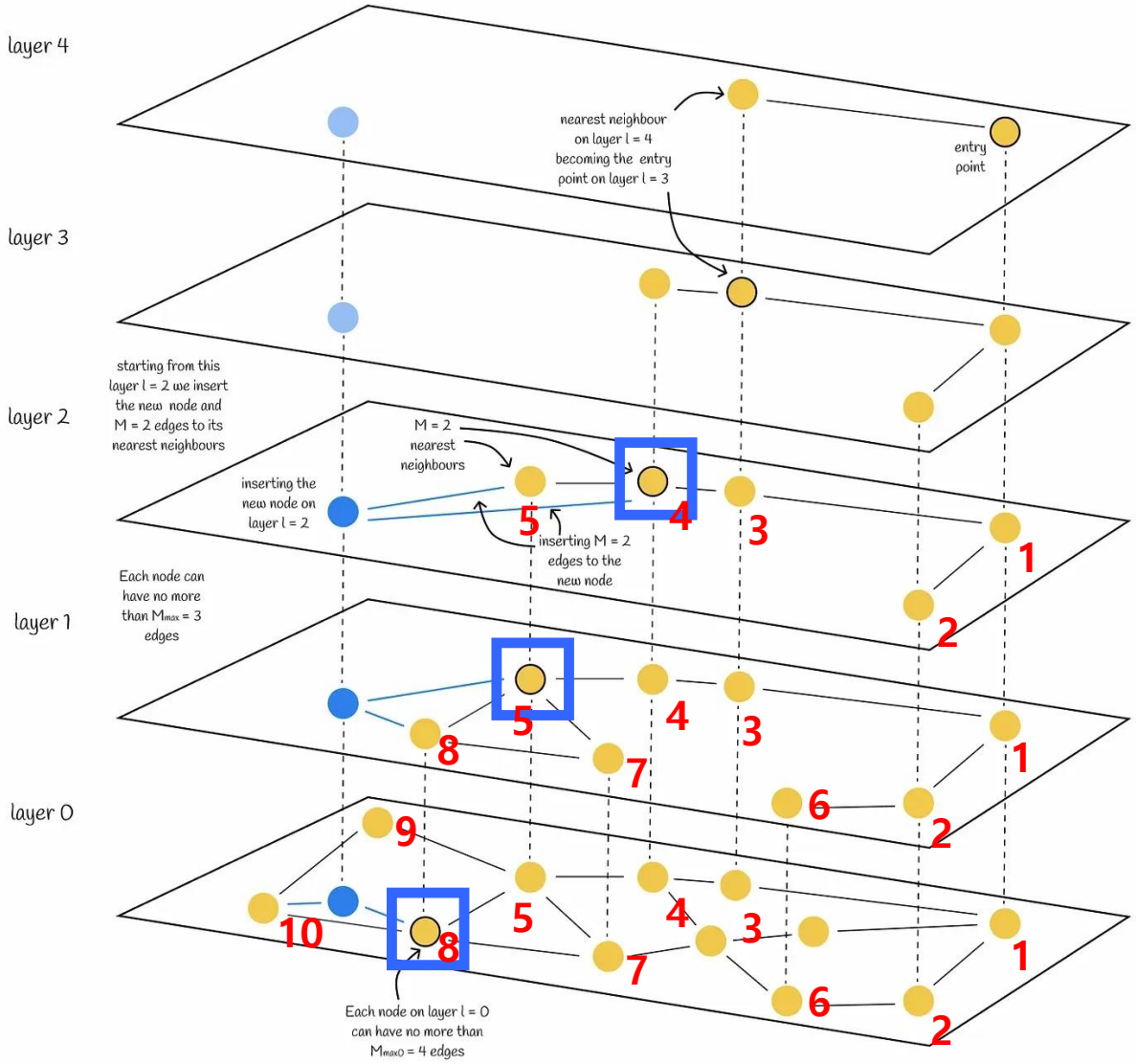
HNSW

HNSW 그래프 Build (Practical Ver.)

- efConstruction = max heap의 크기 = 4
- [L=2] 에 파란색 노드가 entry point에 삽입

1. 초기화 상태
   - 후보 큐 (Min-Heap, 가장 가까운 순): [(0.5, 노드 4)]
   - 결과 큐 (Max-Heap, 가장 먼 순): [(0.5, 노드 4)]
2. 탐색

   [STEP1] 첫 번째 후보 꺼내서 비교
   - 후보 큐에서 Pop: 가장 가까운 C= 노드 4 (거리 0.5)
   - 결과 큐에서 Max 확인: 가장 먼 F= 노드 4 (거리 0.5)
   - 종료 조건 검사: C의 거리(0.5) > F의 거리(0.5) 인가?
     ➔ False. 탐색 진행
   - 이웃 탐색: 노드 4의 이웃 노드 5 (거리 0.2), 노드 3 (거리 0.7)을 두 큐에 모두 적재
     - 현재 후보 큐: [(0.2, 노드 5), (0.7, 노드 3)]
     - 현재 결과 큐: [(0.7, 노드 3), (0.5, 노드 4), (0.2, 노드 5)]



Preliminaries

HNSW

현재 후보 큐: [(0.2, 노드 5), (0.7, 노드 3)]
현재 결과 큐: [(0.7, 노드 3), (0.5, 노드 4), (0.2, 노드 5)]

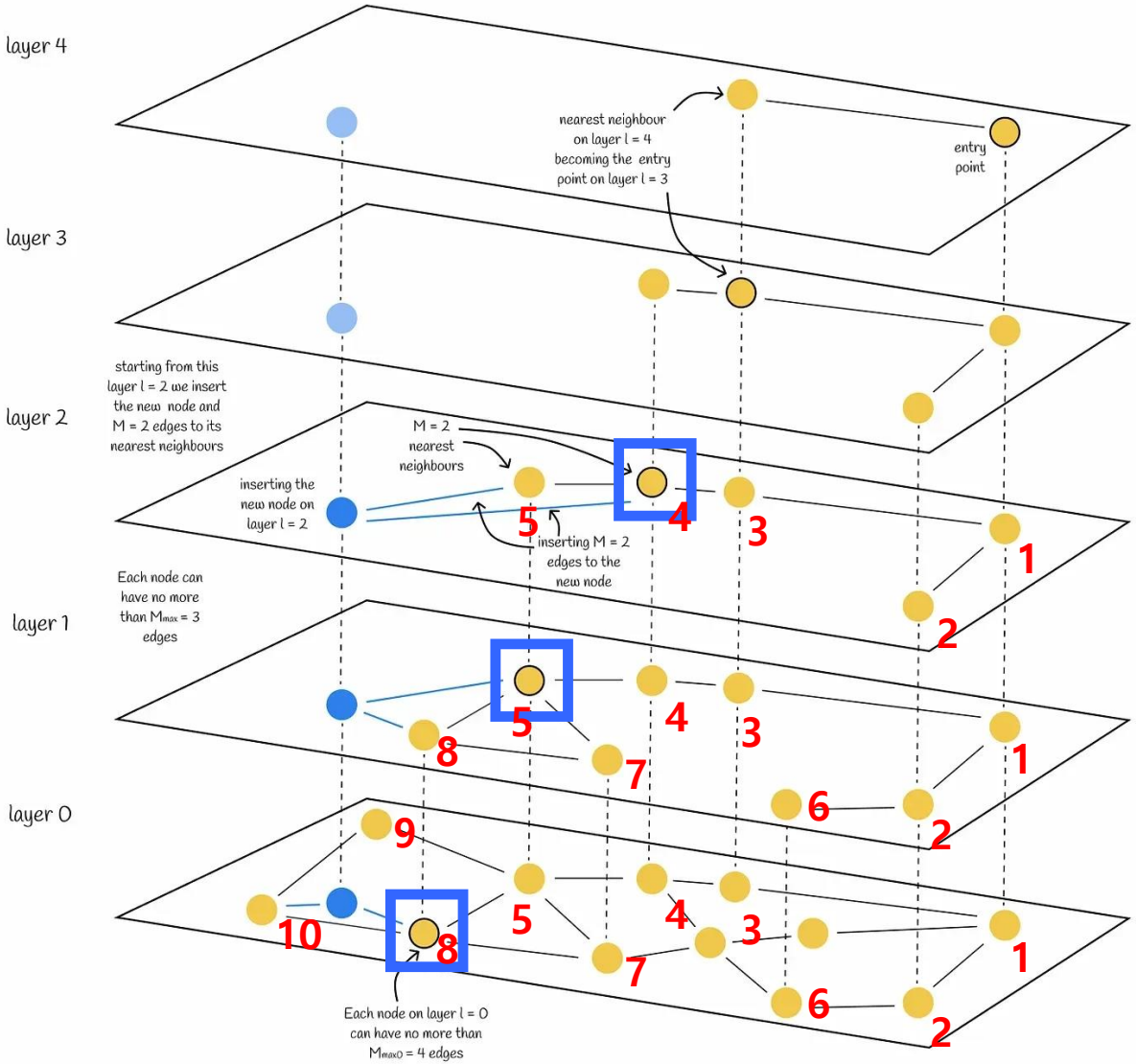
HNSW 그래프 Build (Practical Ver.)

[Step 2] 두 번째 후보 꺼내서 비교

- 후보 큐에서 Pop: 가장 가까운 C= 노드 5 (거리 0.2)
- 결과 큐에서 Max 확인: 가장 먼 F= 노드 3 (거리 0.7)
- 종료 조건 검사: C의 거리(0.2) > F의 거리(0.7) 인가?
  ➔ False. 탐색 진행
- 이웃 탐색: 노드 5의 이웃은 노드 4인데, 결과 큐에 이미 있음. 노드 5만 후보 큐에서 뺌
  ➔ 현재 후보 큐: [(0.7), 노드3]

[Step 3] 세 번째 후보 꺼내서 비교

- 후보 큐에서 Pop: 가장 가까운 C= 노드 3 (거리 0.7)
- 결과 큐에서 Max 확인: 가장 먼 F= 노드 3 (거리 0.7)
- 종료 조건 검사: C의 거리(0.2) > F의 거리(0.7) 인가?
  ➔ False. 탐색 진행
- 이웃 탐색: 노드 3의 이웃은 노드 4, 노드 1인데, 노드 4는결과 큐에 이미 있음. 노드 1과 비교
  - 현재 후보 큐: [(0.8, 노드 1)]
  - 현재 결과 큐: [(0.8, 노드 1), (0.7, 노드 3), (0.5, 노드 4), (0.2, 노드 5)]



Preliminaries

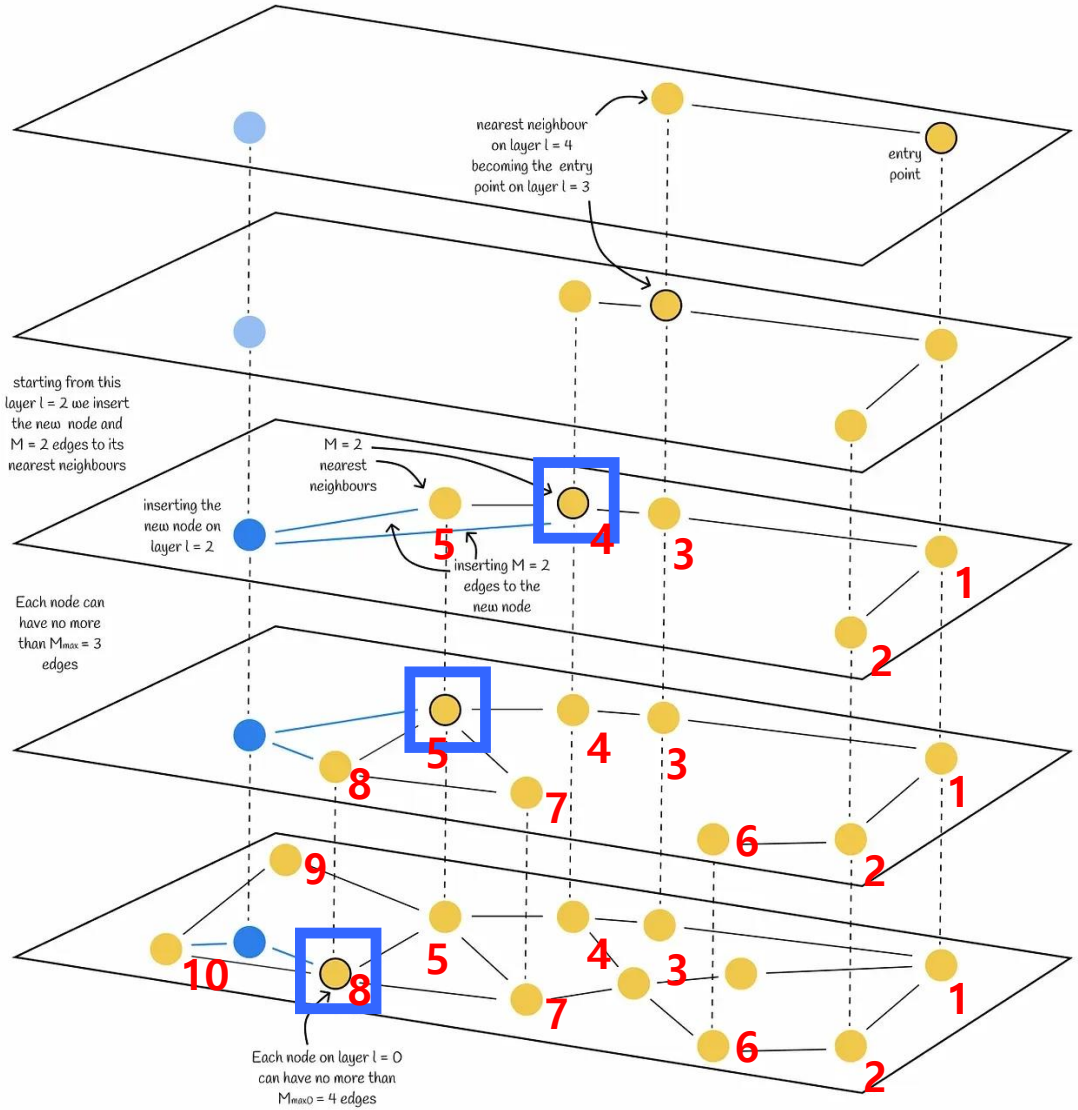
HNSW

현재 후보 큐: [(0.8, 노드 1)]
현재 결과 큐: [(0.8, 노드 1), (0.7, 노드 3), (0.5, 노드 4), (0.2, 노드 5)]

HNSW 그래프 Build (Practical Ver.)

[Step 4] 네 번째 후보 꺼내서 비교

- 후보 큐에서 Pop: 가장 가까운 C= 노드 1 (거리 0.8)
- 결과 큐에서 Max 확인: 가장 먼 F= 노드 1 (거리 0.8)
- 종료 조건 검사: C의 거리(0.8) > F의 거리(0.8) 인가?
  ➔ False. 탐색 진행
- 이웃 탐색: 노드 1의 이웃은 노드 2 (거리 0.9)
  - 결과 큐 꽉 참 (efC=4)
  - 노드 2 (거리 0.9), 노드 1 (거리 0.8) 비교 ➔ 노드 1이 더 가까워서, 결과 큐에 노드 2 반영 X
  - 현재 후보 큐: [()] → 고갈
  - 현재 결과 큐: [(0.8, 노드 1), (0.7, 노드 3), (0.5, 노드 4), (0.2, 노드 5)]
  - 후보 큐 고갈되어 stop condition 발생



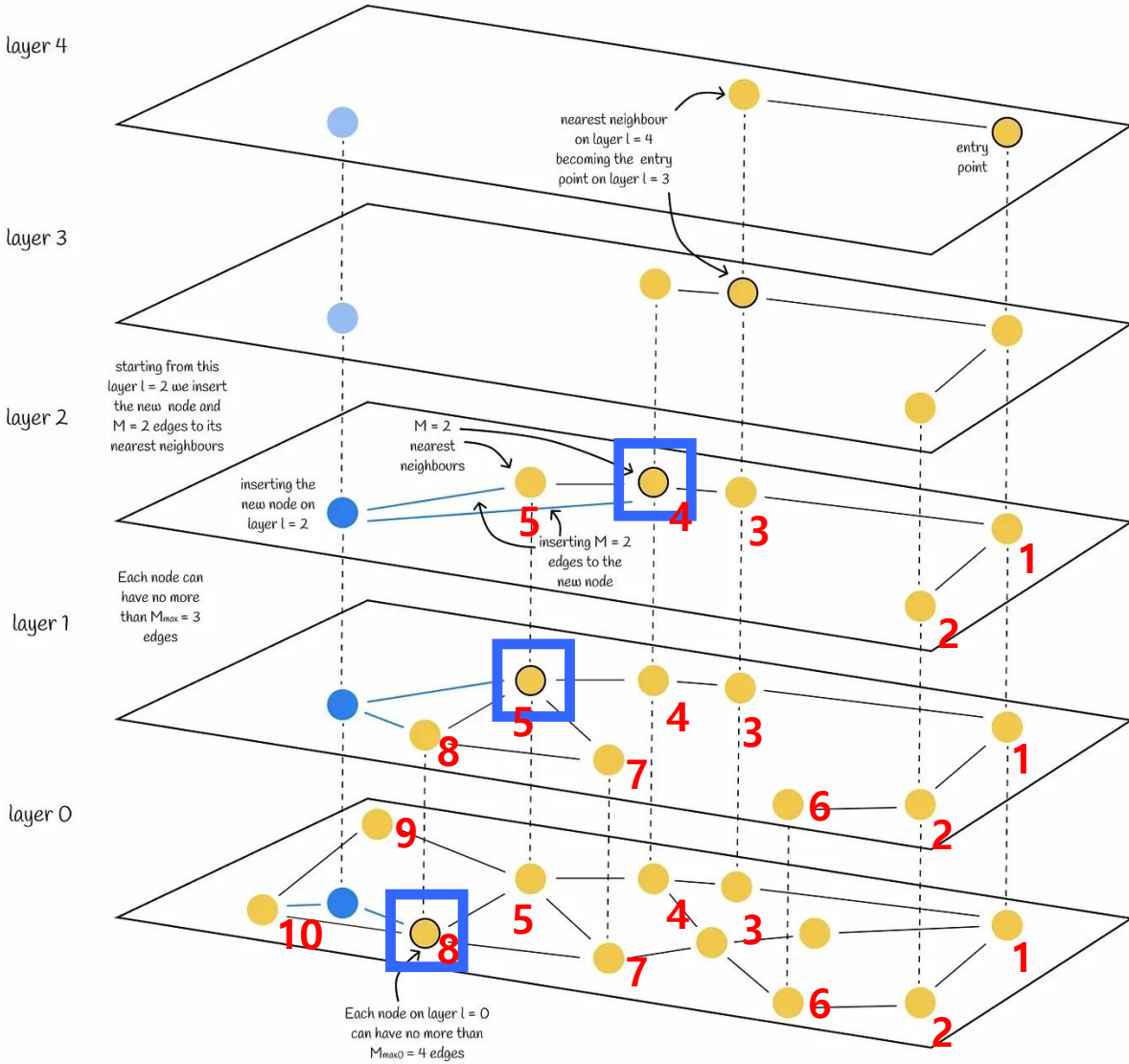
Preliminaries

HNSW

HNSW 그래프 Build (Practical Ver.)

- efConstruction = max heap의 크기 = 4
- [L=2]
  - 결과 queue: [(0.8, 노드 1), (0.7, 노드 3), (0.5, 노드 4), (0.2, 노드 5)]
  - 노드5에서 종료
- [L=1] 에 파란색 노드가 entry point에 삽입
  - 결과 queue: (0.4, 노드4), (0.3, 노드 7), (0.2, 노드 5), (0.1, 노드 8)]
  - 노드8에서 종료
- [L=0]
  - 결과 queue: [(0.2, 노드 5), (0.15, 노드 9), (0.1, 노드 8), (0.02, 노드10)]

➔ 노드10, 노드8과 edge 연결!!



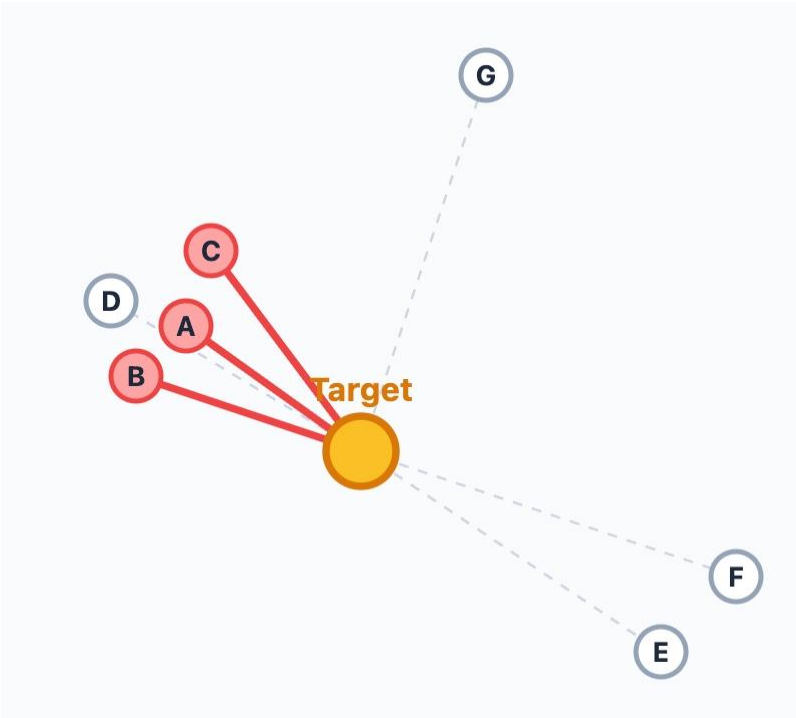
Preliminaries

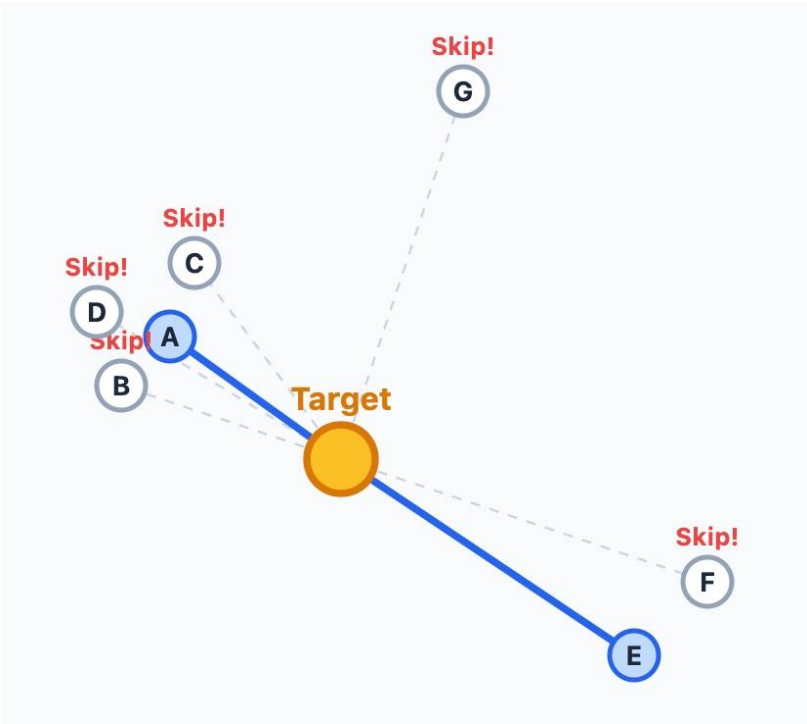
HNSW

결과 queue: [(0.2, 노드 5), (0.15, 노드 9), (0.1, 노드 8), (0.02, 노드10)]

HNSW 그래프 Build (Practical Ver.)

노드10, 노드8과 edge 연결!!...이면 좋겠지만, 실상은 그렇지 X





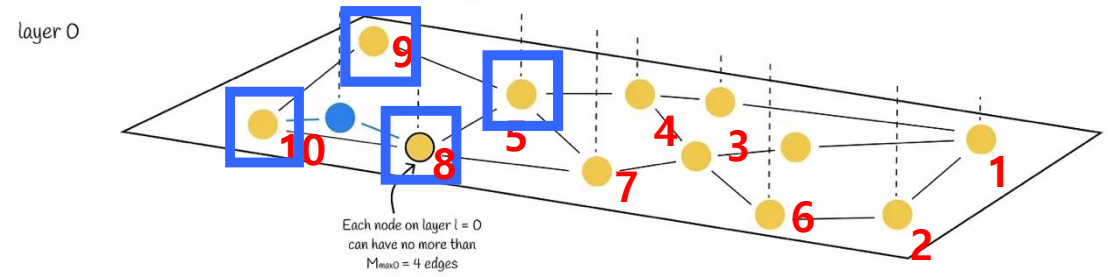
Preliminaries

HNSW

Heuristic Selection

- 결과 queue: [(0.2, 노드 5), (0.15, 노드 9), (0.1, 노드 8), (0.02, 노드10)]
- 채워야 하는 최종 queue (2자리): **[(), ()]**

1. 가장 가까운 노드 10 채움 ➔ **[(0.02, 노드10), ()]**
2. 노드8과 비교
   - **d(파란노드 , 노드8) < d(노드10, 노드8) 을 만족하는 경우만 채움**

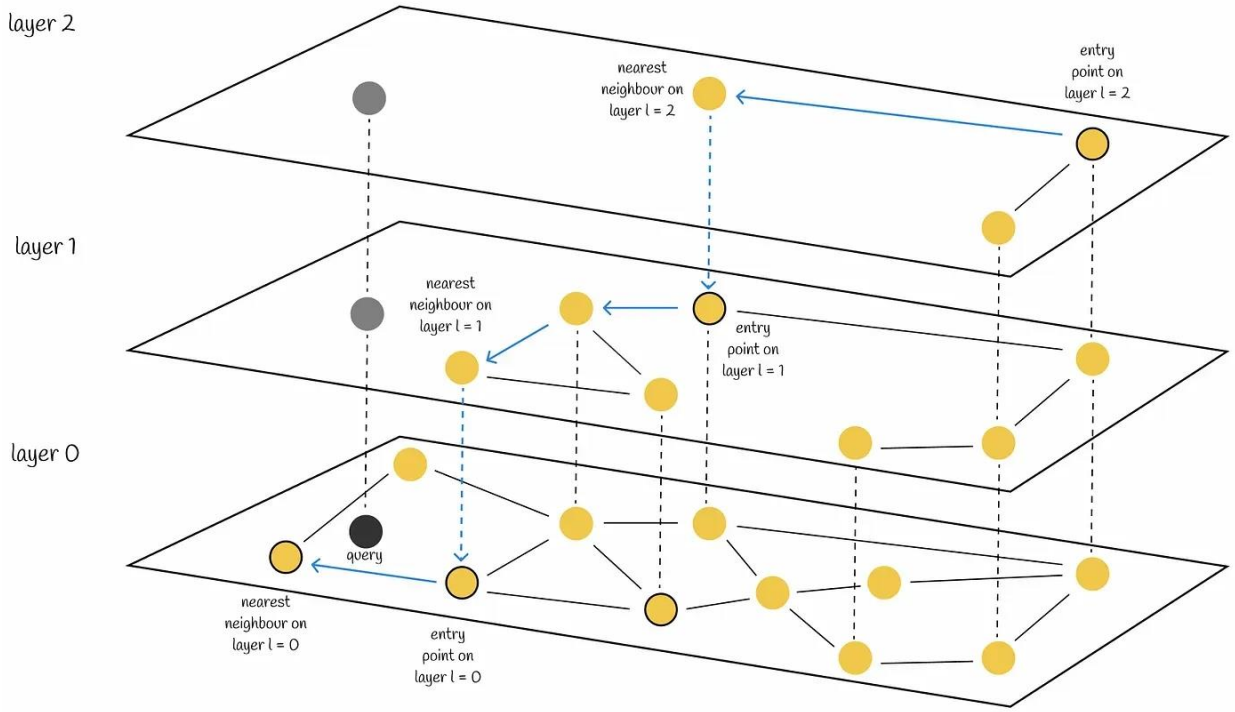


Preliminaries

HNSW

HNSW 탐색

- efSearch: 탐색할 때 사용하는 efConstruction 파라미터
- 탐색 시에는 연결이 아닌, topK개 반환



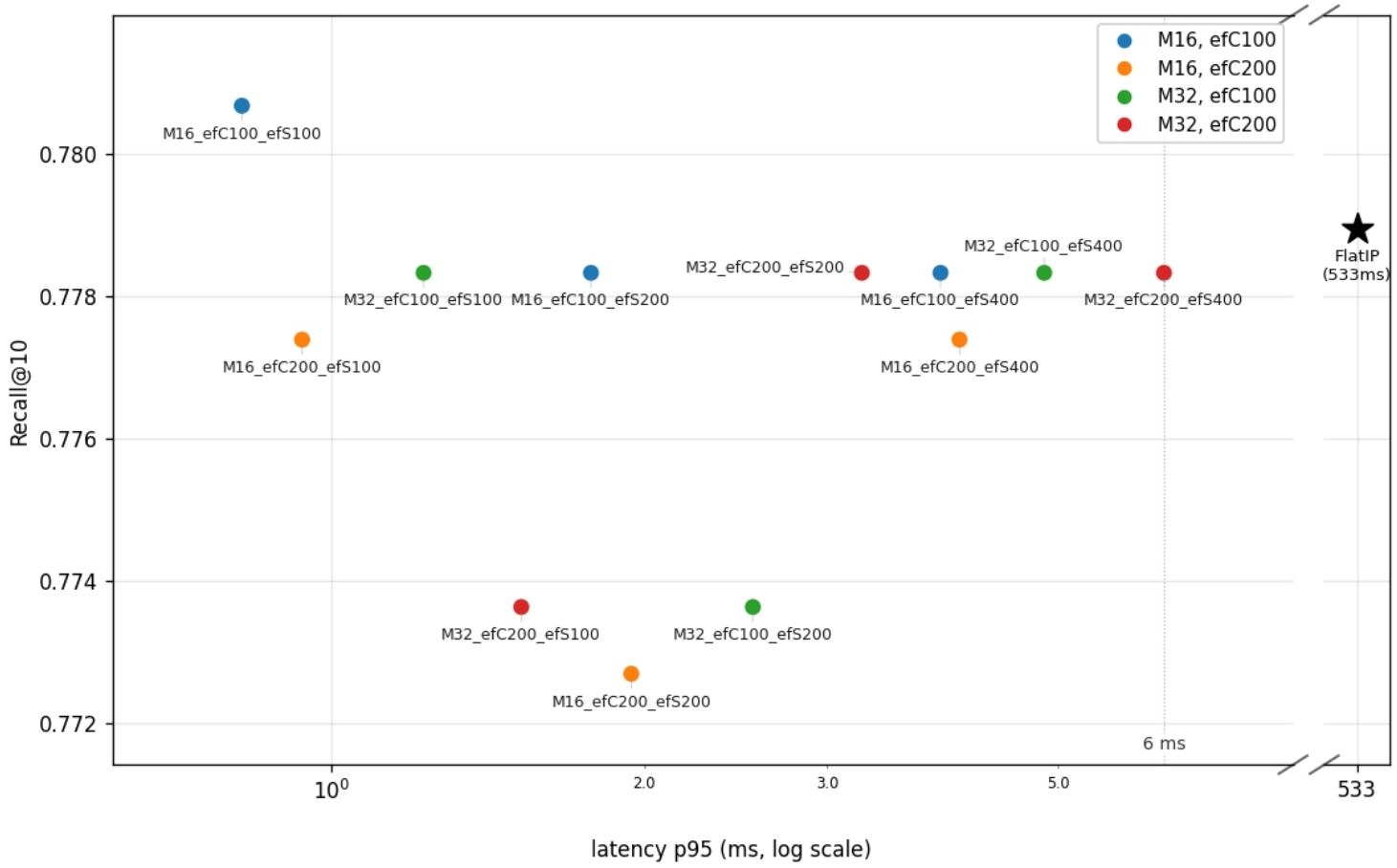
Preliminaries

HNSW

- M: 빌드 시 연결하는 edge 개수
- efC = efConstruction
- efS = efSearch

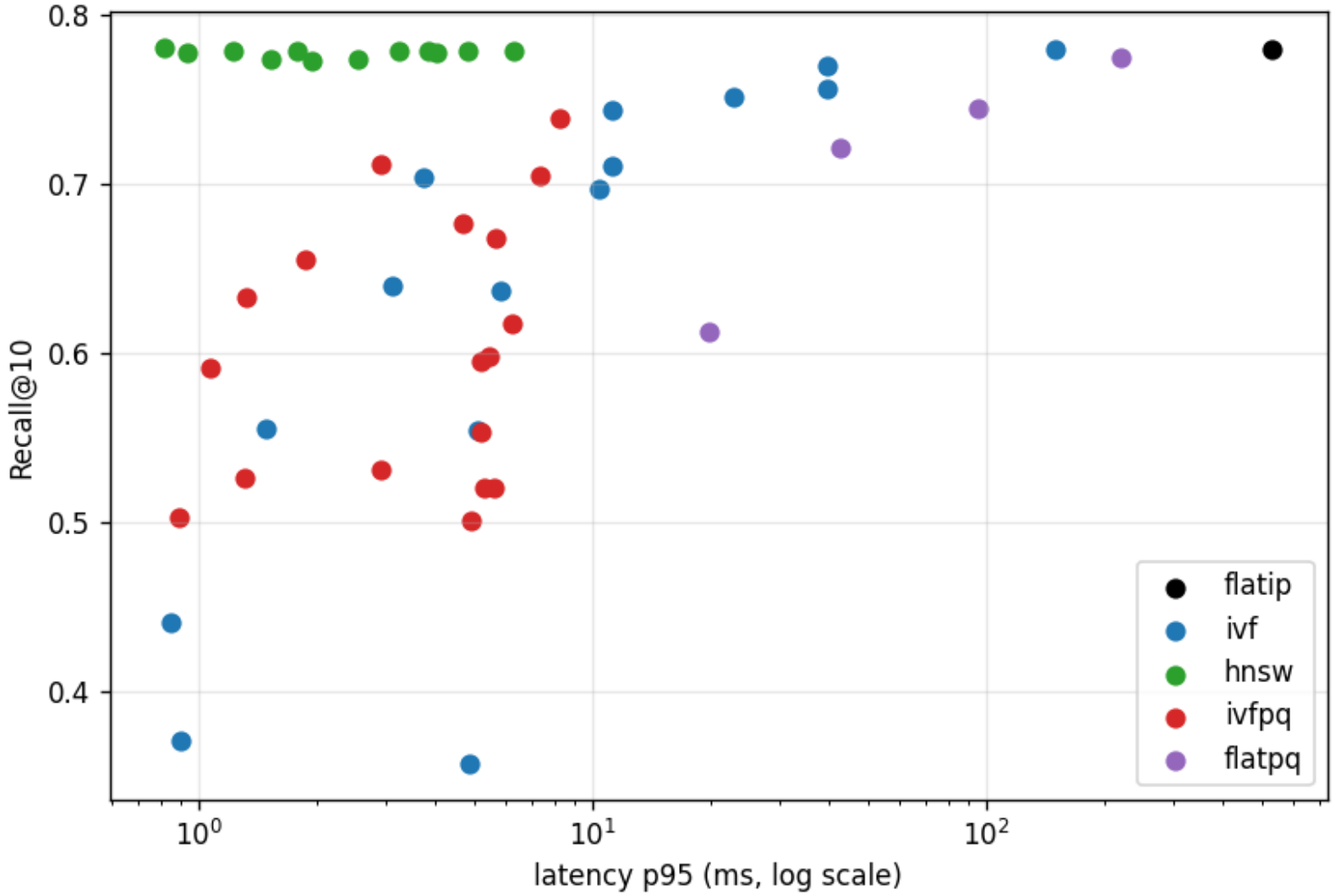
[결과]

- 성능 차이 모두 미미
- 같은 efC에서 efS 커질수록 latency 증가



Conclusion

최종 비교



Thank you

Q&A