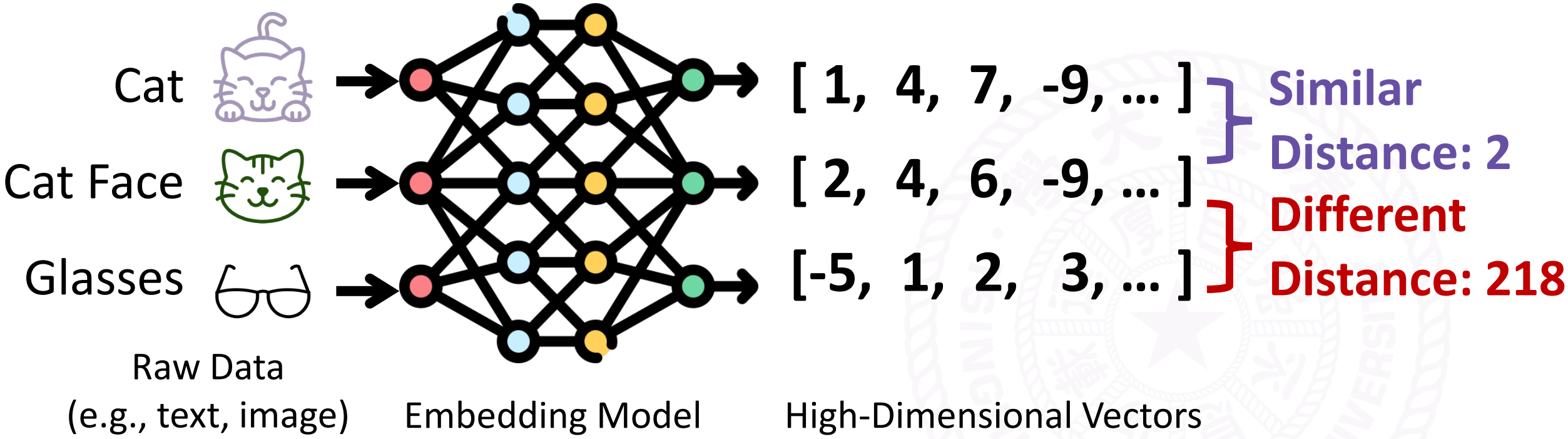
OSDI 25

Achieving **Low-Latency** Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD

Hao Guo, Youyou Lu

Tsinghua University

Vector: Powerful Data Representation



Vector distance reflects **data similarity**

Vector Search: Finding K Similar Vectors

Vector dataset

[1, 4, 7, -9, …]

[2, 4, 6, -9, …]

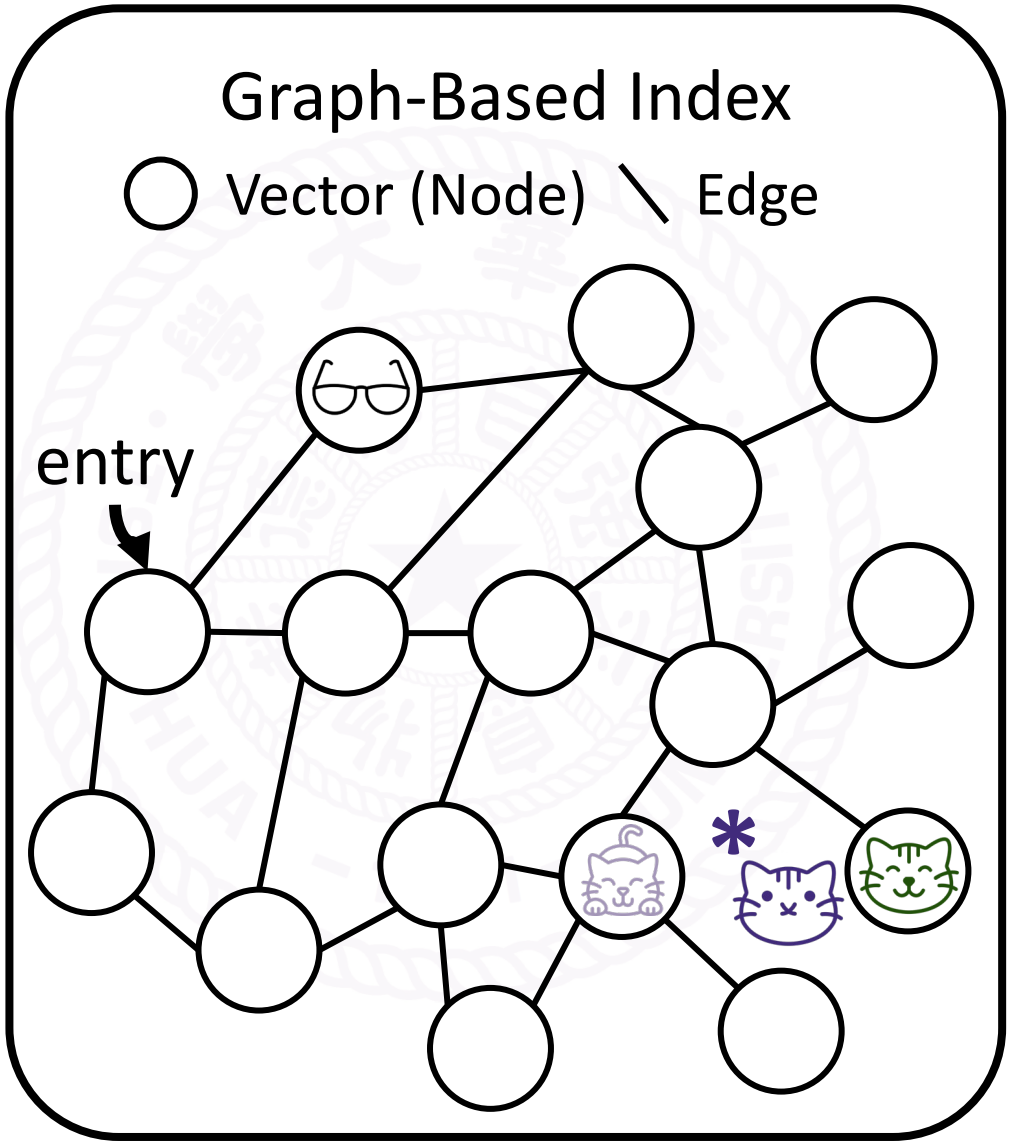
[-5, 1, 2, 3, …]

Build

Target (Query)

[3, 5, 7, -8, …]

Search



Accurate search (e.g., brute-force) is slow

Complexity: O(#vectors × #dims)

So, use approximate search (ANNS) instead

Vector Search: Finding K Similar Vectors

Vector dataset

[1, 4, 7, -9, …]

[2, 4, 6, -9, …]

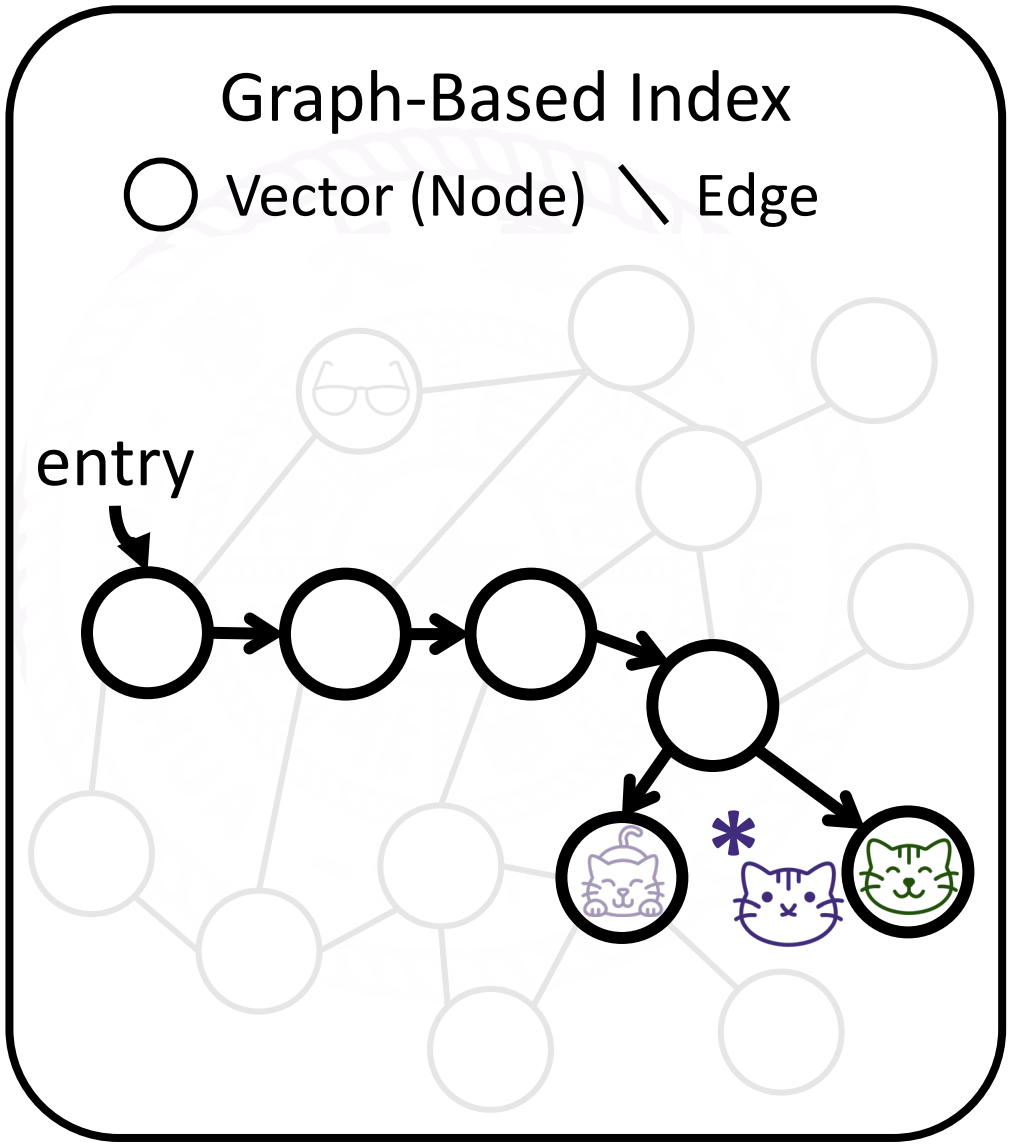
[-5, 1, 2, 3, …]

Build

Target (Query)

[3, 5, 7, -8, …]

Search



Accurate search (e.g., brute-force) is slow

Complexity: O(#vectors × #dims)

So, use approximate search (ANNS) instead

Low Latency Vector Search at Large Scale

This is good because:

- Latency demands of real-world workloads

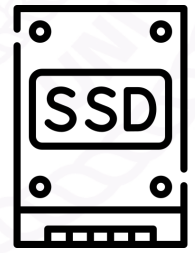
Bing: 100B vectors (40TB), 10ms search

Alibaba.com: 2B vectors (several TB), ~10ms search (98% acc)

- Same latency budget, better accuracy

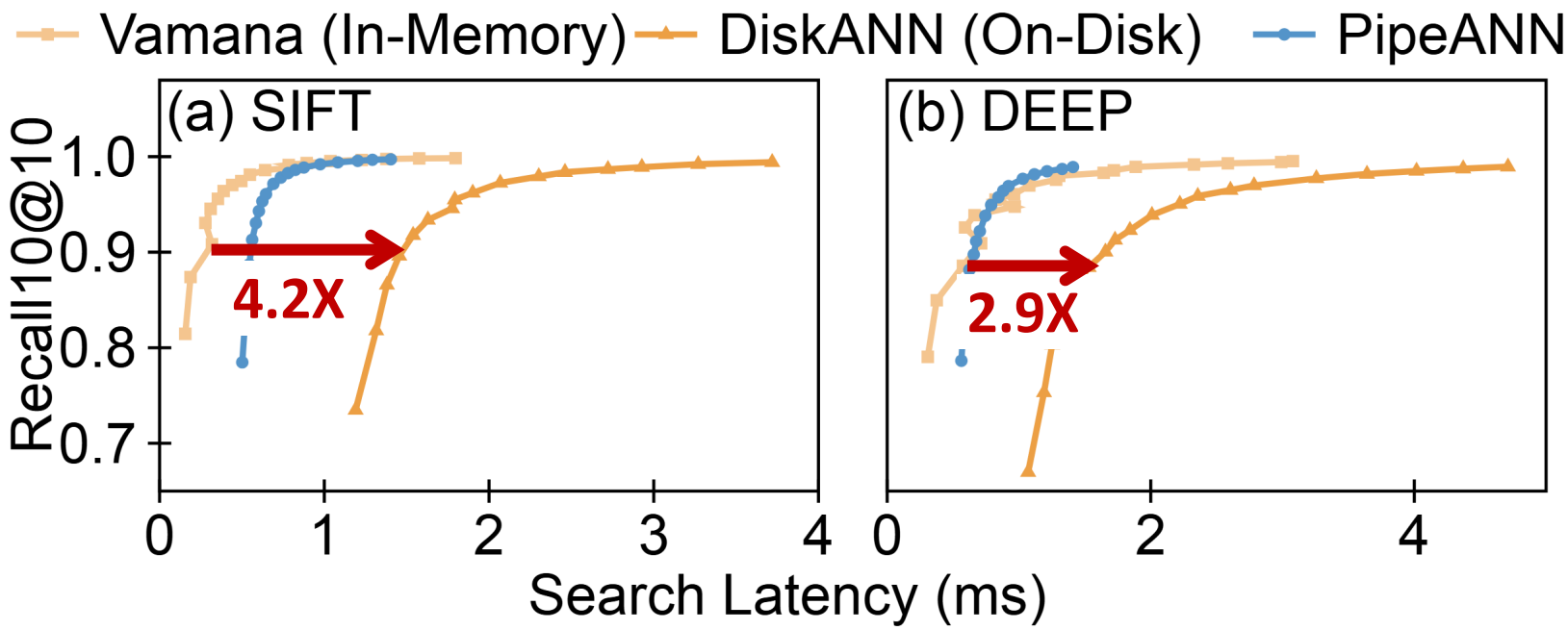
This is possible because:

- **Large scale**: Use second-tier storage (SSD in typical)
- Can we achieve **low latency** simultaneously?



~~Low Latency~~ Vector Search at Large Scale

On-SSD ANNS has much higher latency than in-memory ANNS…

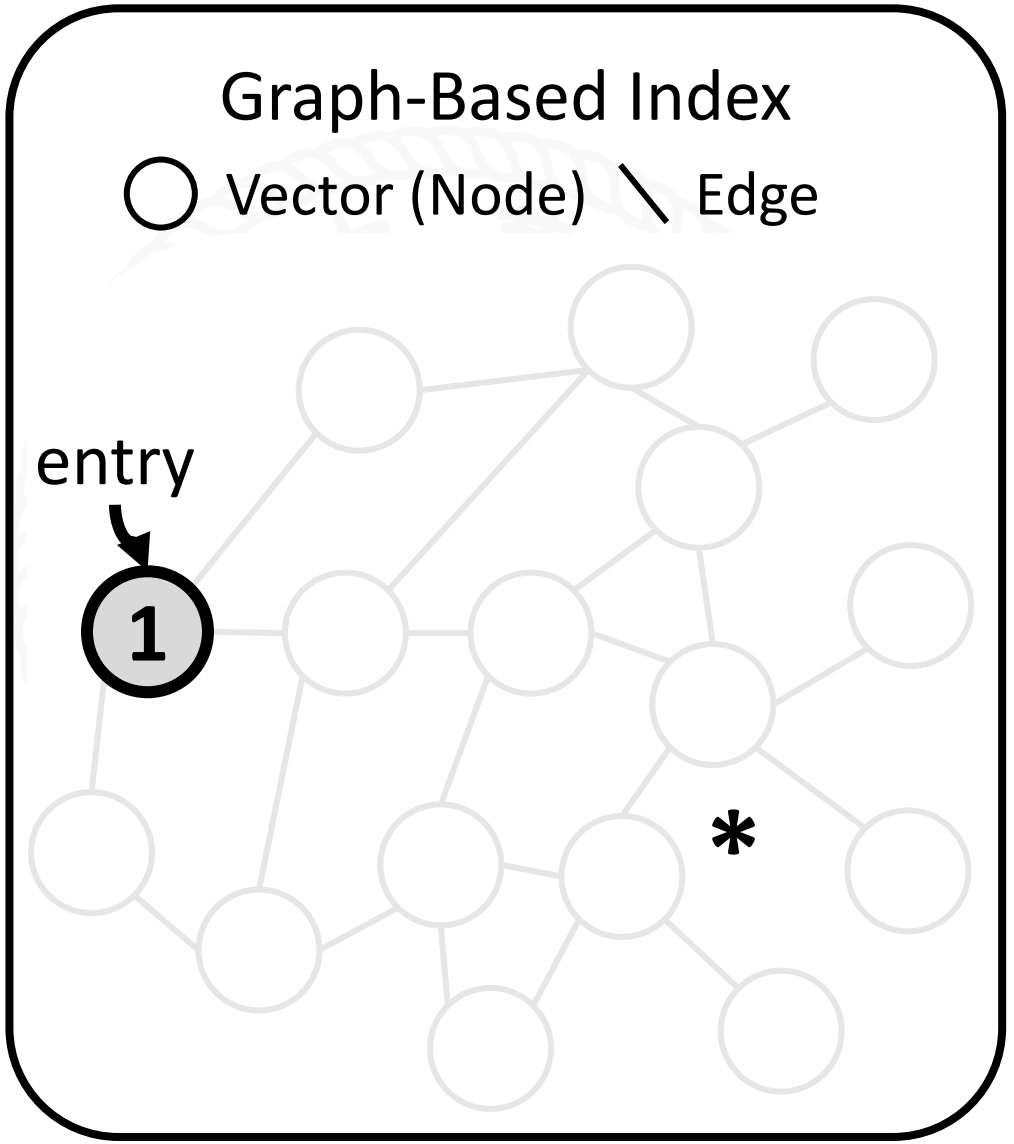


Our finding: Mismatch between
Search Algorithm and **SSD**

Best-First Search Algorithm

Candidate pool: **{ 1 }**

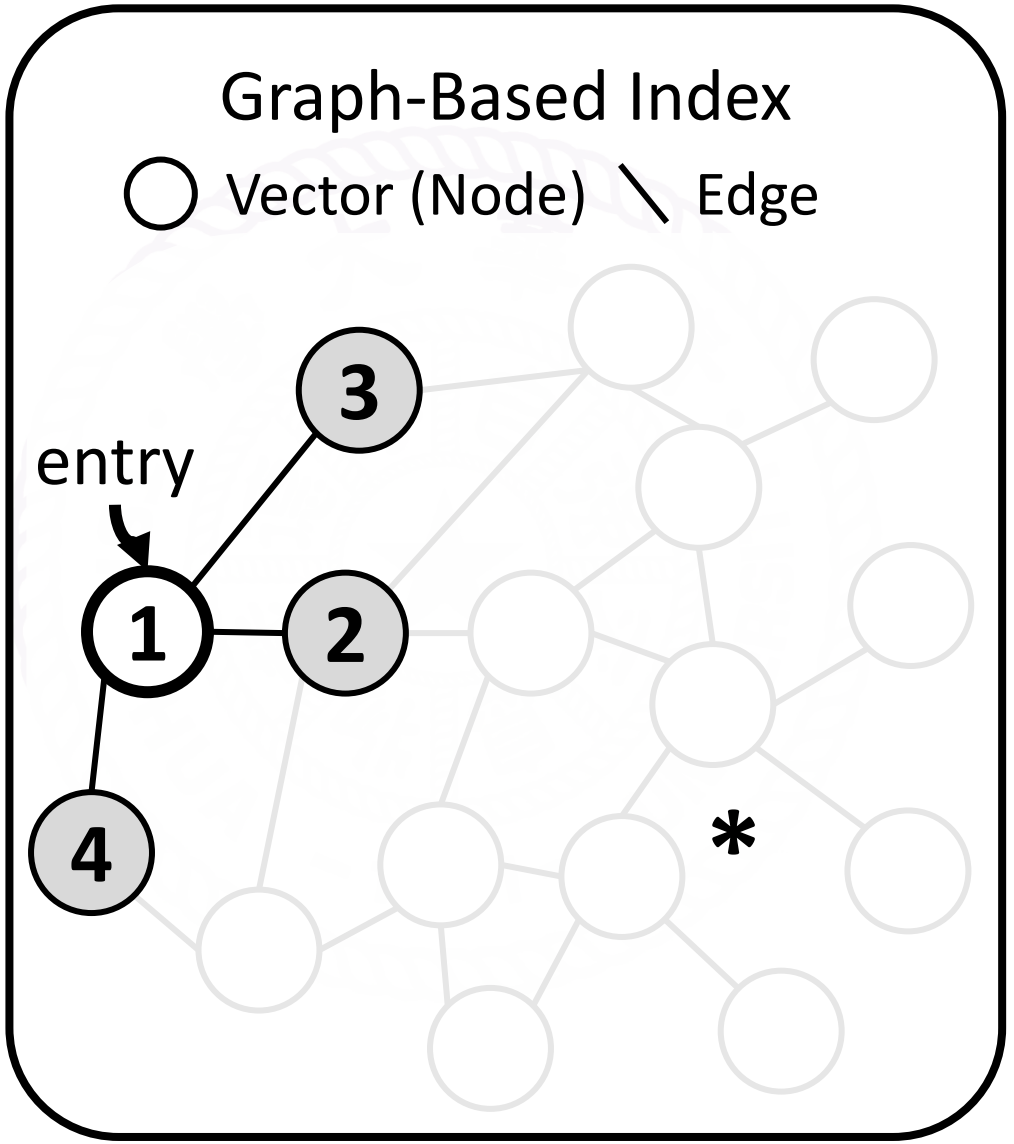
- **Select** the nearest unvisited vector **1** in the *candidate pool*



Best-First Search Algorithm

Candidate pool: **{ 1 }**

- **Select** the nearest unvisited vector **1** in the *candidate pool*
- **Read** its neighbor IDs **{** 2, 3, 4 **}**

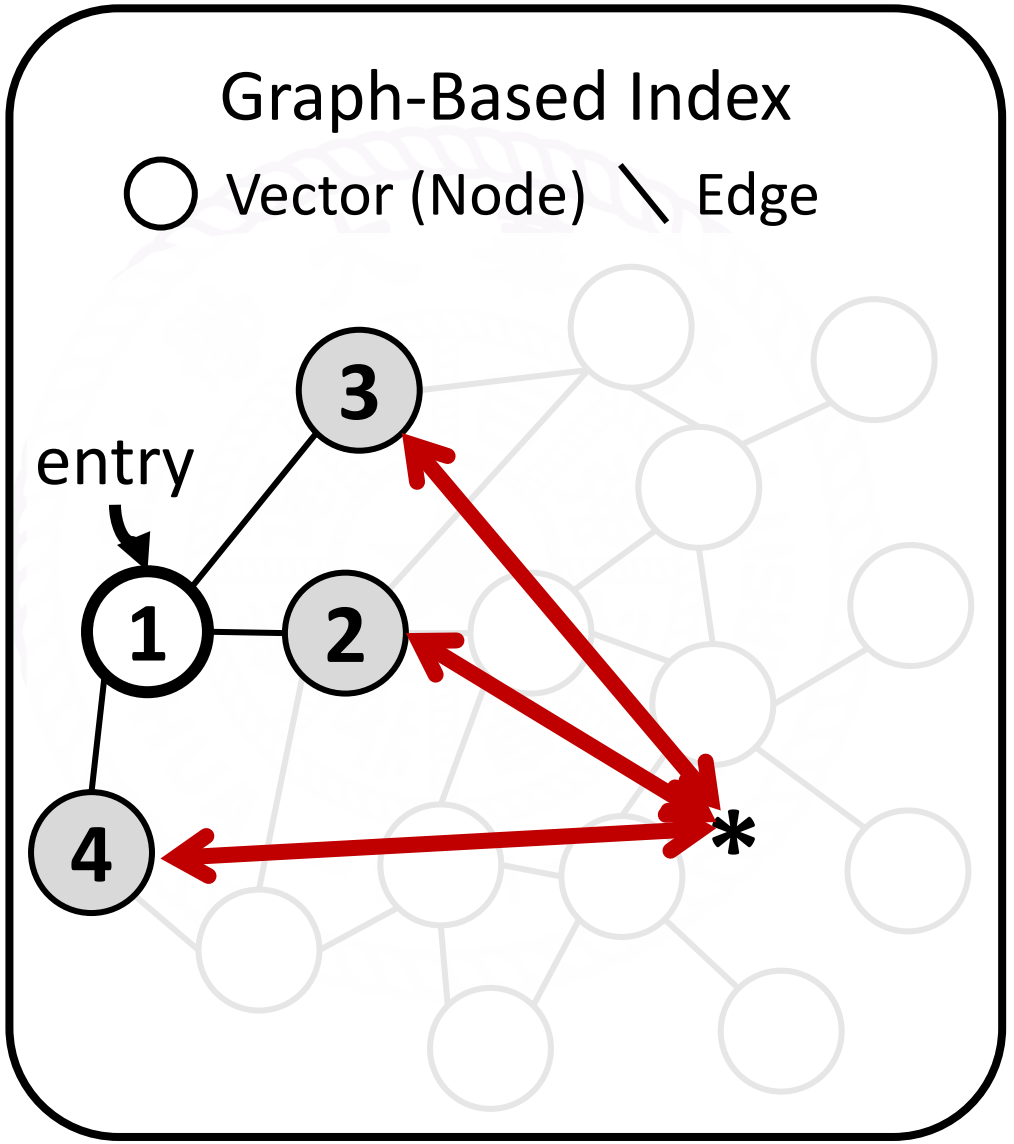


Best-First Search Algorithm

Candidate pool: **{ 1 }**

- **Select** the nearest unvisited vector **1** in the *candidate pool*
- **Read** its neighbor IDs **{** 2, 3, 4 **}**
- **Compare** its neighbors' distances to the target vector ∗

  SSD read is not required, due to in-memory compressed vectors.

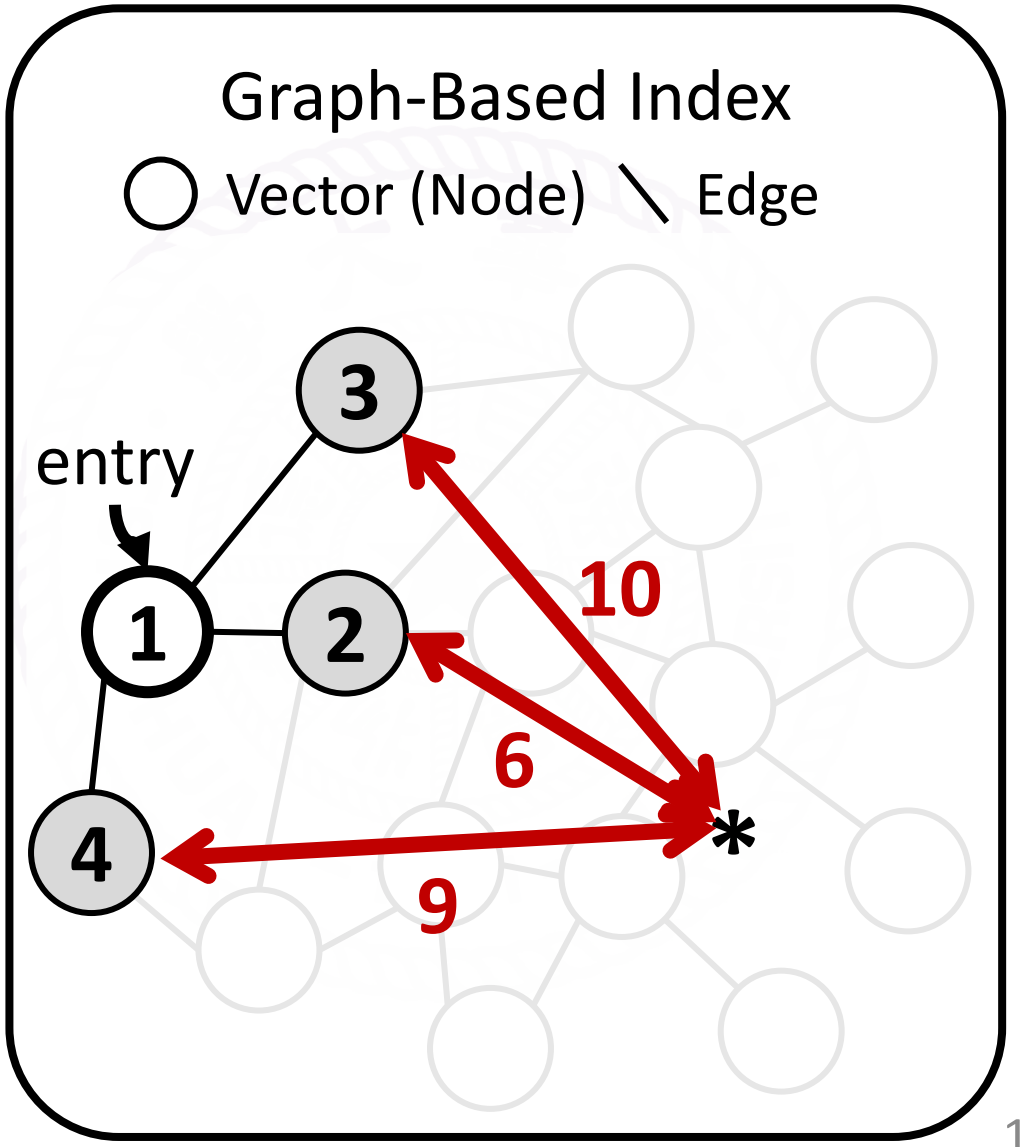


Best-First Search Algorithm

Candidate pool: **{** 2, 3, 4, ~~1~~ **}**

- **Select** the nearest unvisited vector **1** in the *candidate pool*
- **Read** its neighbor IDs **{** 2, 3, 4 **}**
- **Compare** its neighbors' distances to the target vector ∗
- **Add** its neighbors to the candidate pool, sort them, mark **1** as visited

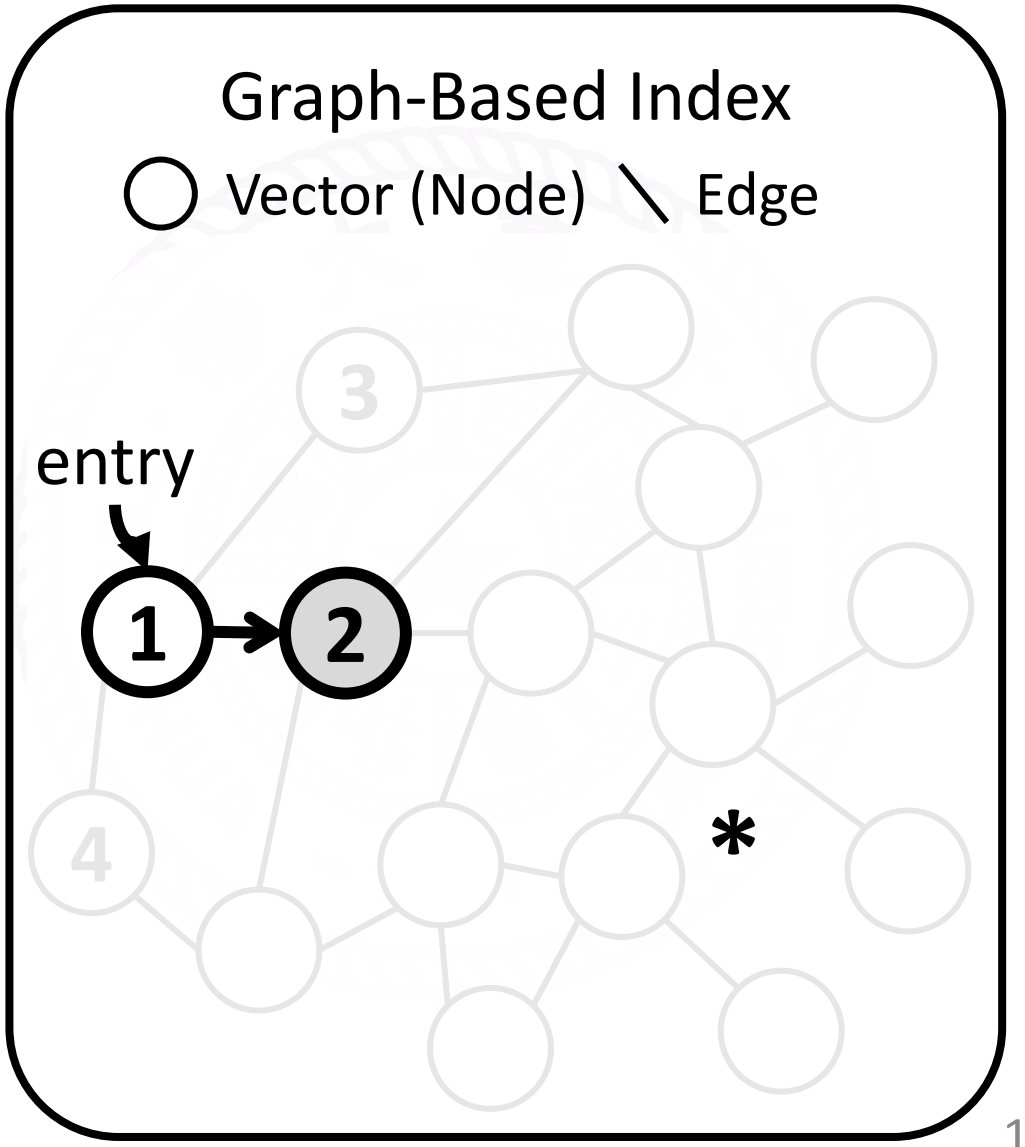
Repeat this process



Best-First Search Algorithm

Candidate pool: **{ 2**, 3, 4, ~~1~~ **}**

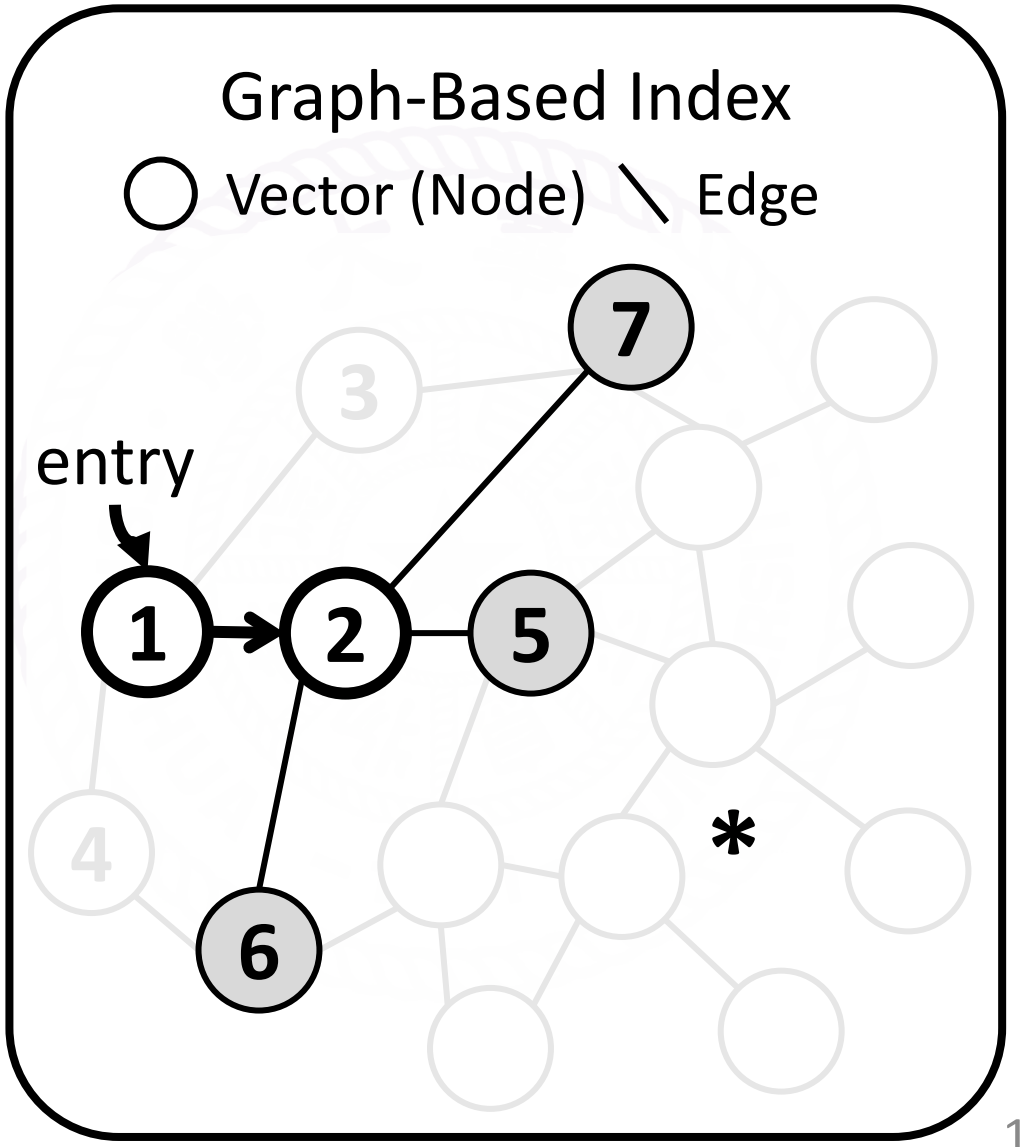
- **Select** the nearest unvisited vector **2** in the *candidate pool*



Best-First Search Algorithm

Candidate pool: **{ 2**, 3, 4, ~~1~~ **}**

- **Select** the nearest unvisited vector **2** in the *candidate pool*
- **Read** its neighbor IDs **{** 5, 6, 7 **}**

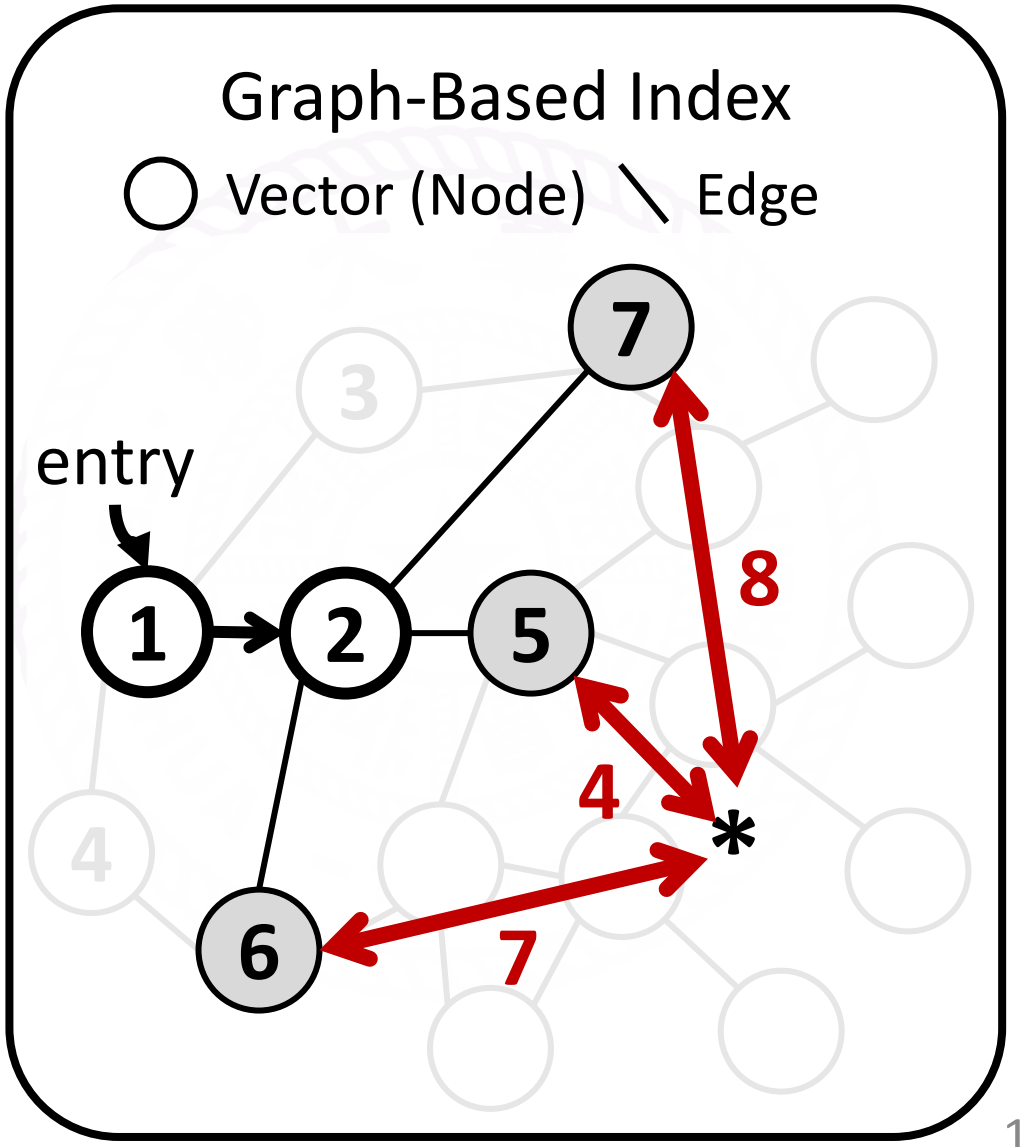


Best-First Search Algorithm

Candidate pool: **{** 5, ~~2~~, 6, 7, 3, 4, ~~1~~ **}**

- **Select** the nearest unvisited vector **2** in the *candidate pool*
- **Read** its neighbor IDs **{** 5, 6, 7 **}**
- **Compare** its neighbors' distances to the target vector ∗
- **Add** its neighbors to the candidate pool, sort them, mark **2** as visited

Repeat this process

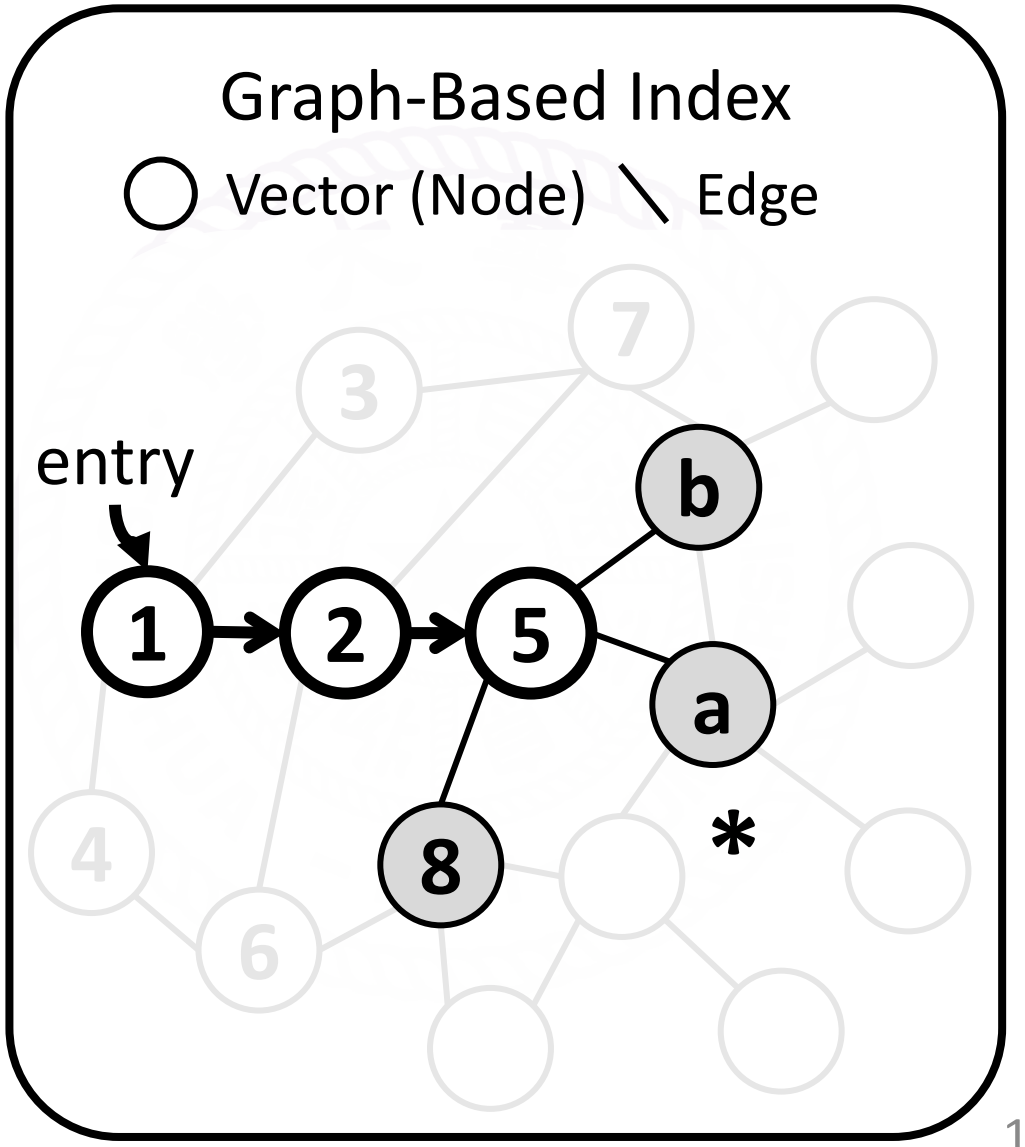


Best-First Search Algorithm

Candidate pool: **{** a, **5**, b, 8, ~~2~~, 6, 7 **}**

- **Select** the nearest unvisited vector **5** in the *candidate pool*
- **Read** its neighbor IDs **{** 8, a, b **}**
- **Compare** its neighbors' distances to the target vector ∗
- **Add** its neighbors to the candidate pool, sort them, mark **5** as visited

Repeat this process

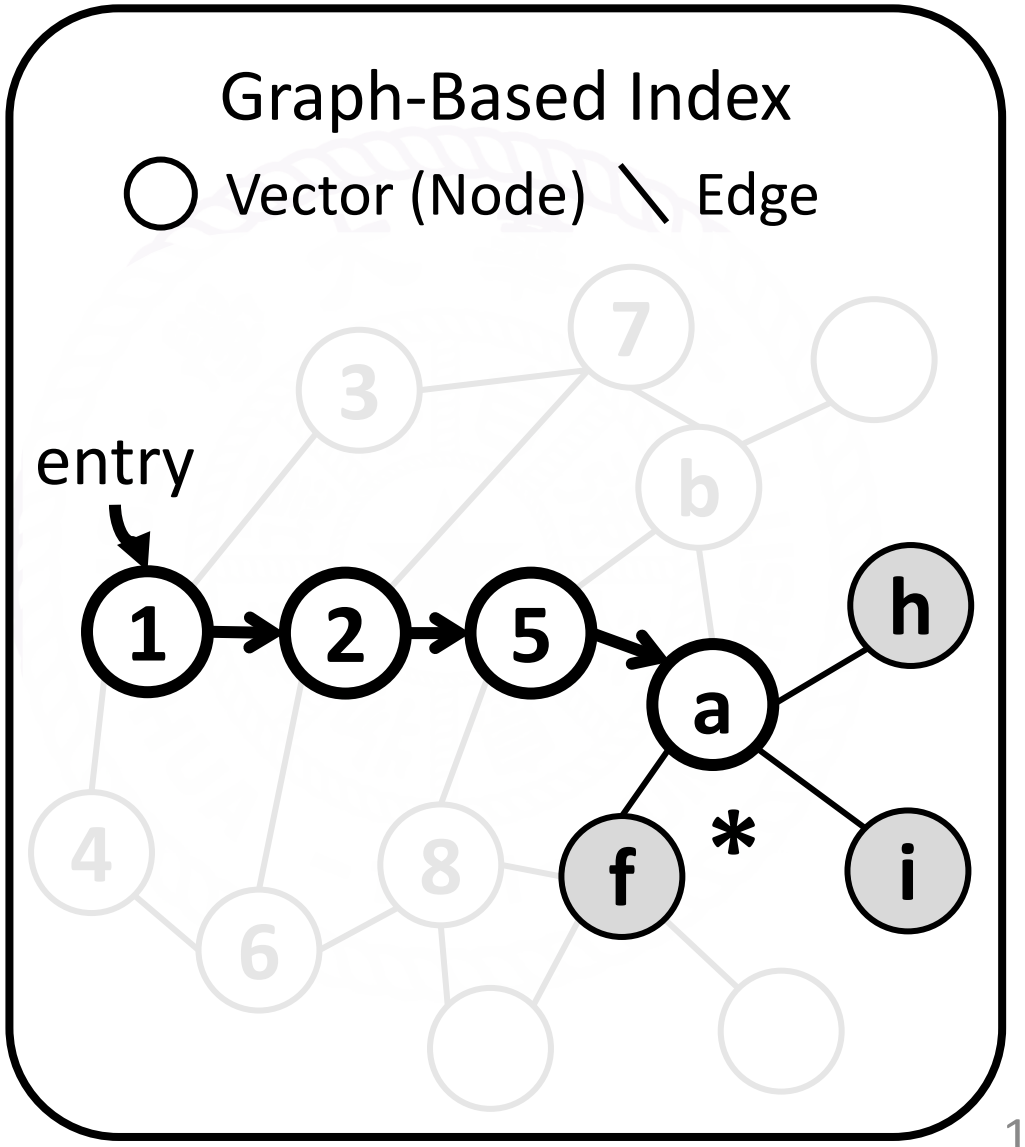


Best-First Search Algorithm

Candidate pool: **{ a**, f, i, h, ~~5~~, b, 8 **}**

- **Select** the nearest unvisited vector **a** in the *candidate pool*
- **Read** its neighbor IDs **{** f, h, i **}**
- **Compare** its neighbors' distances to the target vector ∗
- **Add** its neighbors to the candidate pool, sort them, mark **a** as visited

Repeat this process

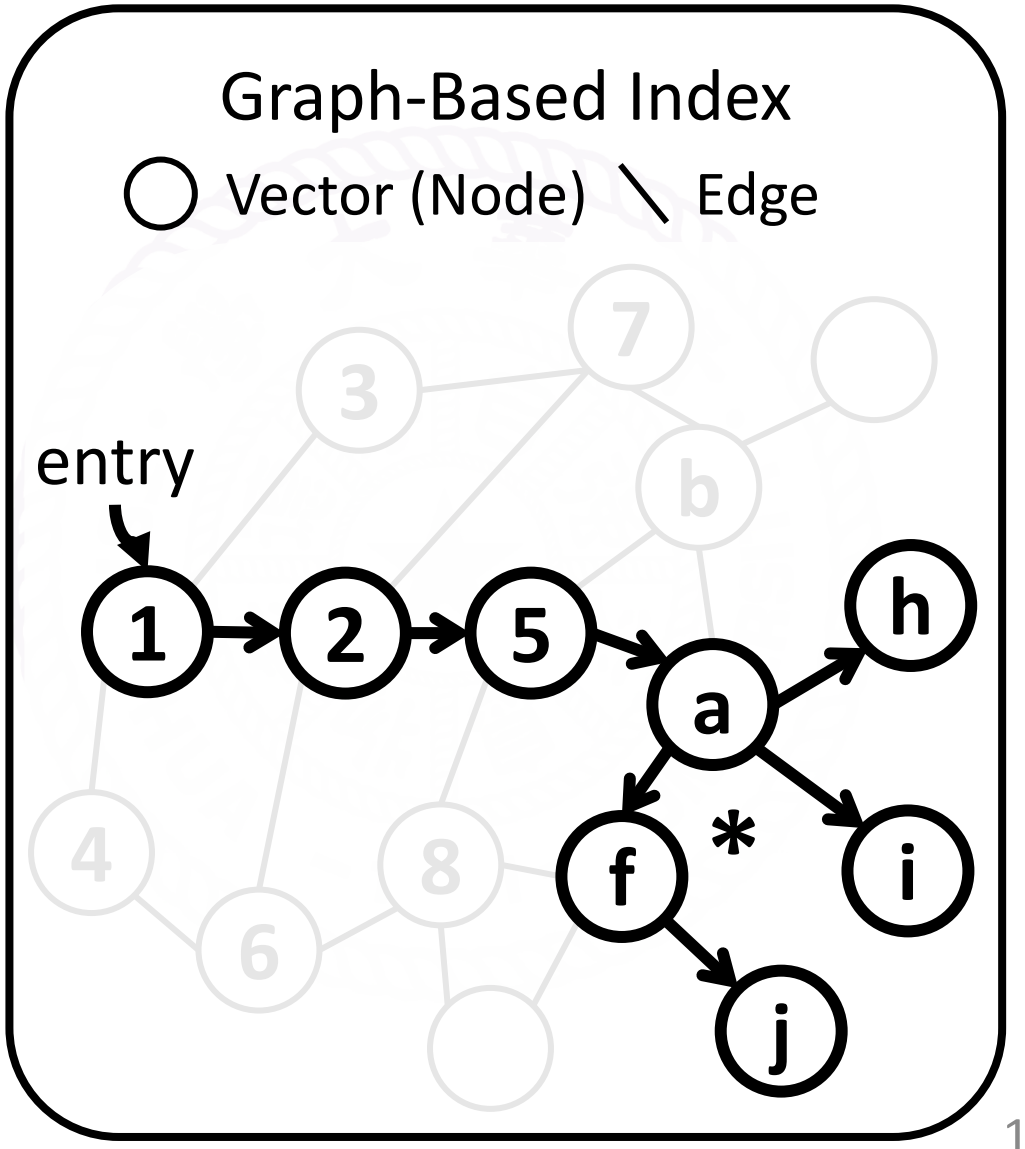


Best-First Search Algorithm

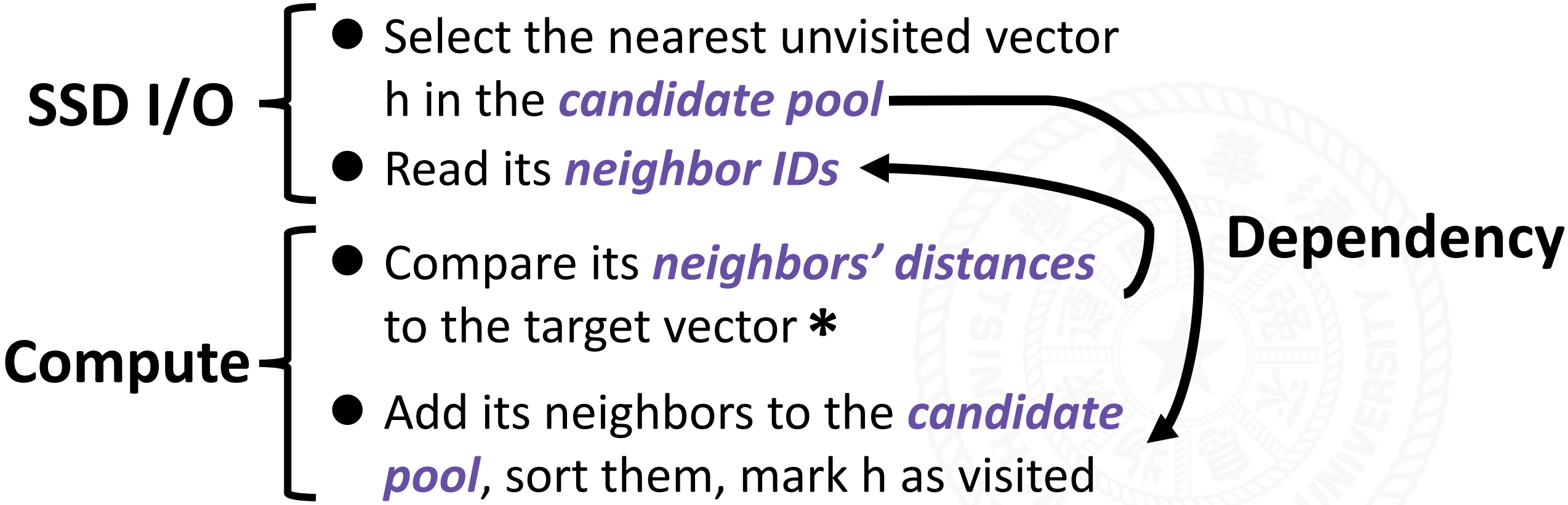
Candidate pool: **{** ~~a, f, i, j, h, 5, b~~ **}**

- **Select** the nearest unvisited vector **h** in the *candidate pool*
- **Read** its neighbor IDs
- **Compare** its neighbors' distances to the target vector ∗
- **Add** its neighbors to the candidate pool, sort them, mark **h** as visited

Repeat until all vectors are visited



~~Low Latency~~ Due to Dependency

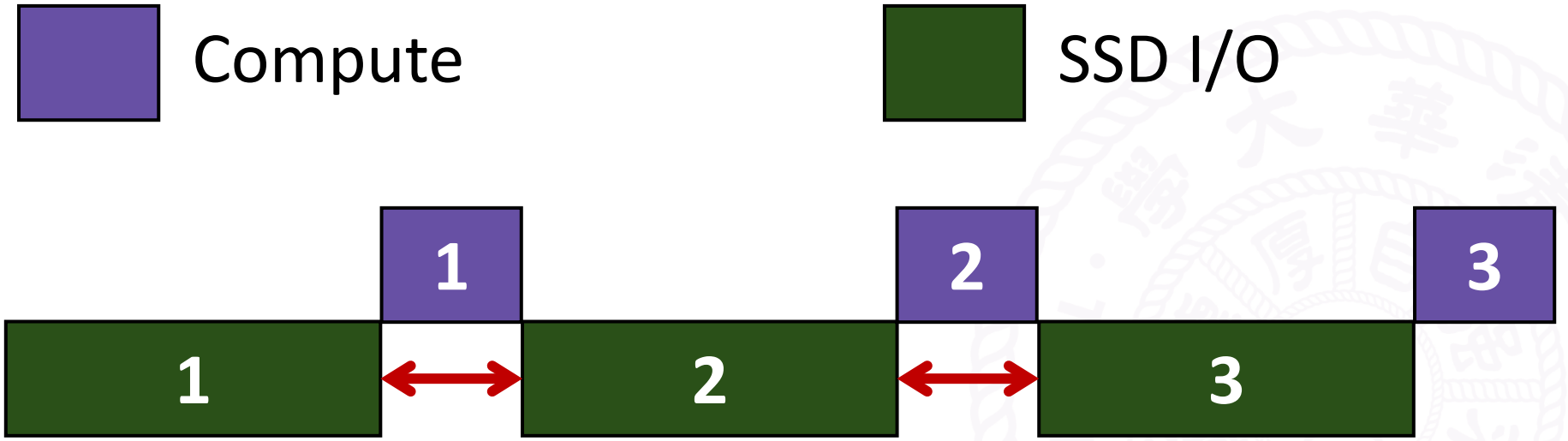


Compute depends on I/O for **neighbor ID**

I/O depends on compute for **best candidate**

~~Low Latency~~ Due to Dependency

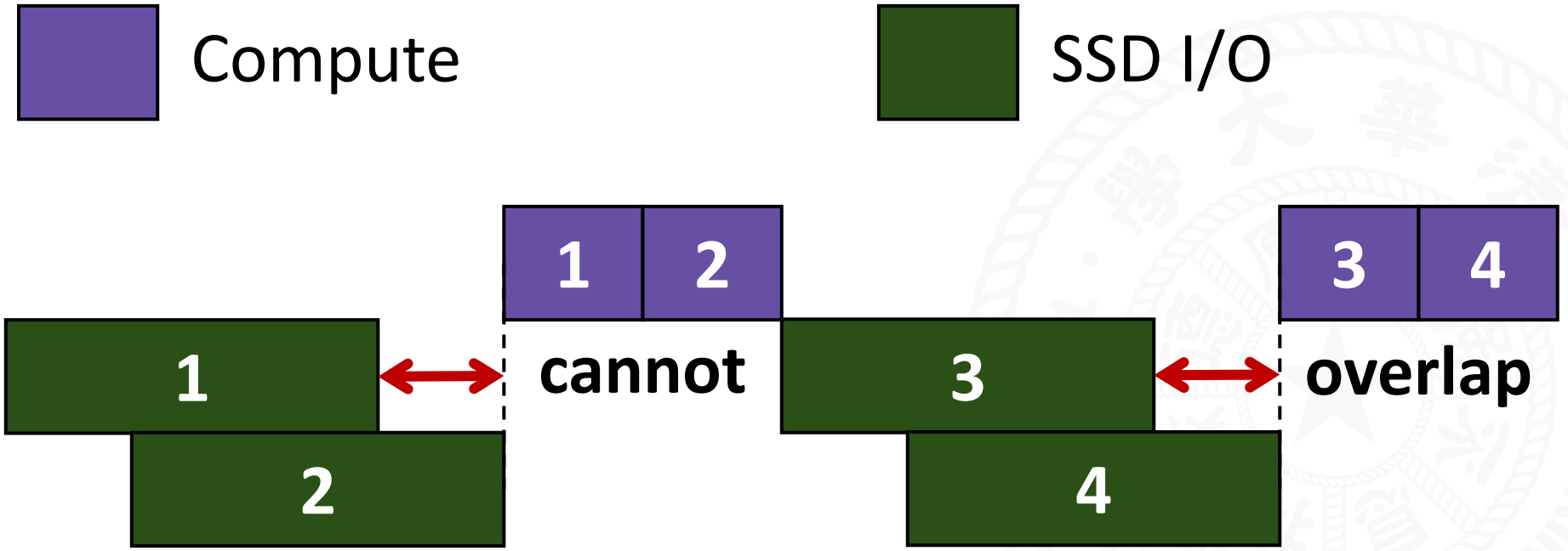
Best-first search (1 candidate at a time)



Compute and I/O **cannot fully overlap**…

~~Low Latency~~ Due to Dependency

Best-first search (2 candidates at a time)

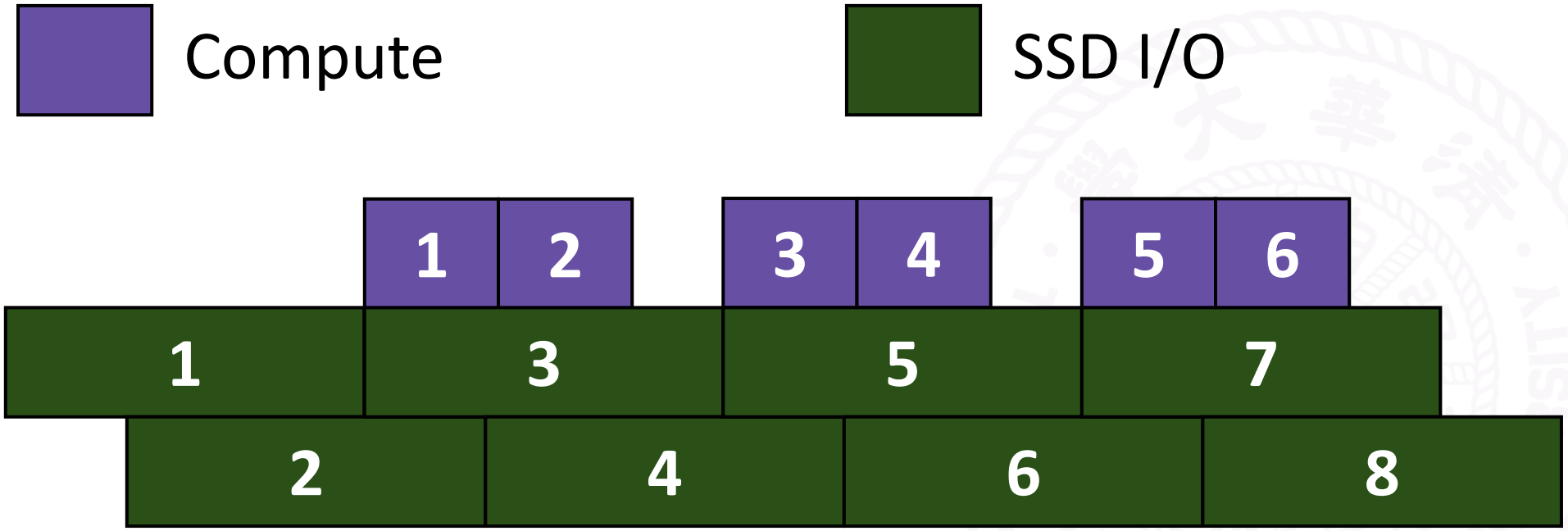


Compute and I/O **cannot fully overlap**…

SSD I/O pipeline **cannot be fully utilized…**

Break the Dependency… Is it Possible?

Our approach: PipeSearch (I/O pipeline width = 2)



Idea: decide SSD I/O ***independently***, regardless of previous compute and I/O

Put the Idea into Vector Search…

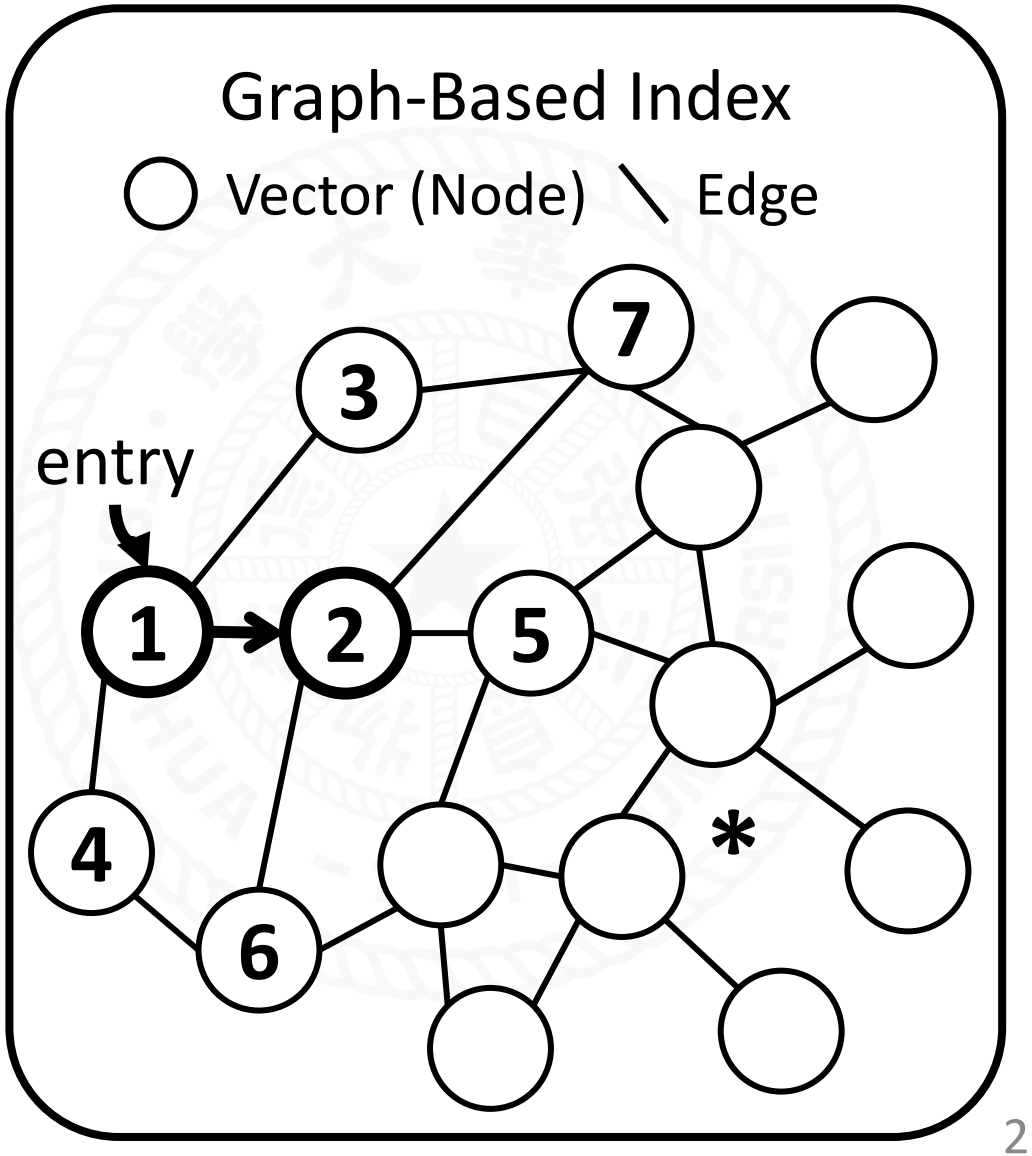
Idea: decide SSD I/O ***independently***, regardless of ongoing compute and I/O

This is viable because…

- Select the nearest unvisited vector **5** in the ***candidate pool***

In memory, we can directly decide I/O using this!

Candidate pool: **{ 5, ~~2~~, 6, 7, 3, 4, ~~1~~ }**



Put the Idea into Vector Search…

Idea: decide SSD I/O ***independently***, regardless of ongoing compute and I/O

Th

- Select …
  5 in the …

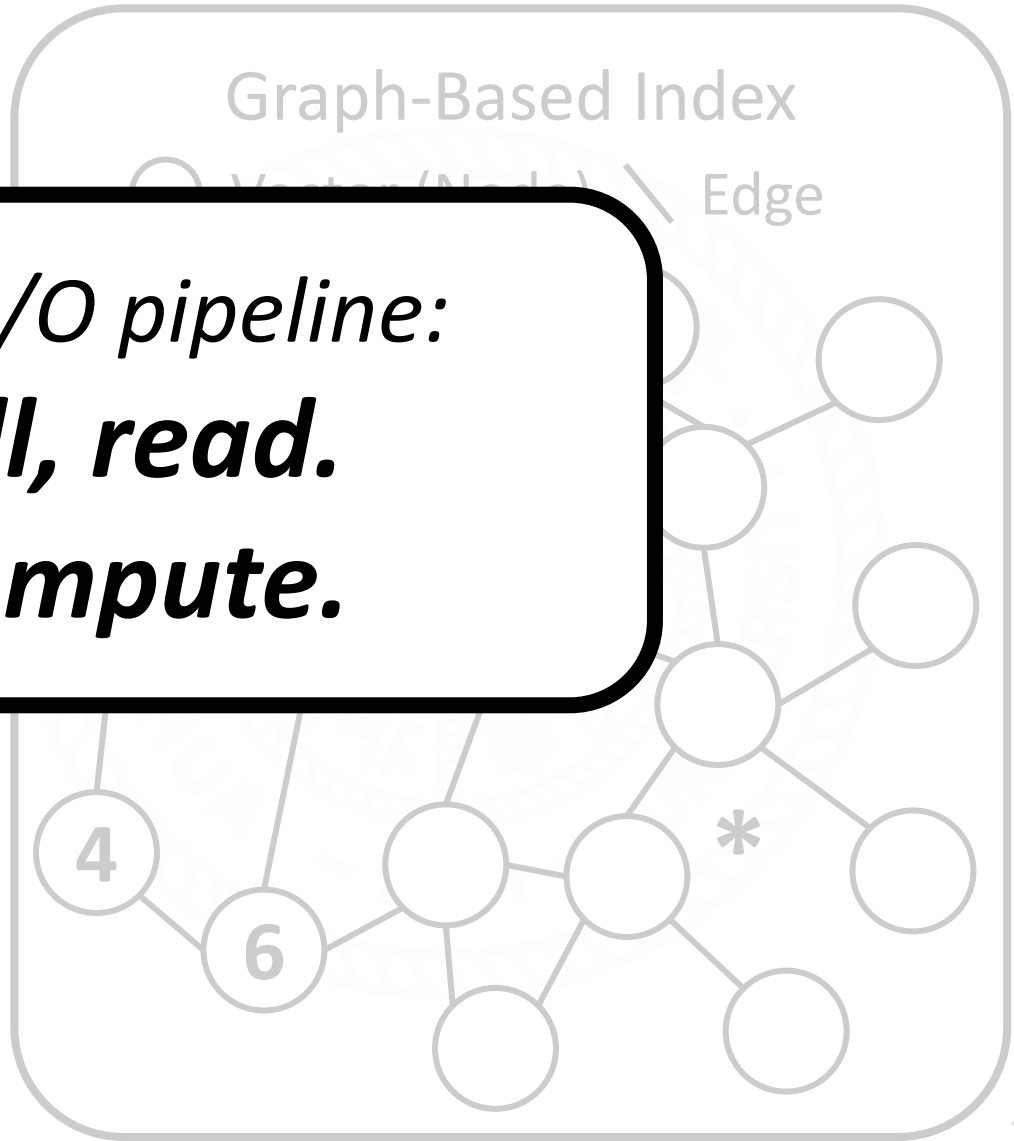
In memory, we can directly decide I/O using this!

Candidate pool: **{ 5, ~~2~~, 6, 7, 3, 4, ~~1~~ }**

Principle to saturate I/O pipeline:

Pipeline not full, read.

Pipeline full, compute.

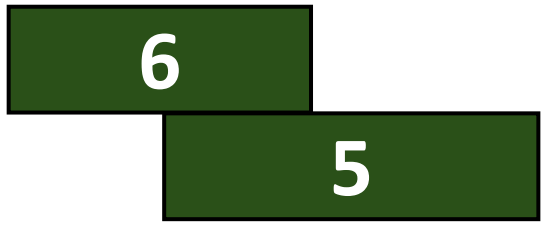


Put the Idea into Vector Search…

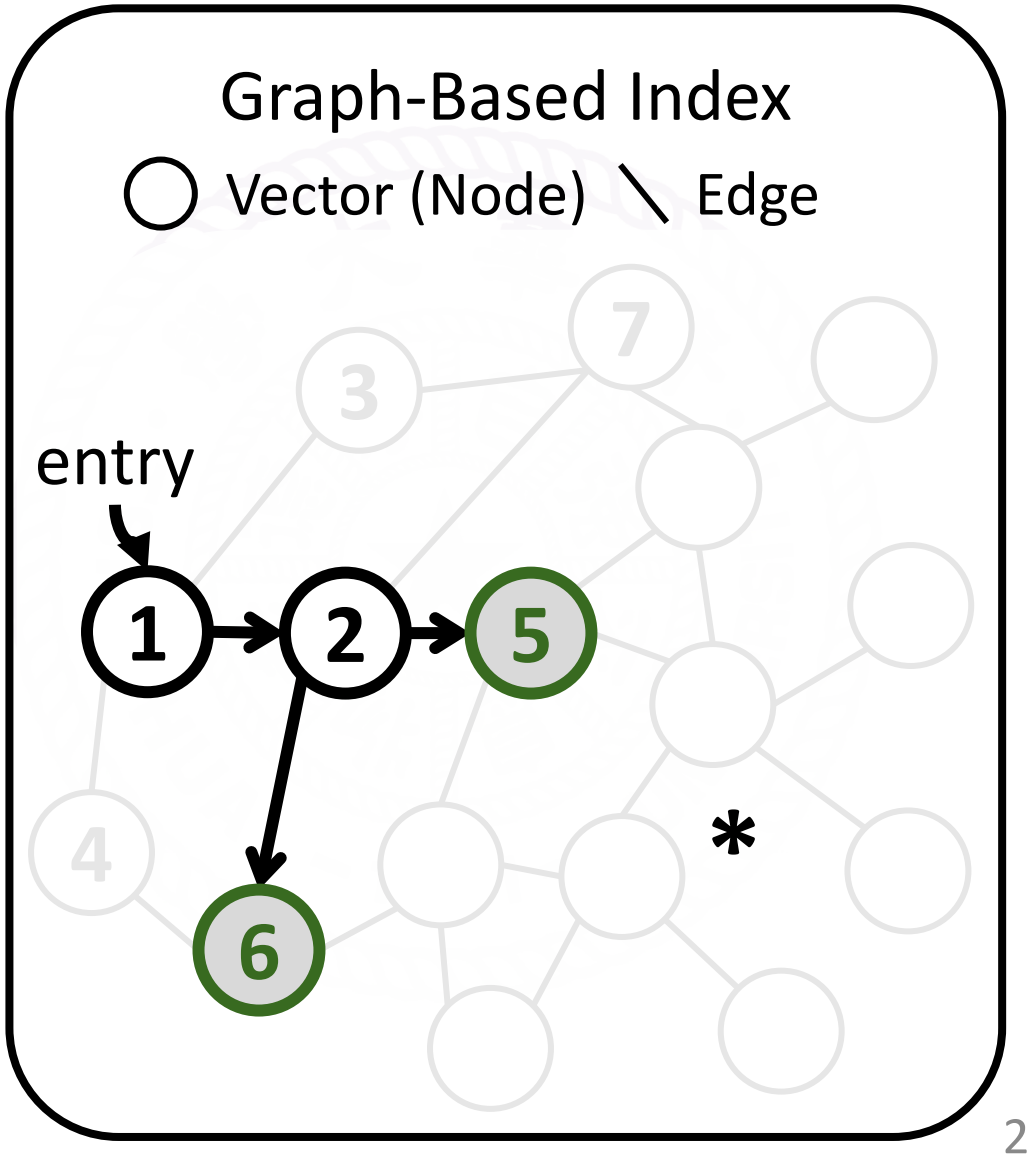
Pipeline not full, read.

Pipeline full, compute.

- Select the nearest unvisited vectors **5** and **6** in the *candidate pool*
- Async read **5** and **6**



Candidate pool: { **5,** ~~2~~, **6,** 7, 3, 4, ~~1~~ }

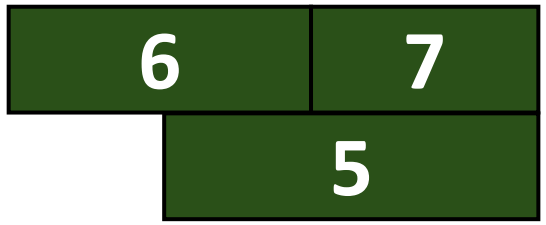


Put the Idea into Vector Search…

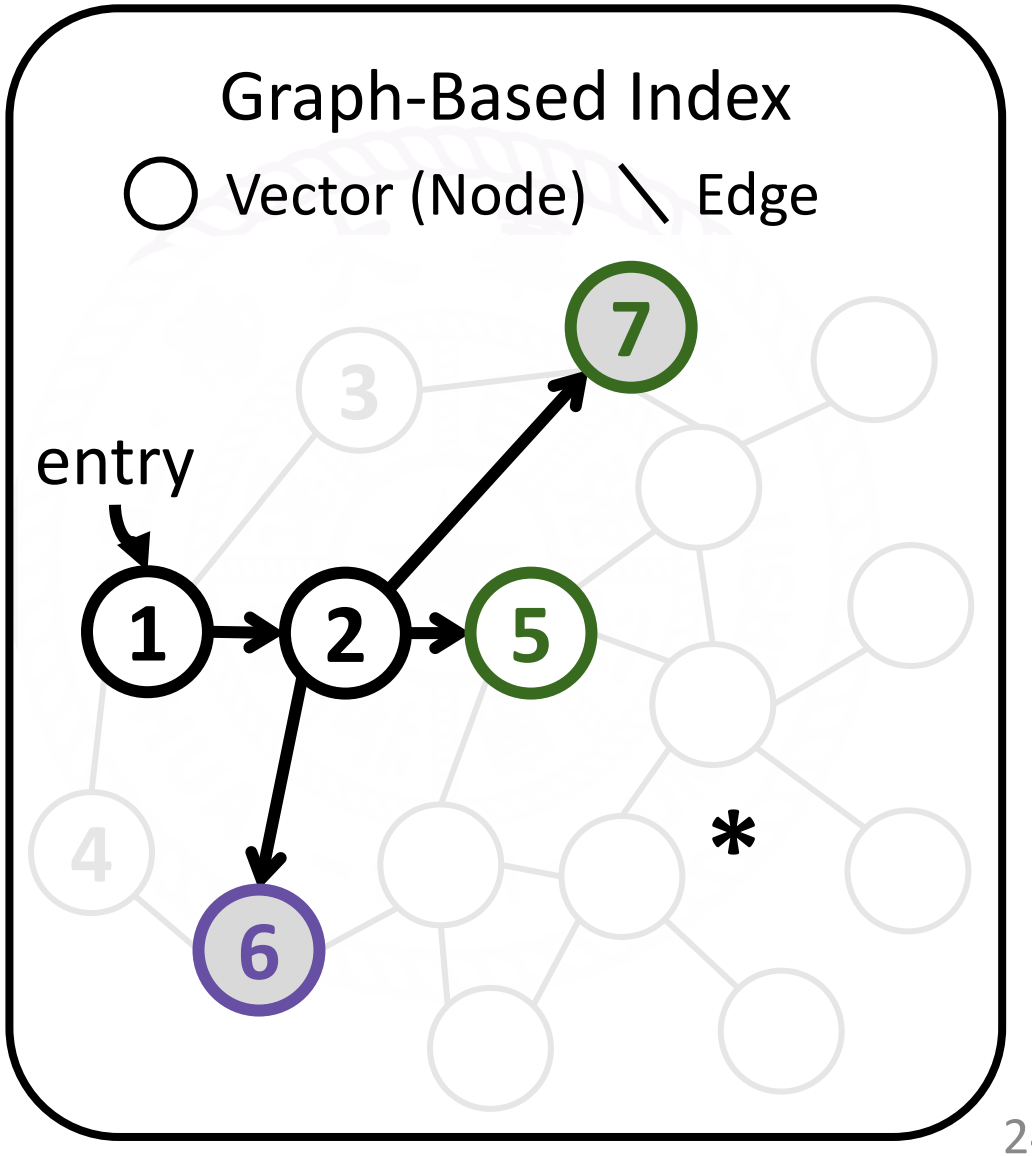
Pipeline not full, read.

Pipeline full, compute.

- Select the nearest unvisited vectors **5** and **6** in the *candidate pool*
- Async read **5** and **6**
- When one read (e.g., **6**) finishes, ***immediately read*** the nearest **7**.



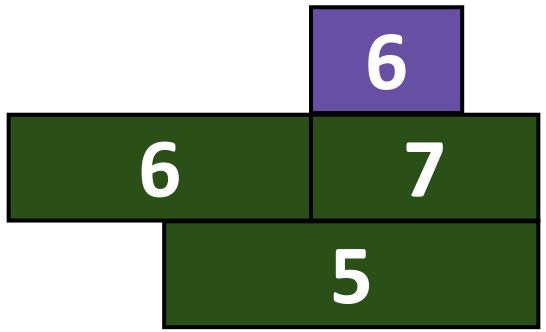
Candidate pool: { **5,** ~~2~~, **6, 7,** 3, 4, ~~1~~ }



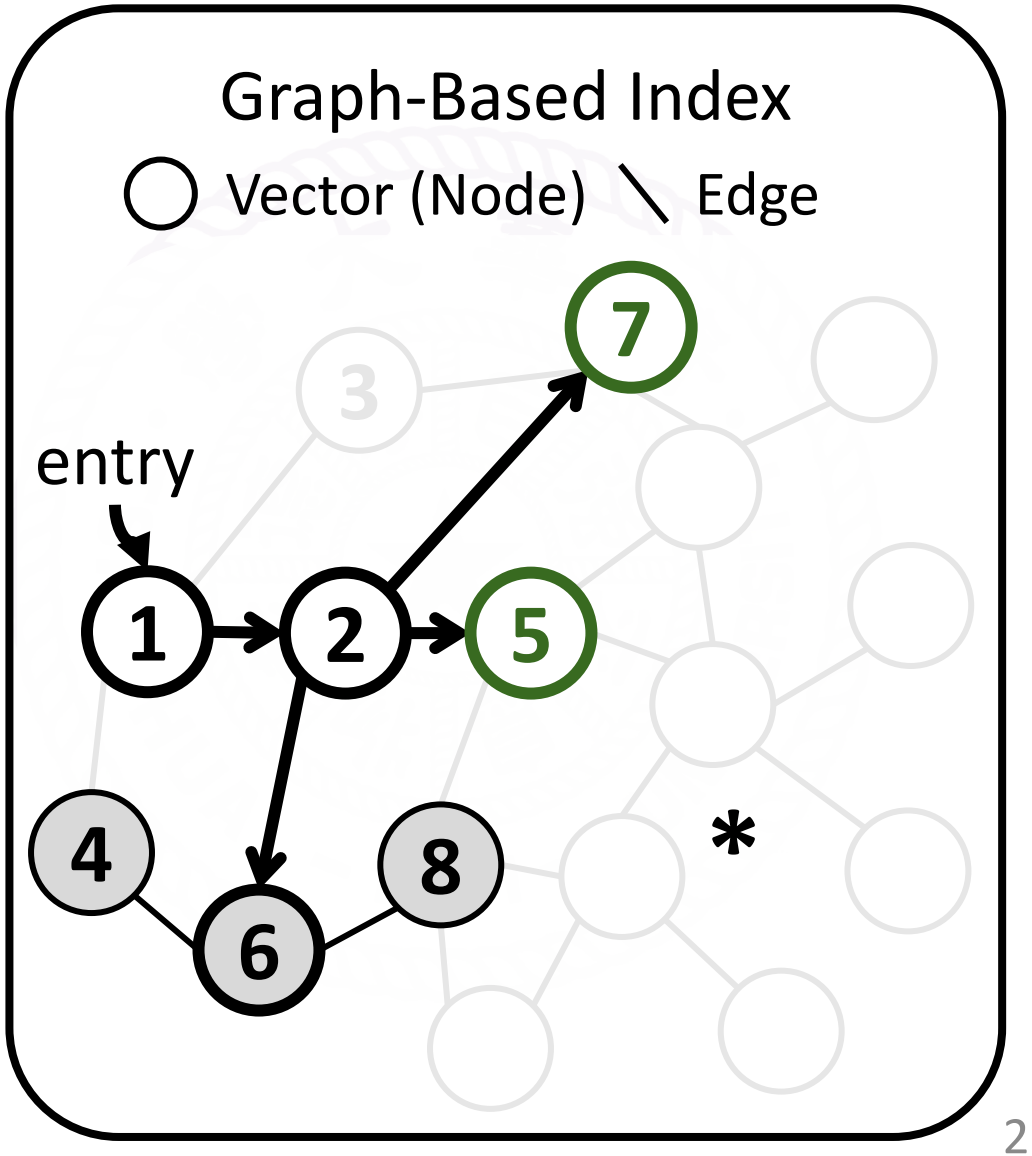
Put the Idea into Vector Search…

Pipeline not full, read.
Pipeline full, compute.

- Compare **6**'s neighbors' distances to the target vector
- Add **6**'s neighbors to the candidate pool, sort them, mark **6** as visited



Candidate pool: { **5,** 8, ~~2~~, ~~6~~, **7,** 3, 4 }

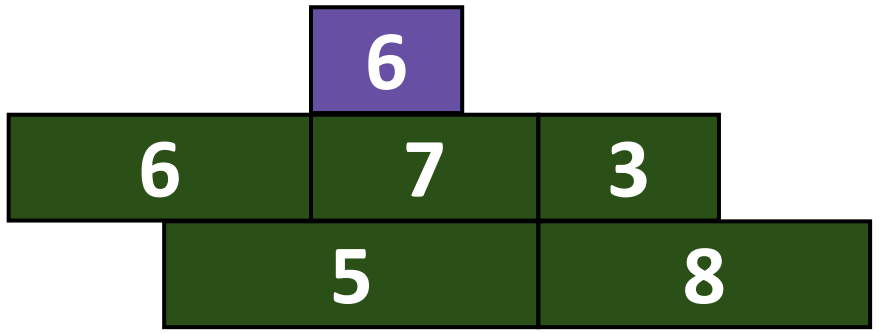


Put the Idea into Vector Search…

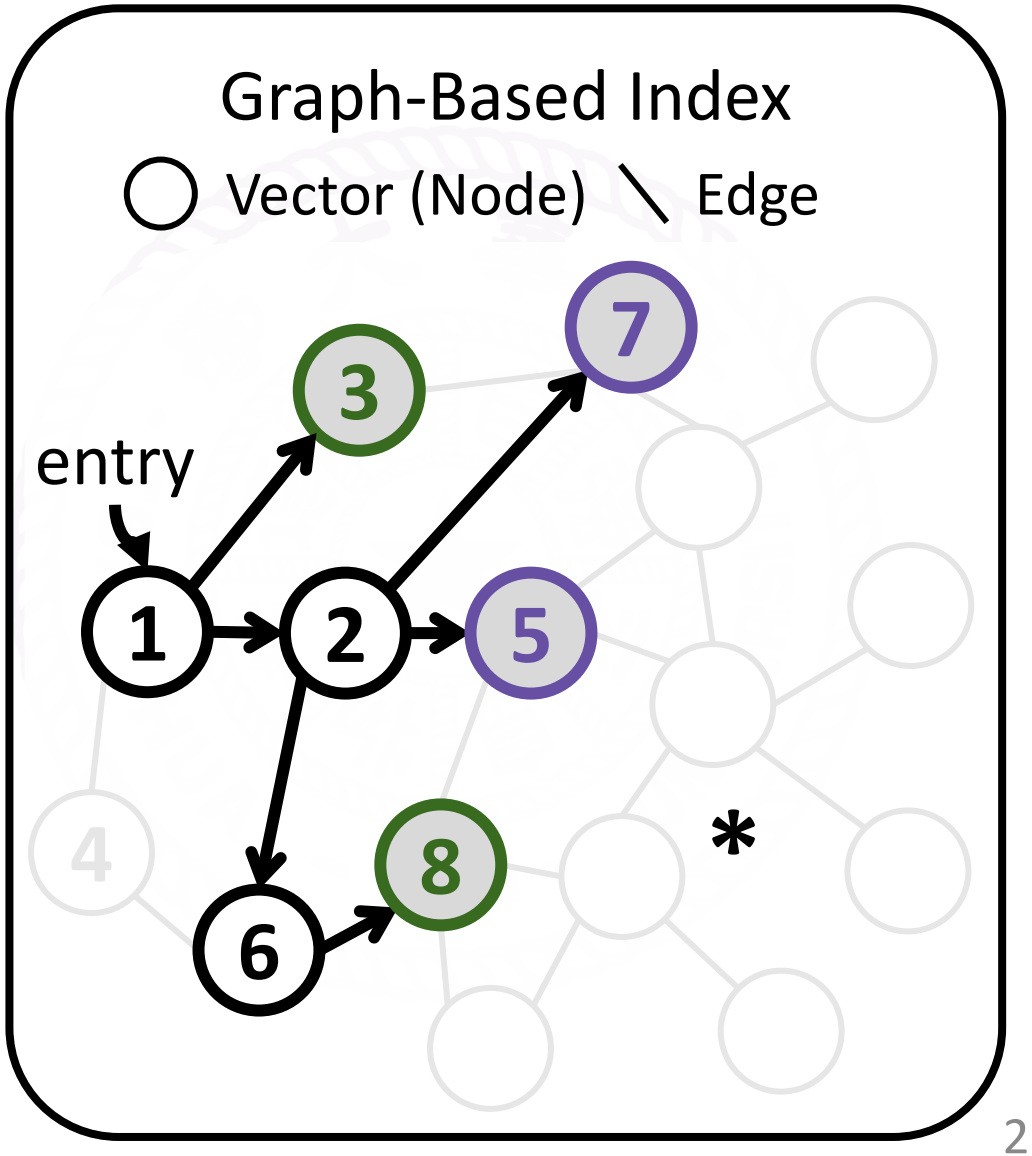
Pipeline not full, read.

Pipeline full, compute.

- Read 5 finishes, read 8
- Read 7 finishes, read 3



Candidate pool: { 5, 8, ~~2~~, ~~6~~, 7, 3, 4 }

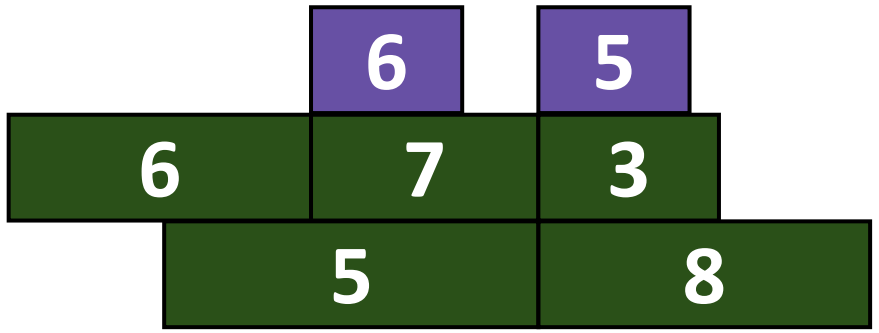


Put the Idea into Vector Search…

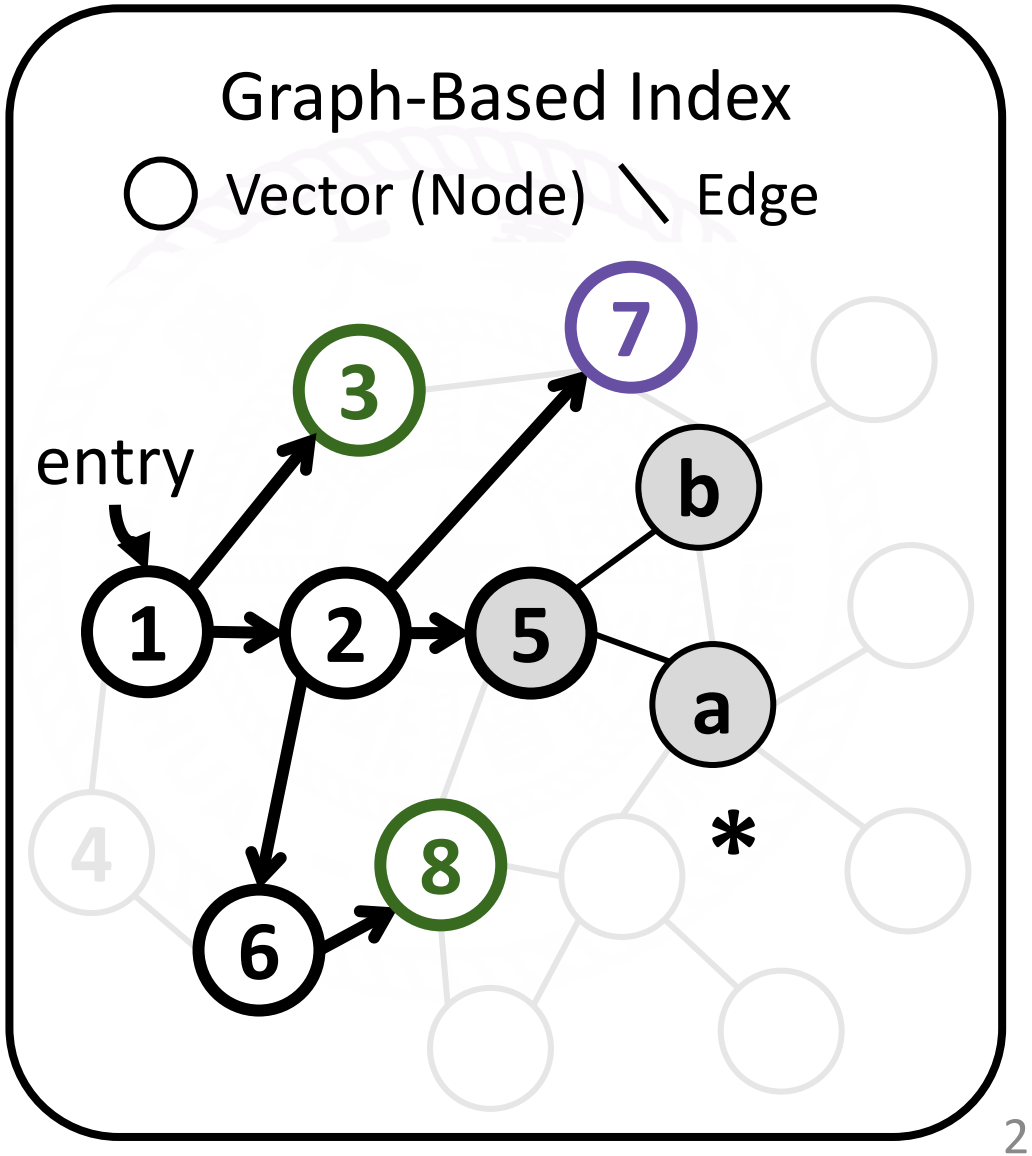
Pipeline not full, read.

Pipeline full, compute.

- Read **5** finishes, read **8**
- Read **7** finishes, read **3**
- I/O pipeline full, compute **5**



Candidate pool: { **a,** ~~5~~, **8,** b, ~~2~~, ~~6~~, **7** }

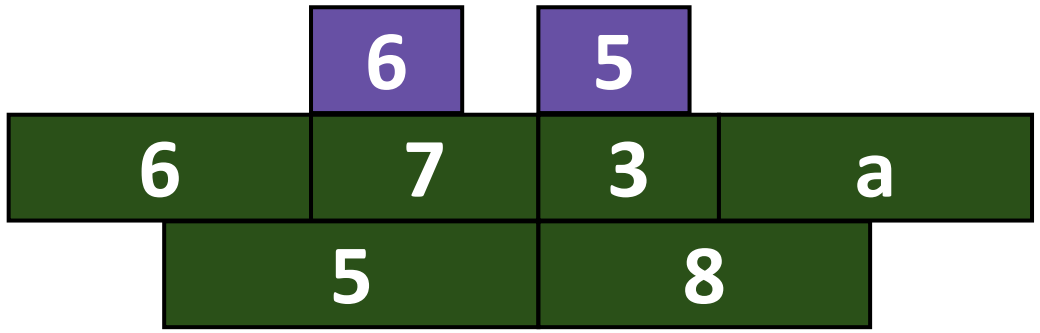


Put the Idea into Vector Search…

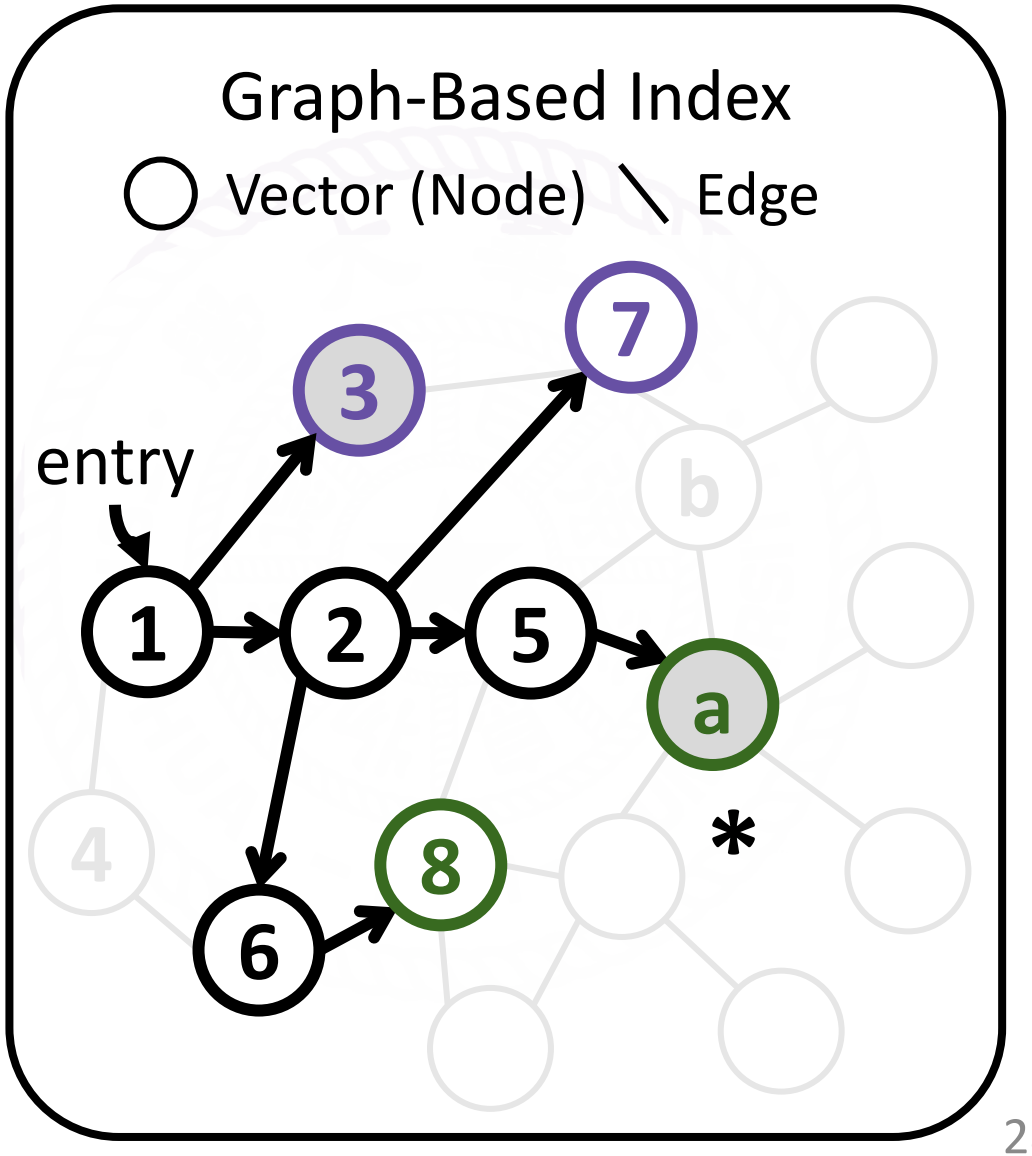
Pipeline not full, read.

Pipeline full, compute.

- Read **5** finishes, read **8**
- Read **7** finishes, read **3**
- I/O pipeline full, compute **5**
- Read **3** finishes, read **a**



Candidate pool: { **a,** ~~5~~, **8,** b, ~~2~~, ~~6~~, **7** }

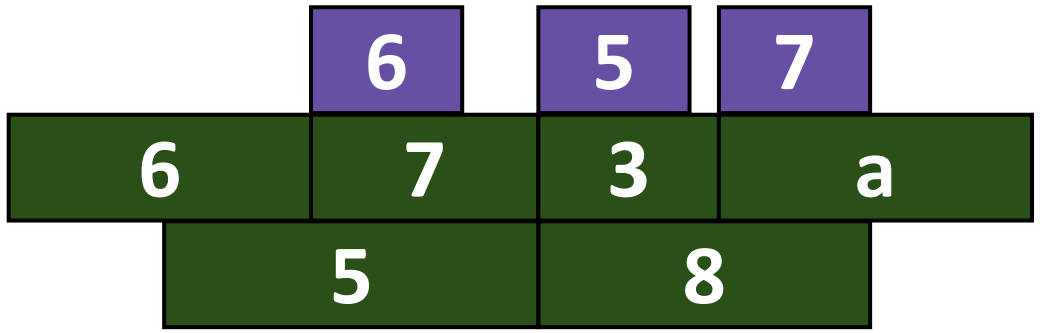


Put the Idea into Vector Search…

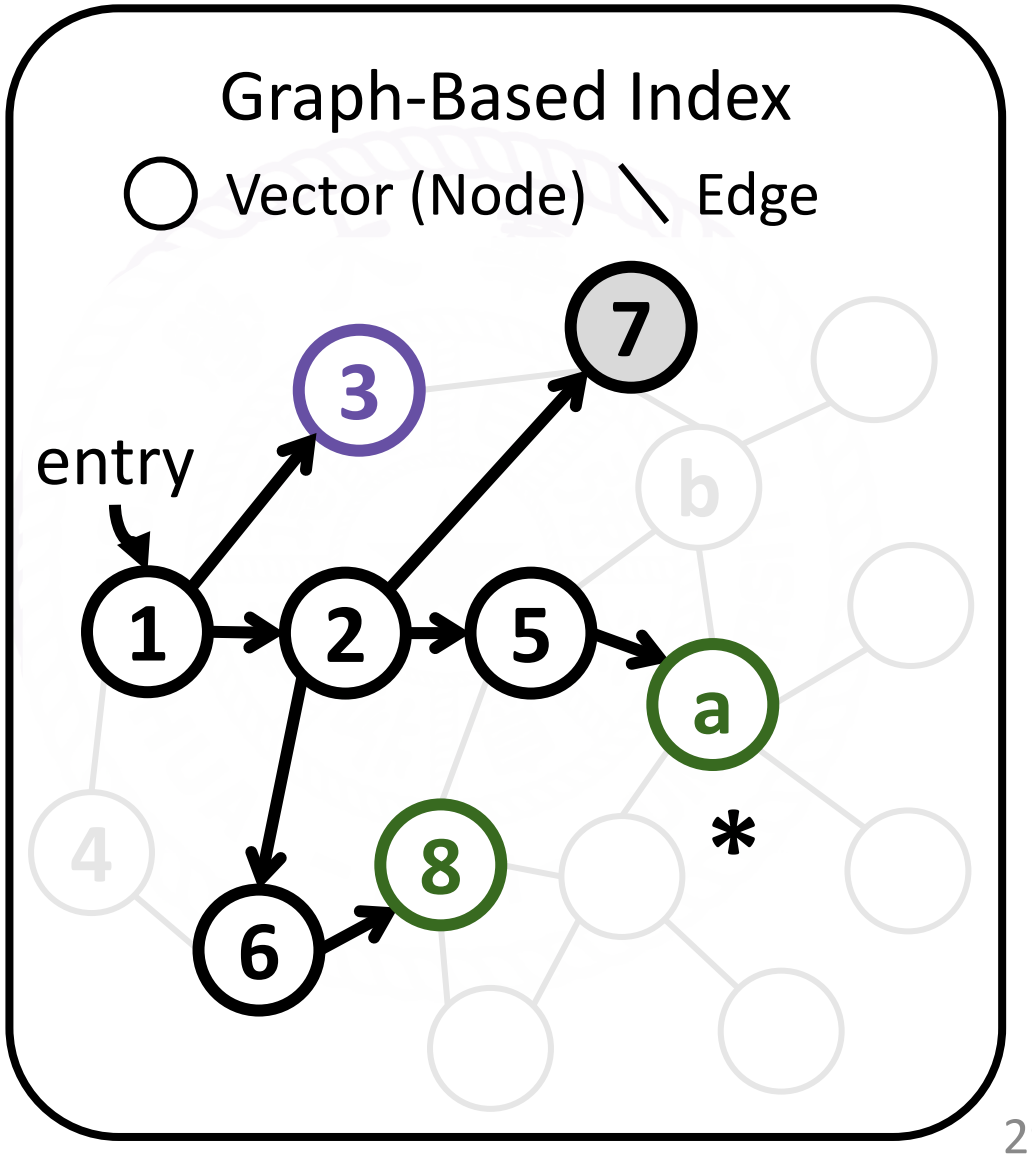
Pipeline not full, read.

Pipeline full, compute.

- I/O pipeline full, compute **5**
- Read **3** finishes, read **a**
- I/O pipeline full, compute **7**



Candidate pool: { **a,** ~~5~~, **8,** b, ~~2~~, ~~6~~, ~~7~~ }

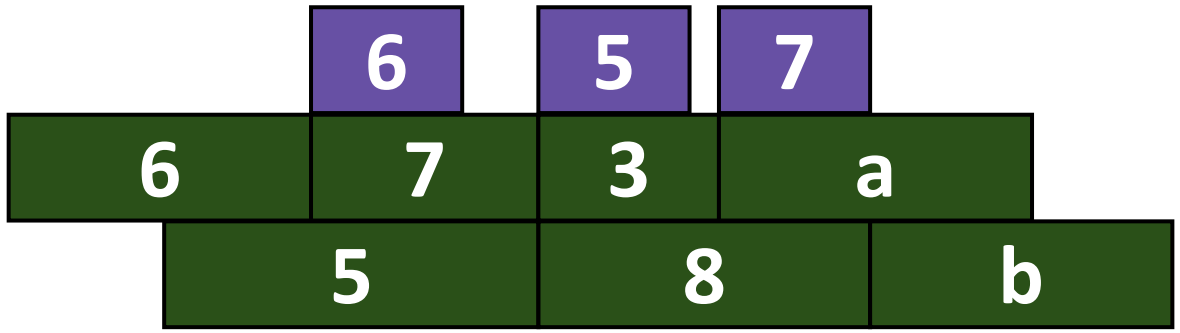


Put the Idea into Vector Search…

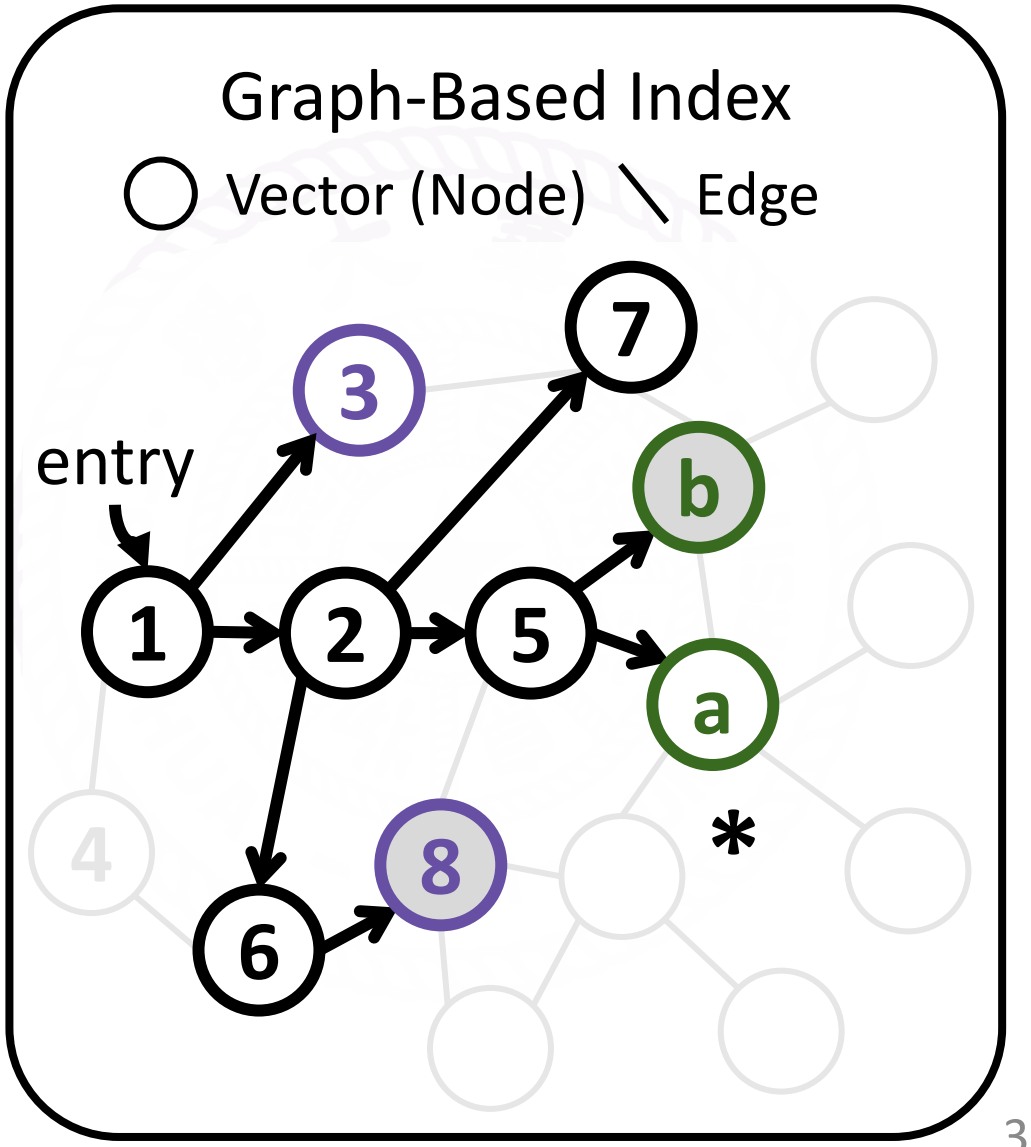
Pipeline not full, read.

Pipeline full, compute.

- I/O pipeline full, compute **5**
- Read **3** finishes, read **a**
- I/O pipeline full, compute **7**
- Read **8** finishes, read **b**



Candidate pool: { **a,** ~~5~~, **8, b,** ~~2~~, ~~6~~, ~~7~~ }

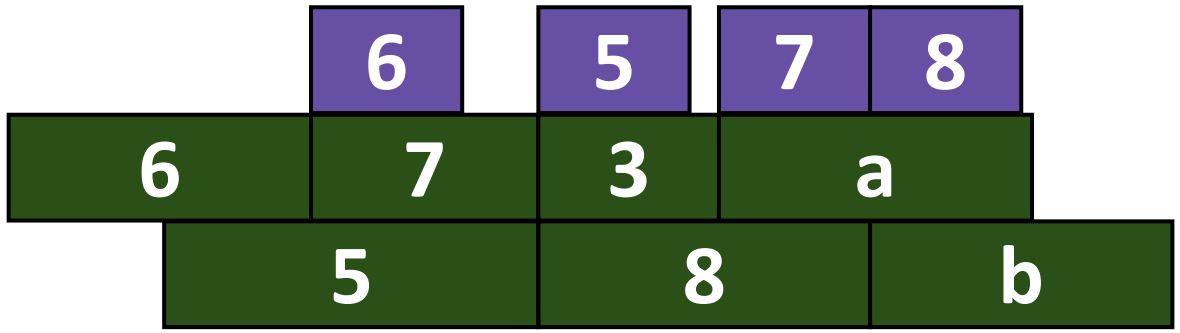


Put the Idea into Vector Search…

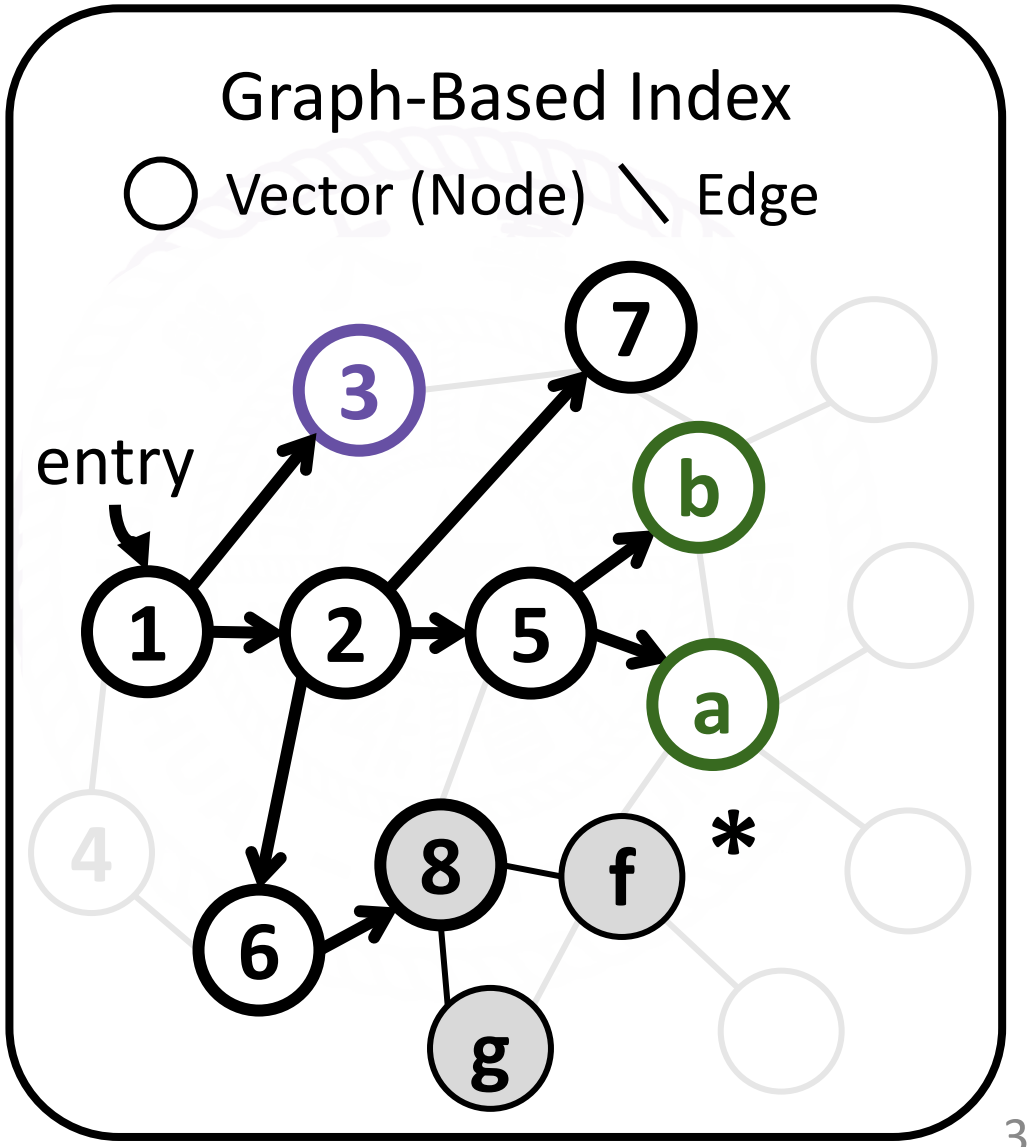
Pipeline not full, read.

Pipeline full, compute.

- I/O pipeline full, compute **7**
- Read **8** finishes, read **b**
- I/O pipeline full, compute **8**



Candidate pool: { f, **a,** ~~5~~, ~~8~~, **b,** ~~2~~, g, ~~6~~ }

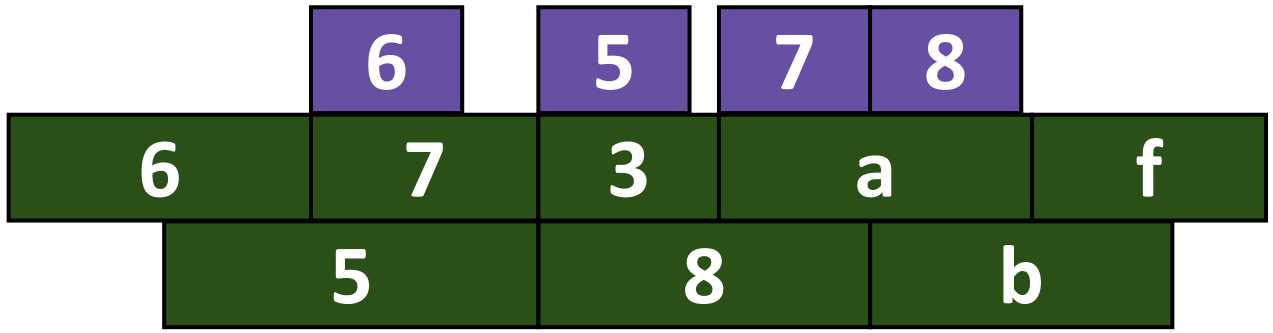


Put the Idea into Vector Search…

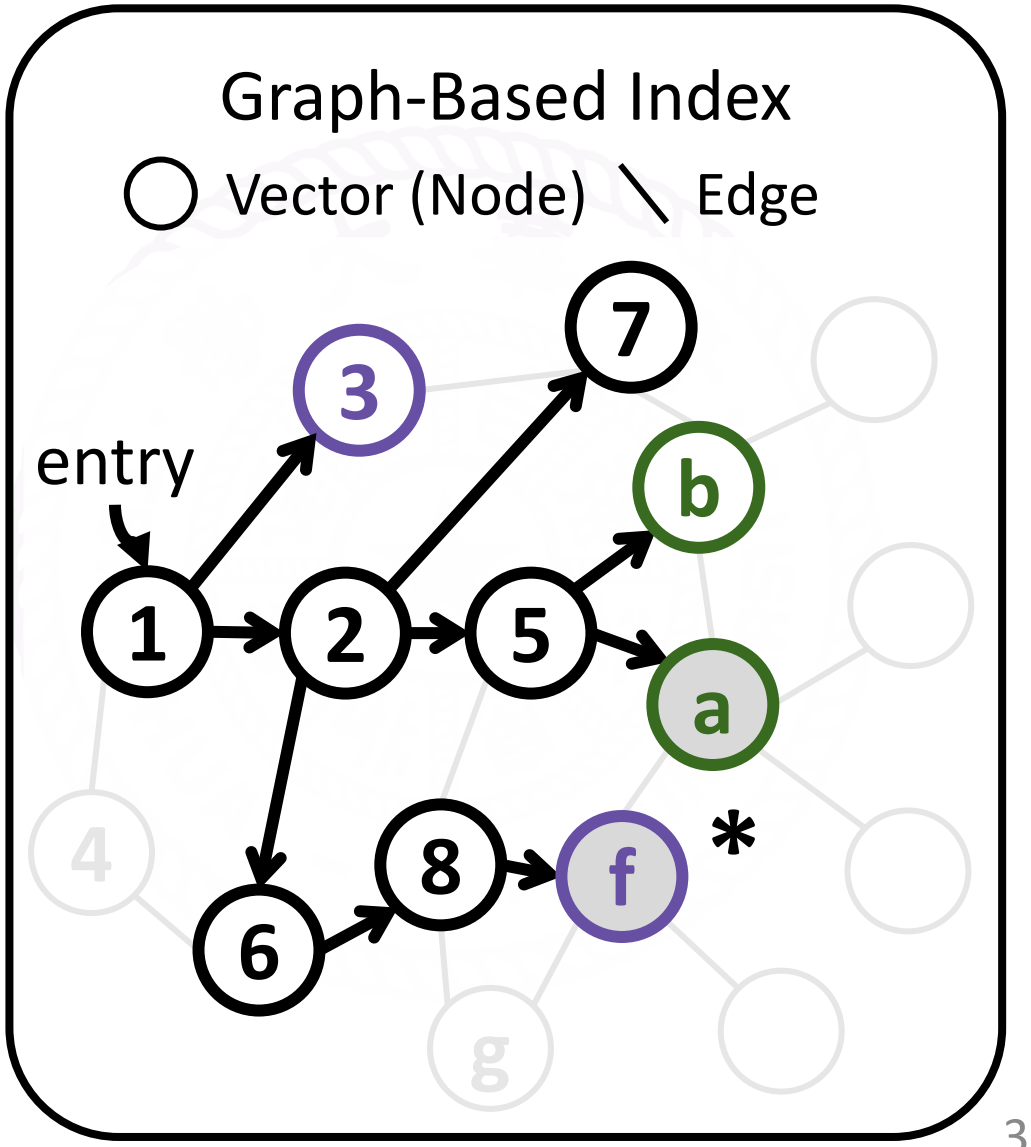
Pipeline not full, read.

Pipeline full, compute.

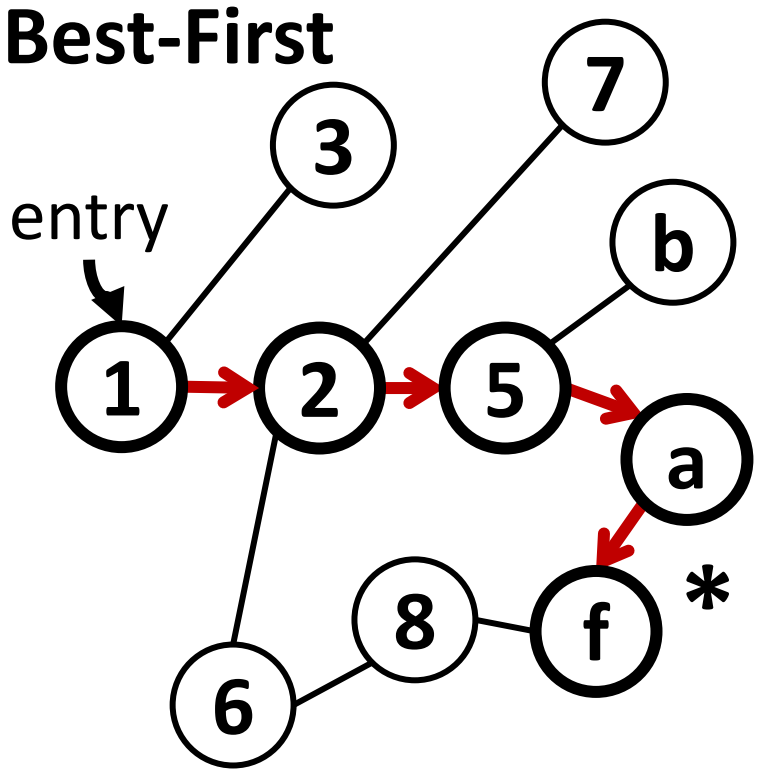
- I/O pipeline full, compute **7**
- Read **8** finishes, read **b**
- I/O pipeline full, compute **8**
- Read **a** finishes, read **f**

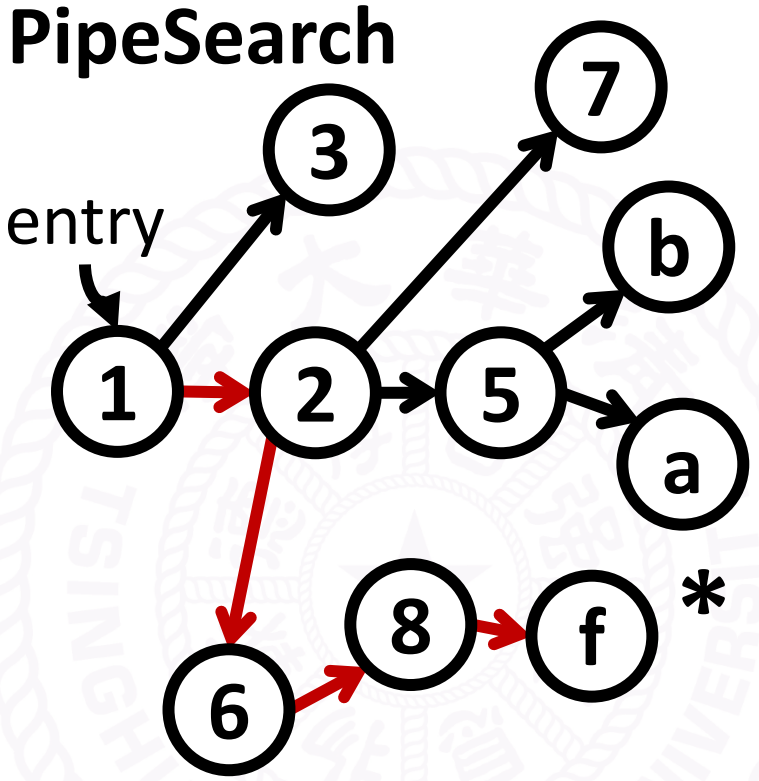


Candidate pool: { **f, a,** ~~5~~, ~~8~~, **b,** ~~2~~, g, ~~6~~ }



PipeSearch is fast, because…

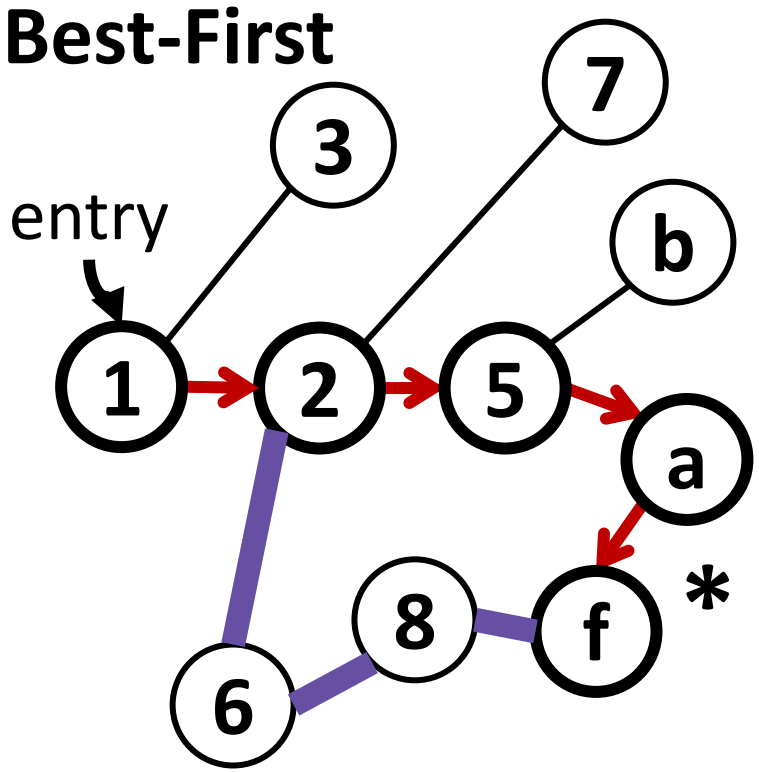


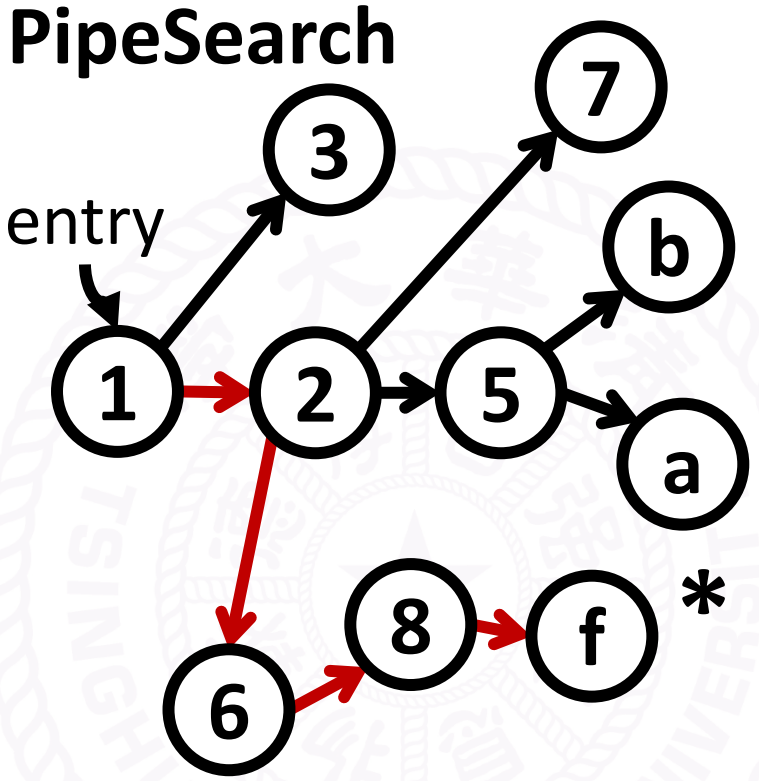


The same critical path length, but

More vectors, full I/O pipeline, due to…

PipeSearch is fast, because…





The same critical path length, but

More vectors, full I/O pipeline, due to…

Exploiting ***other search paths***

PipeSearch is fast, but…

How wide the I/O pipeline should be?

Wide pipeline

Parallel reads, **lower latency**

Speculative reads, **lower throughput**

Narrow pipeline

Best-first reads, **higher throughput**

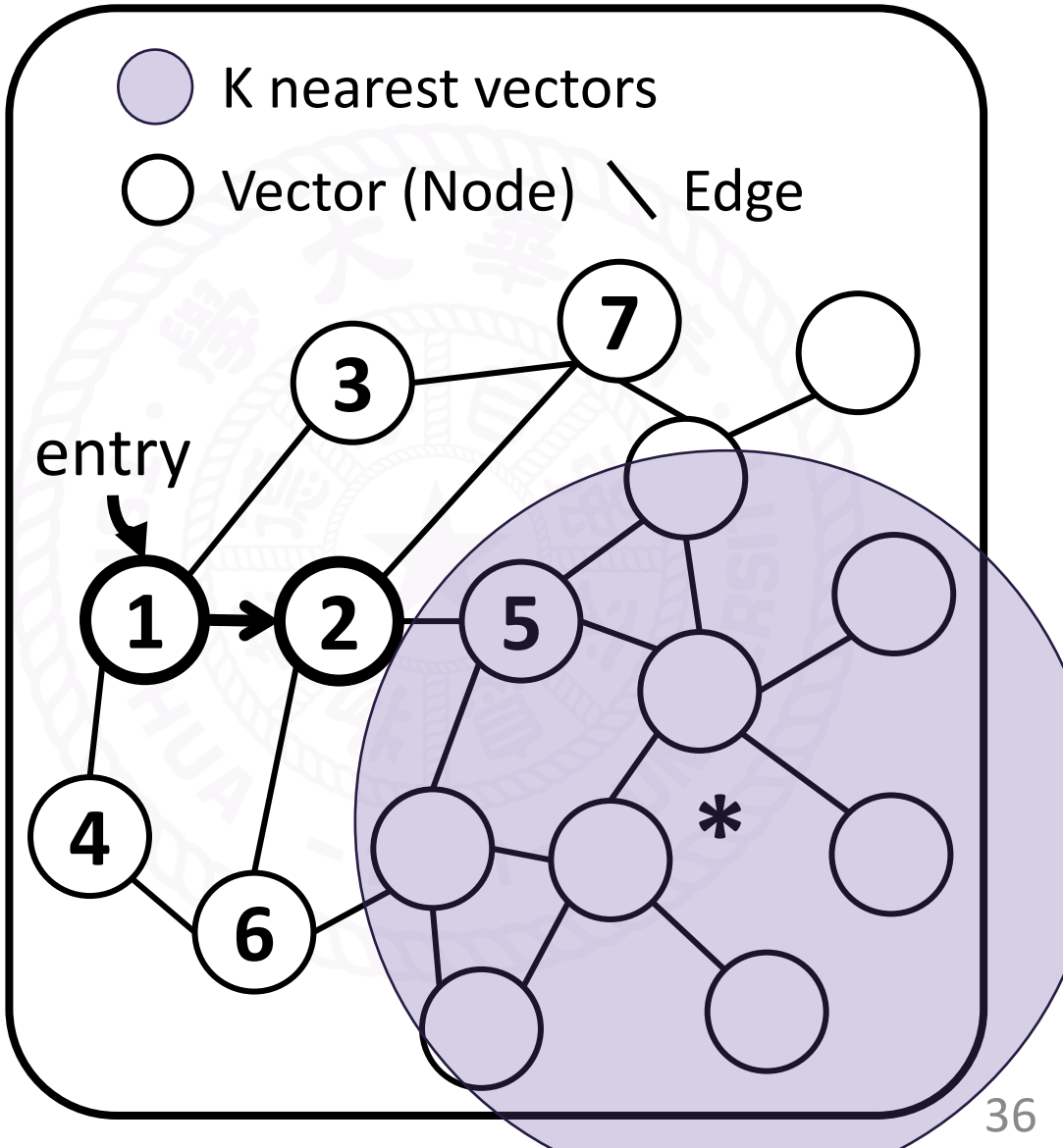
Sequential reads, **higher latency**

Increasingly Speculative as the Search Progresses

The search is divided into two phases

Approach phase: approaching the target vector ∗ from the entry

- ***Less*** useful but unvisited vectors
- Speculative leads to ***wastes***



Increasingly Speculative as the Search Progresses

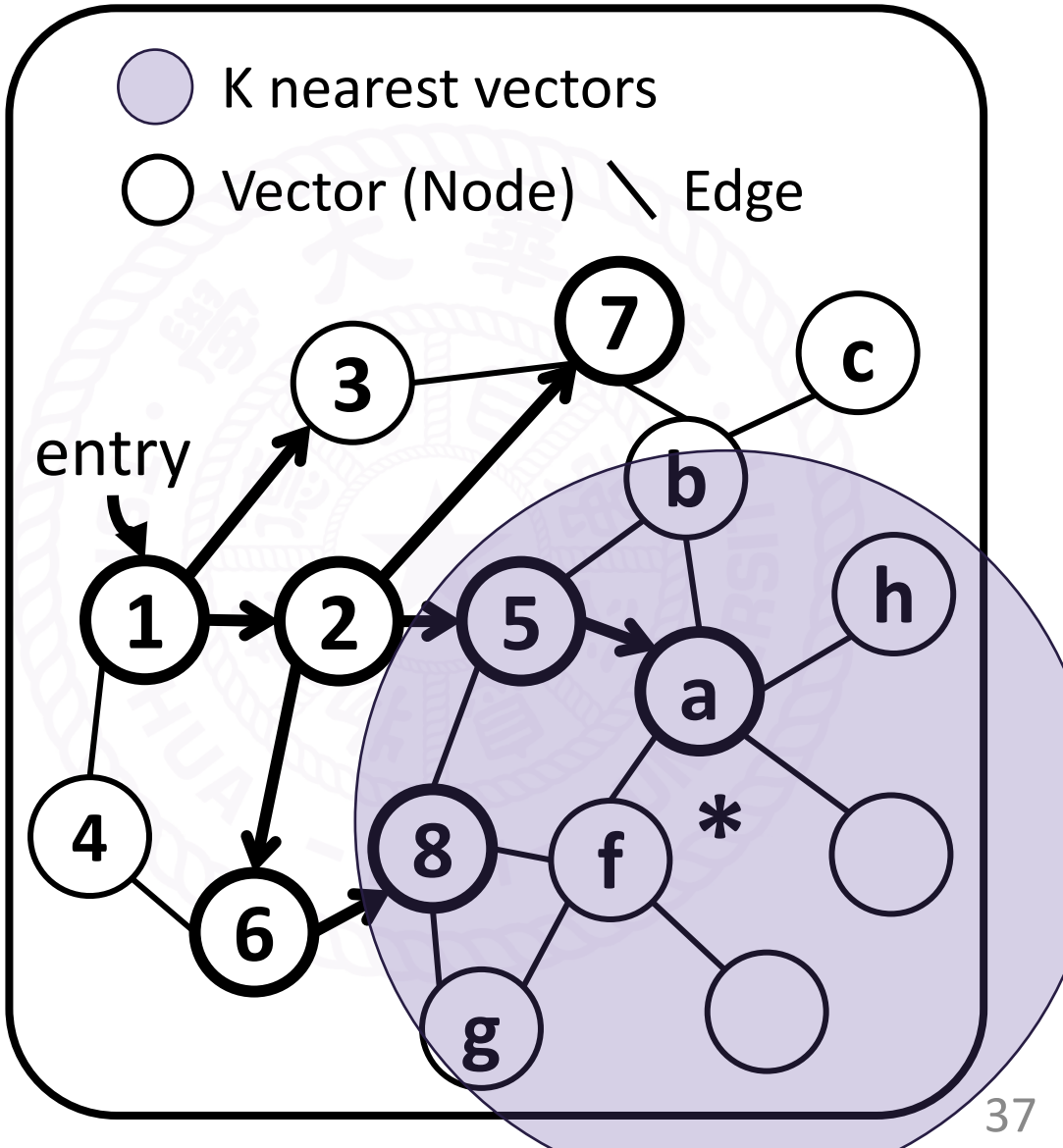
The search is divided into two phases

Approach phase: approaching the target vector ∗ from the entry

- ***Less*** useful but unvisited vectors
- Speculative leads to ***wastes***

Converge phase: recalling the nearest vectors of the target vector

- ***More*** useful but unvisited vectors
- Speculative leads to ***acceleration***



How to Adjust I/O Pipeline Width?

TLDR: ***Gradually increase*** as the search progresses

When to increase?

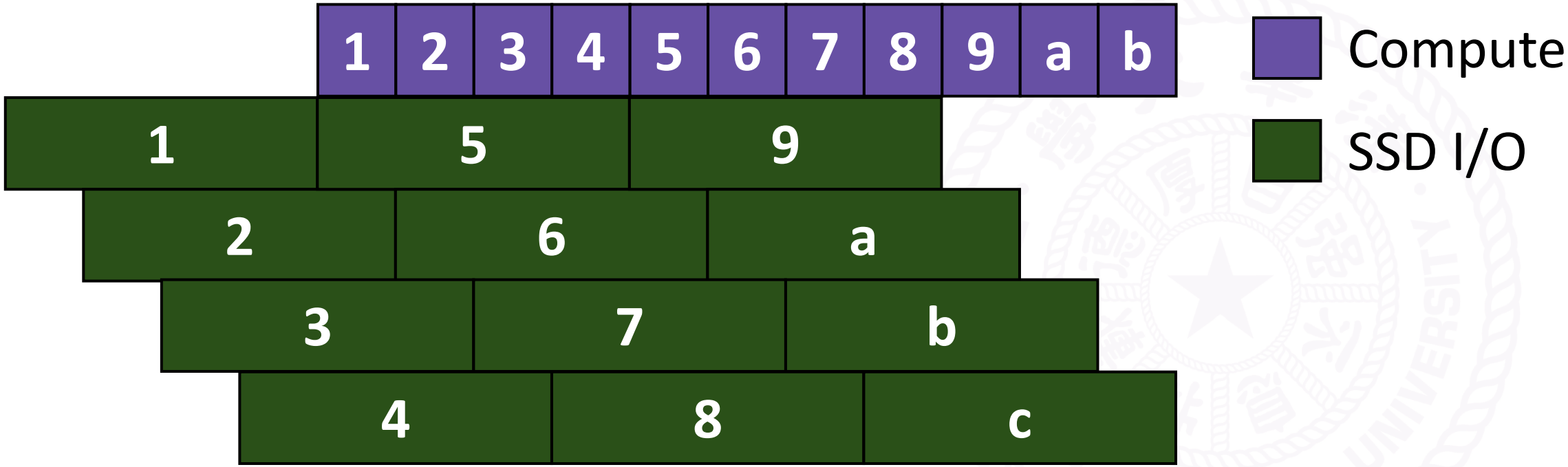
Metric: % fetched vectors still in the candidate pool

- Small: Candidate pool frequently changes due to ***approaching***.
- Large: Candidate pool ***converges*** to the nearest vectors.

If Metric >= threshold (e.g., 0.9), width += 1

When Multiple I/O Finish Simultaneously…

PipeSearch ideal case (I/O pipeline width = 4)

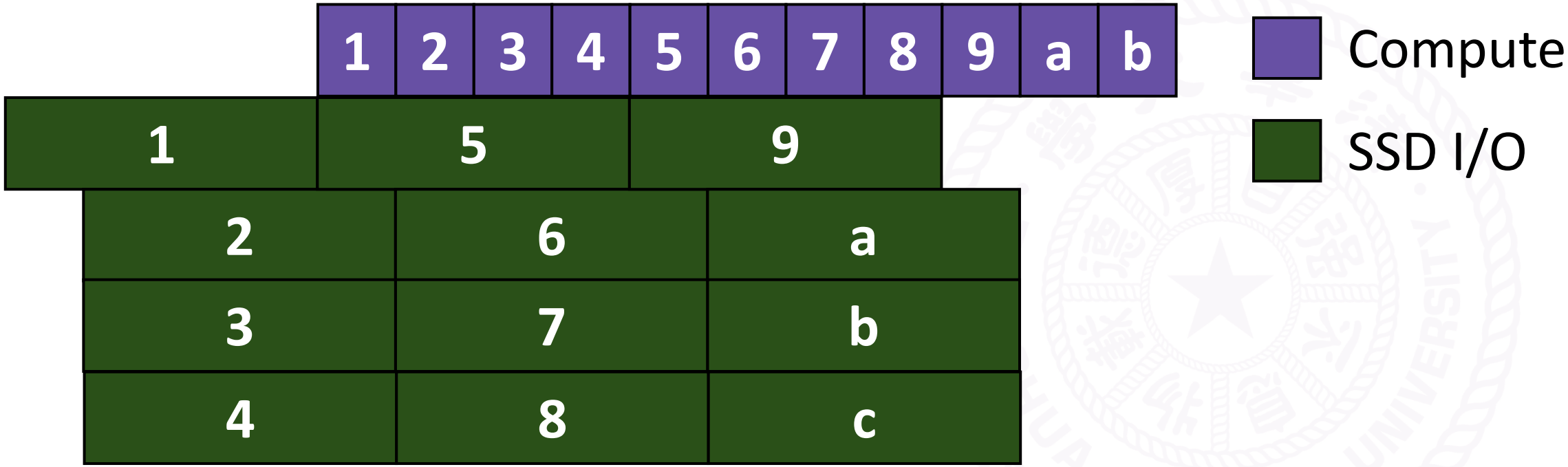


Each I/O misses at most **4 vectors' neighbors**

(e.g., 8 misses 4 5 6 7)

When Multiple I/O Finish Simultaneously…

PipeSearch bad case (I/O pipeline width = 4)

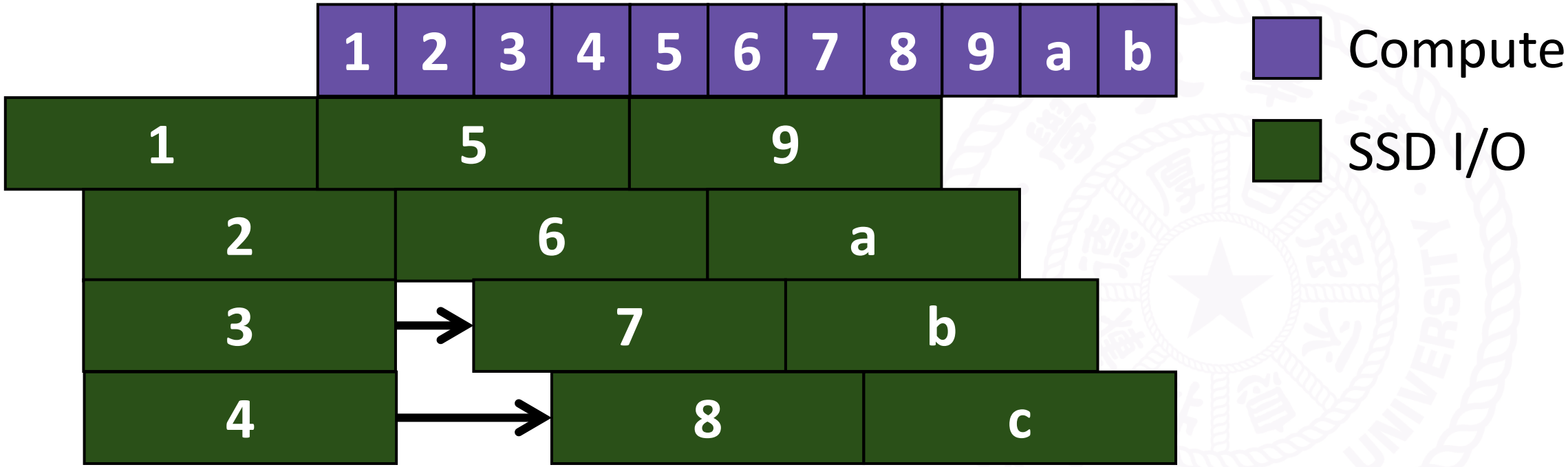


Each I/O misses **more vectors' neighbors!**

(e.g., 8 misses 2 3 4 5 6 7)

When Multiple I/O Finish Simultaneously…

Delay I/O to ensure an upper bound of missed neighbors



Each I/O misses at most **4 vectors' neighbors**

(e.g., 8 misses 4 5 6 7)

Put Them Altogether: PipeANN

A large-scale vector search system with low latency

- Key enabler for low latency: ***PipeSearch*** algorithm
- Two techniques for throughput:
  - ***Increase pipeline width*** as the search progresses
  - ***Delay I/O*** for an upper bound of missed neighbors

Check our paper for more details!

Evaluation Setup

Hardware configuration

CPU	Intel Xeon Gold 6330, 2.0GHz
DRAM	512GB DDR4 (32GB * 16)
SSD	3.84TB PCIe 4.0 (4KB Read: 1.5M IOPS)

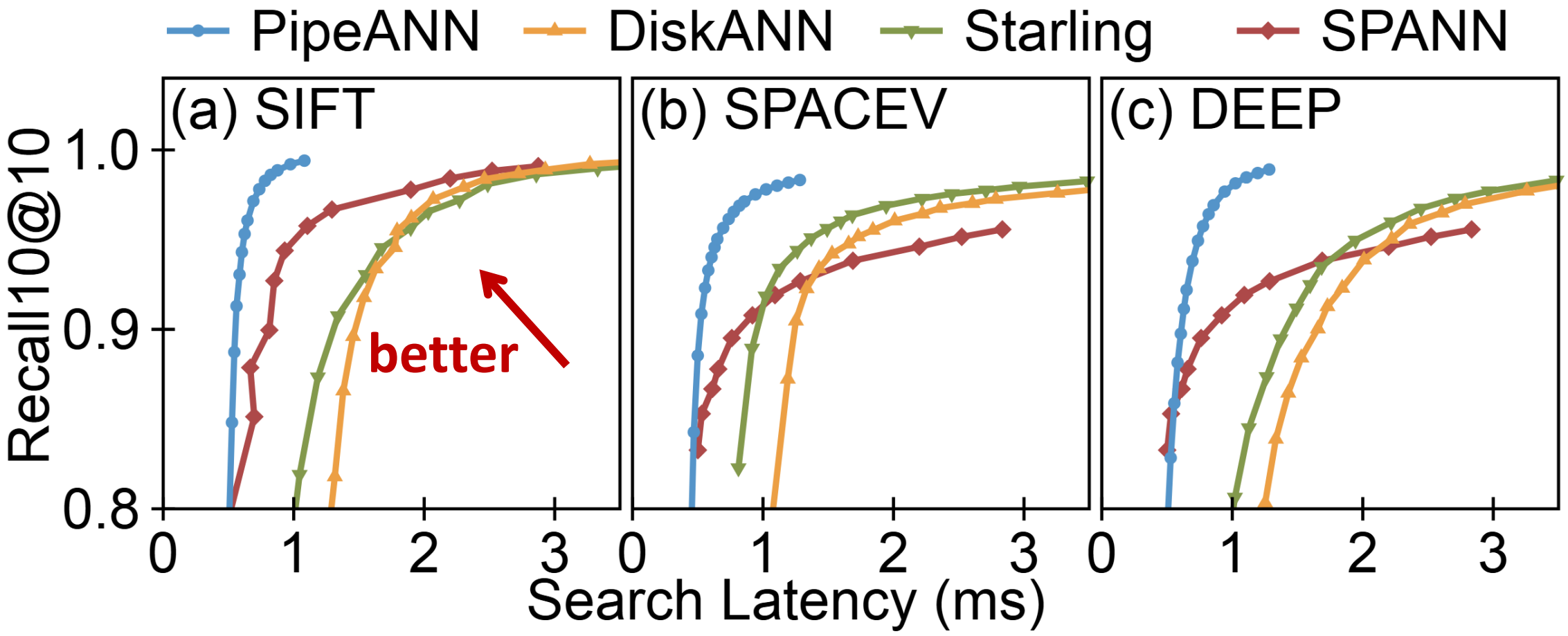
Compared with on-SSD vector search systems:

- **Graph-based:** DiskANN [NIPS'19], Starling [SIGMOD'24]
- **Cluster-based:** SPANN [NIPS'21]

Vector datasets:

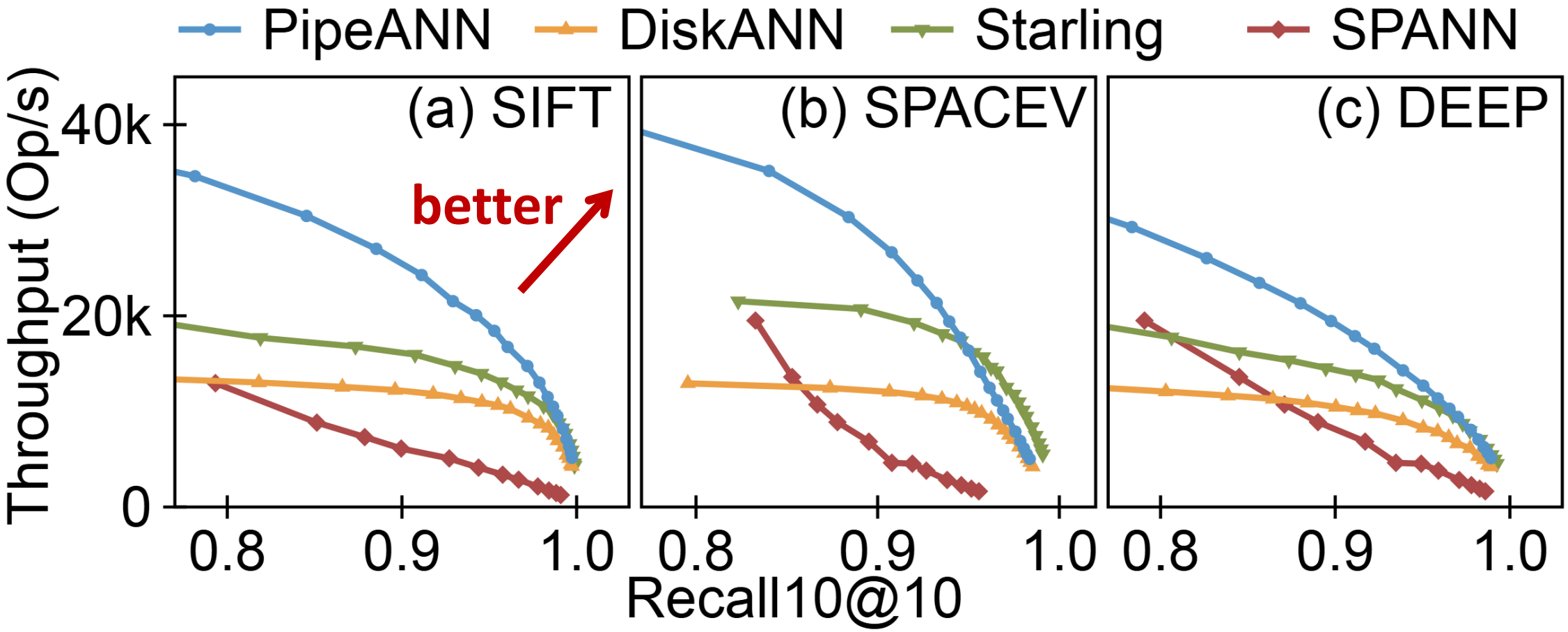
- **100 million vectors:** SIFT100M, SPACEV100M, DEEP100M
- **1 billion vectors:** SIFT1B, SPACEV1B

Overall Performance: Latency



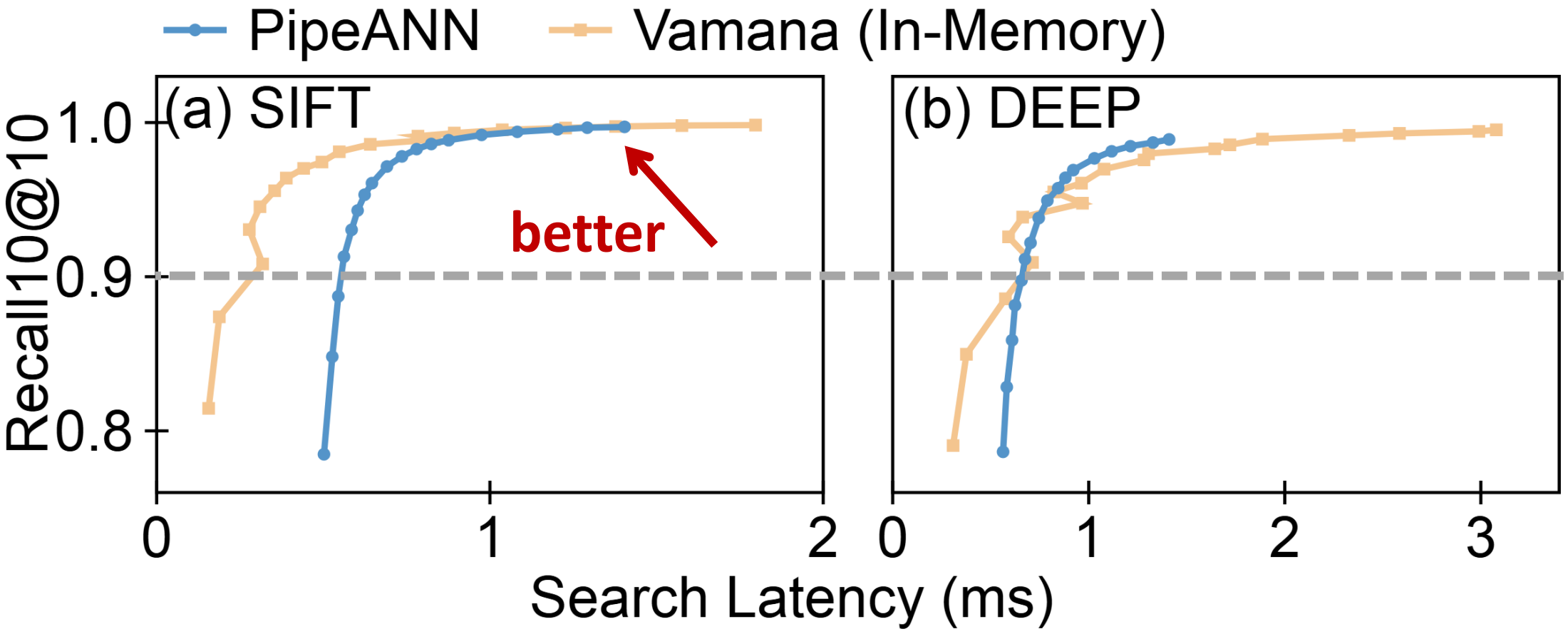
PipeANN shows lower latency compared to both graph- and cluster-based indexes

Overall Performance: Throughput



PipeANN shows similar throughput compared to graph- and cluster-based indexes

Compared with In-Memory Index



On-SSD PipeANN shows 2.12X/1.14X latency than in-memory Vamana for recall = 0.9

Thank you! Questions are welcome.

PipeANN: a **large-scale** vector search system with **low latency**

- Enabler for ultra low latency: **PipeSearch**
  - **Deviate from best-first search algorithm for pipelining**
- Two techniques for throughput:
  - **Increase pipeline width** as the search progresses
  - **Delay I/O** for an upper bound of missed neighbors
- **Significantly lower latency**, similar throughput

Artifact: github.com/thustorage/PipeANN

Contact: gh23@mails.tsinghua.edu.cn

image: Flaticon.com ☺