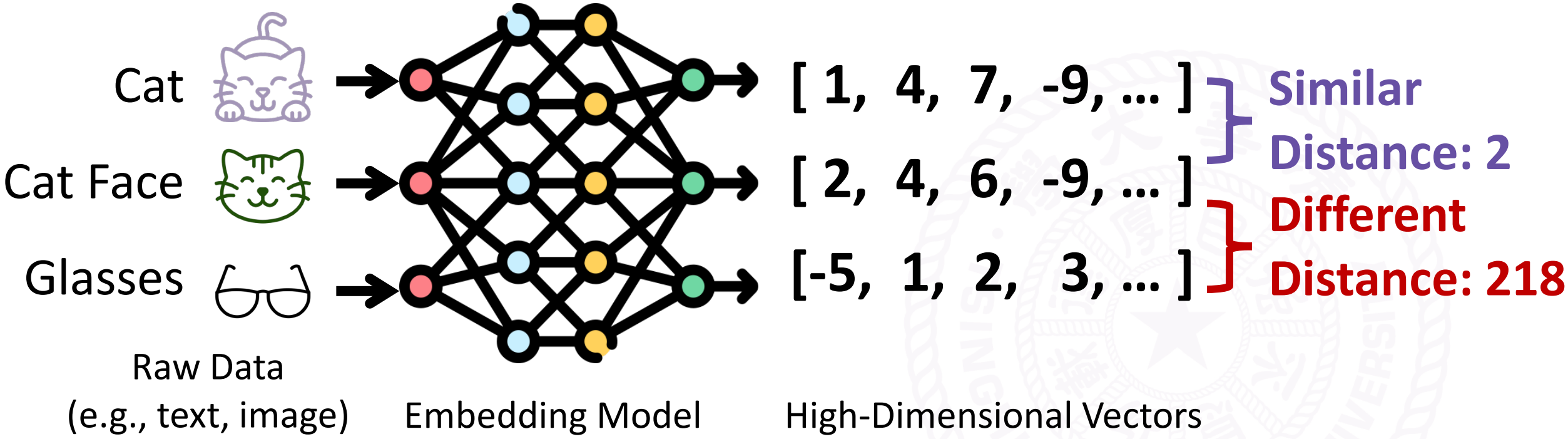
FAST'26

OdinANN: **Direct Insert** for Consistently Stable Performance in Billion-Scale Graph-Based Vector Search

Hao Guo, Youyou Lu

Tsinghua University

Vector: Powerful Data Representation



Vector distance reflects **data similarity**

Graph-Based Vector Index: Search

Vector dataset

[1, 4, 7, -9, …]

[2, 4, 6, -9, …]

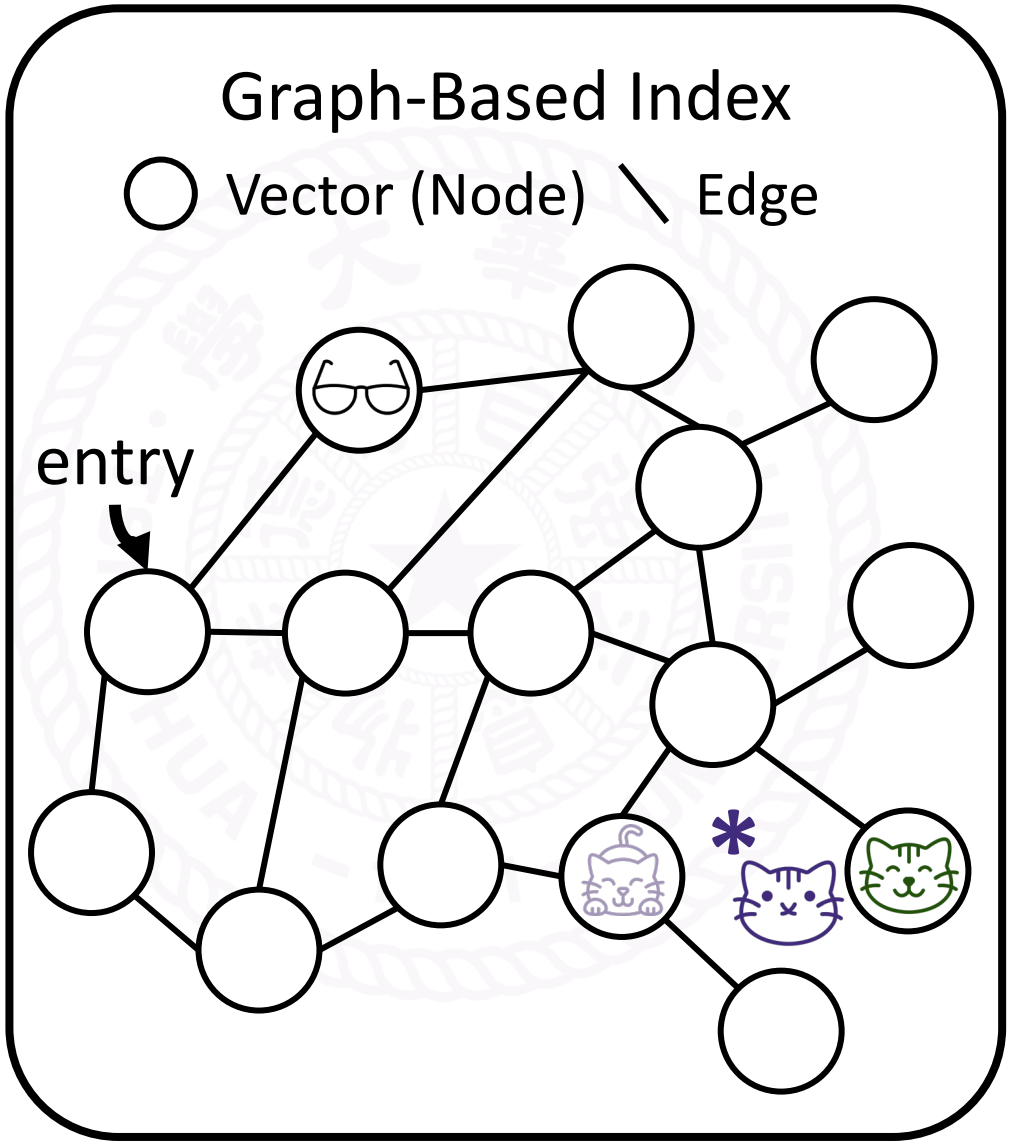
[-5, 1, 2, 3, …]

Target (Query)

[3, 5, 7, -8, …]

Build

Search



Graph-Based Vector Index: Search

Vector dataset

[1, 4, 7, -9, …]

[2, 4, 6, -9, …]

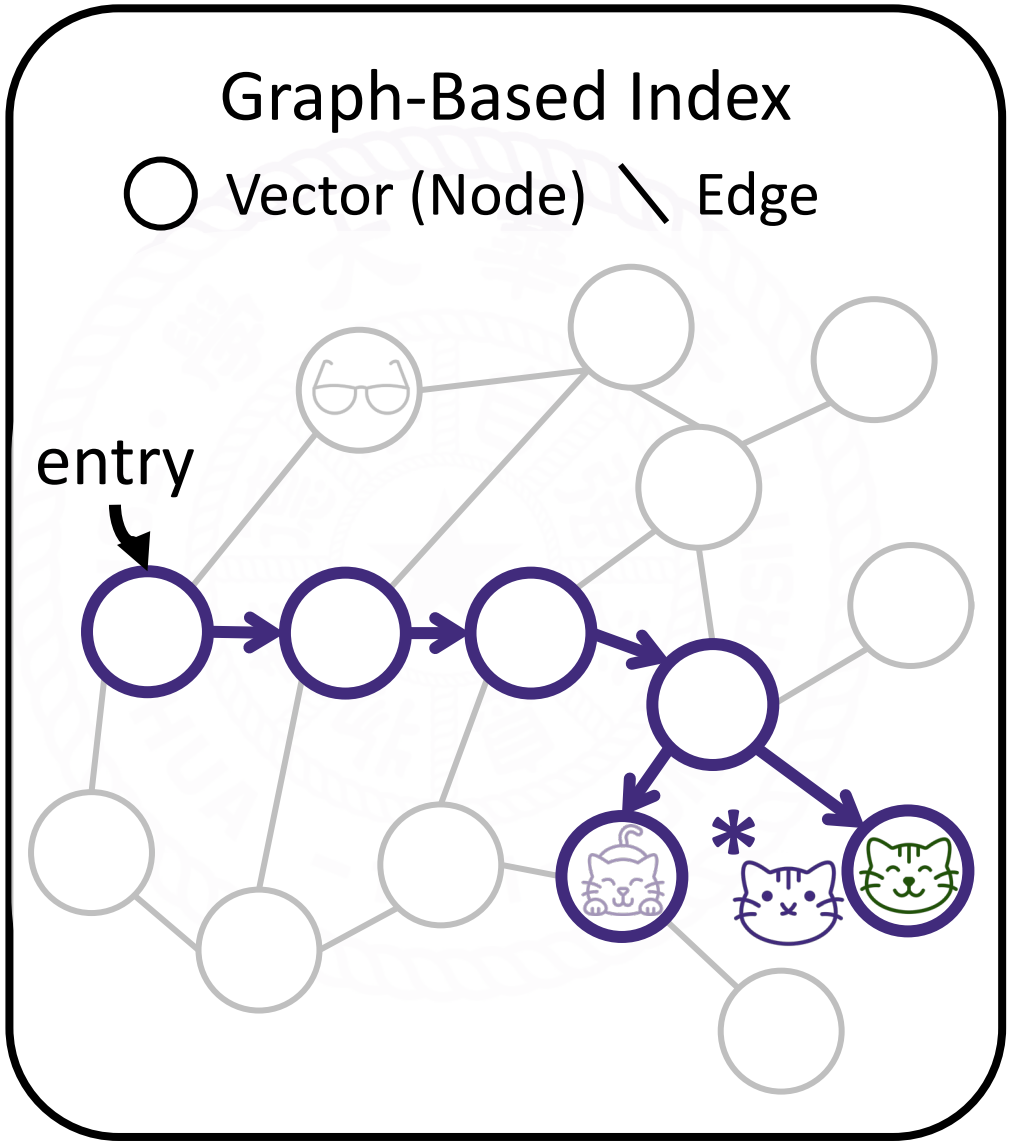
[-5, 1, 2, 3, …]

Target (Query)

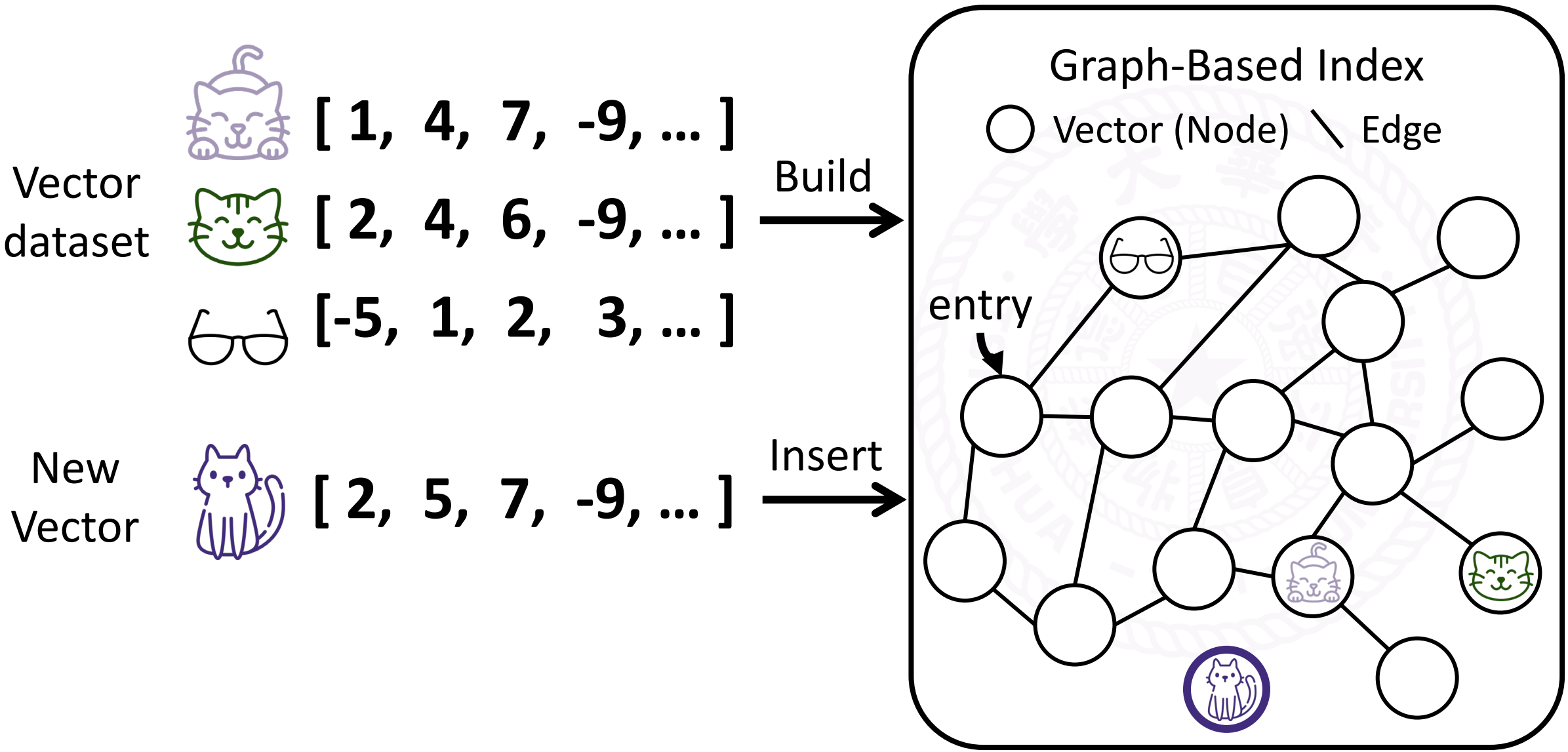
[3, 5, 7, -8, …]

Build

Search



Graph-Based Vector Index: Insert



Graph-Based Vector Index: Insert

Vector dataset


[1, 4, 7, -9, …]


[2, 4, 6, -9, …]

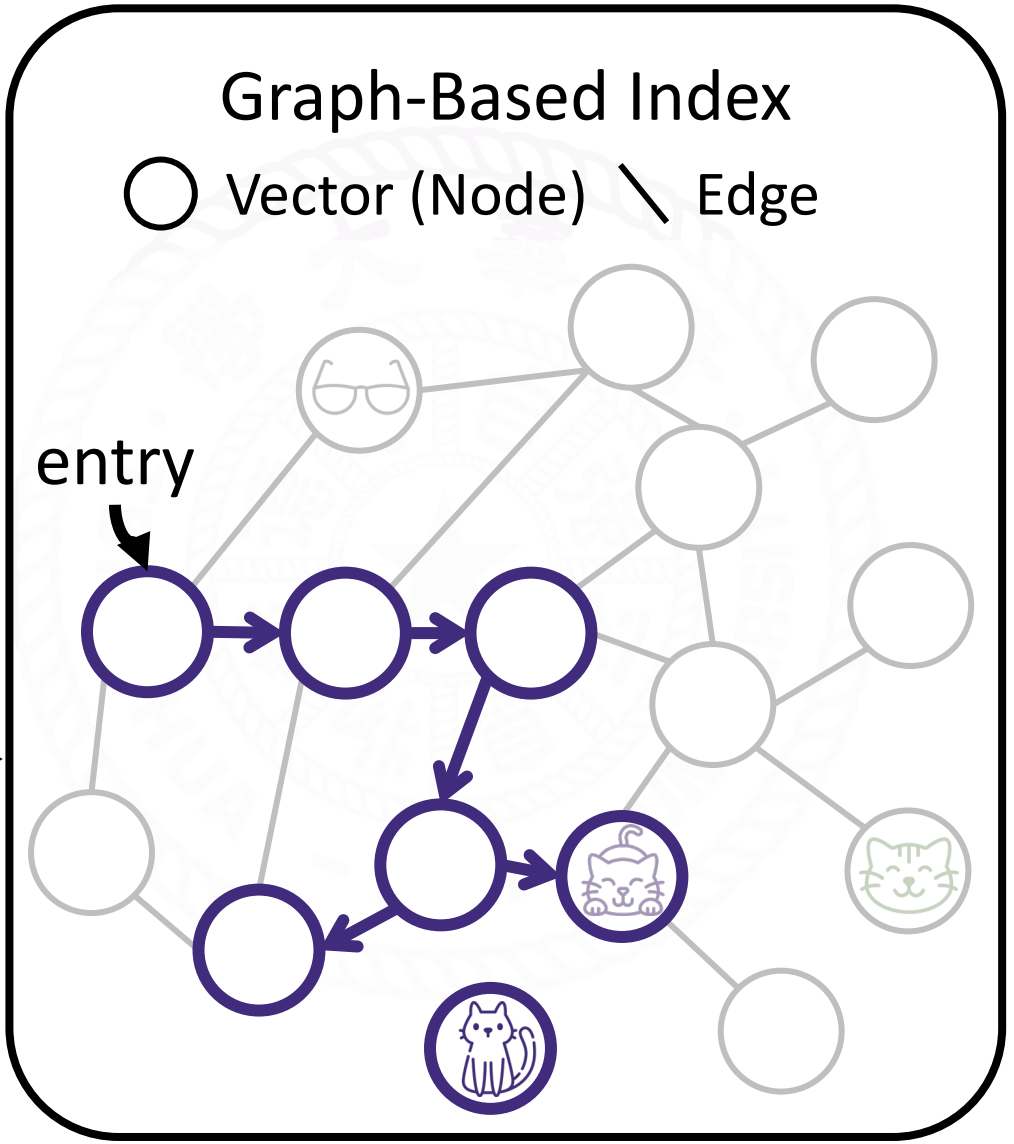

[-5, 1, 2, 3, …]

New Vector


[2, 5, 7, -9, …]

Build

Insert



1. Find candidate neighbors using search

Graph-Based Vector Index: Insert

Vector dataset


[1, 4, 7, -9, …]


[2, 4, 6, -9, …]

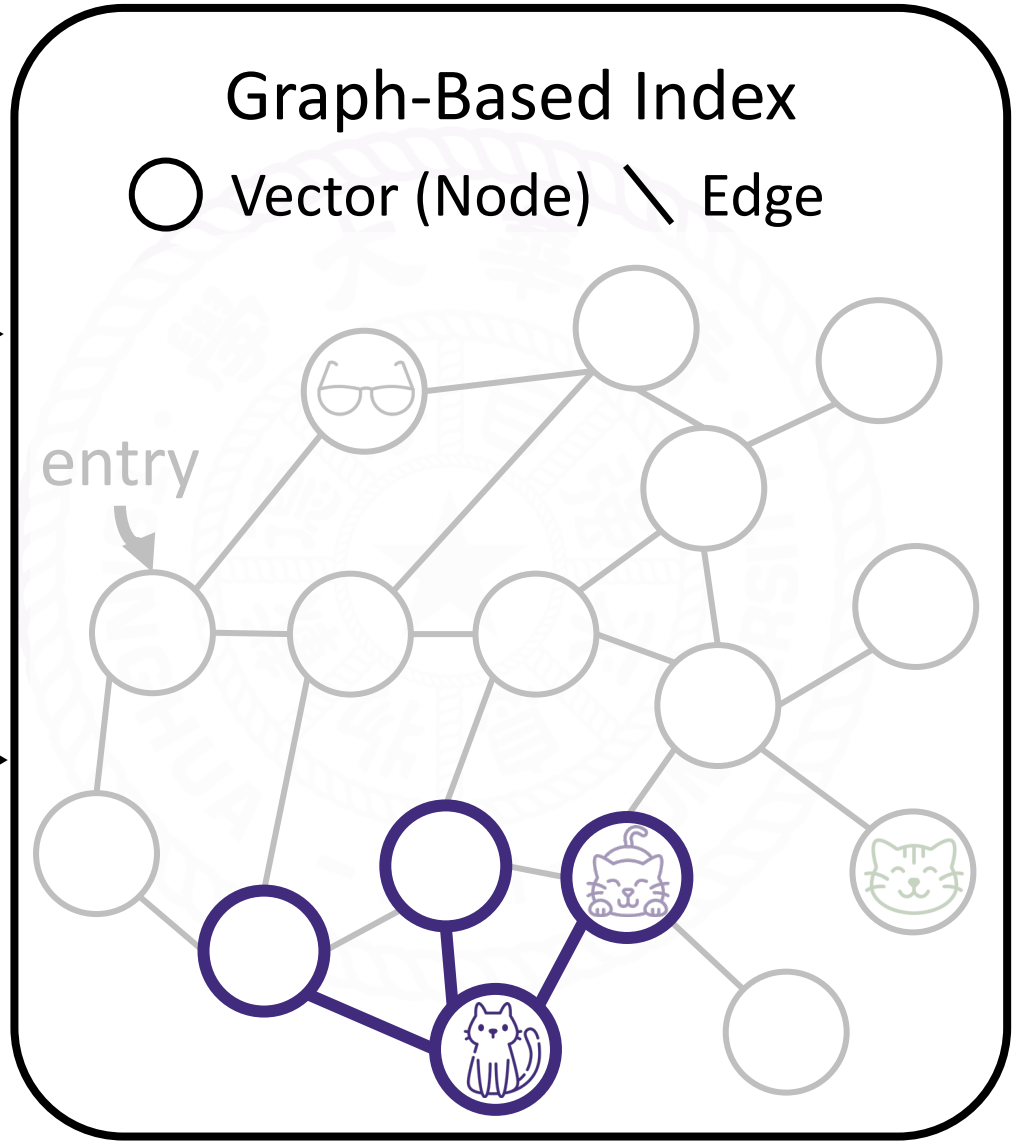

[-5, 1, 2, 3, …]

Build →

New Vector


[2, 5, 7, -9, …]

Insert →



1. Find candidate neighbors using search
2. **Connect to part of candidate neighbors**

Vector Inserts at Large Scale

Demand: **Billions** of vectors, **100 million** inserts per day

Total: **11.7 billion** road/vehicle vectors

New: **100 million** per day

Total: **100 billion** image vectors

New: **500 million** per day

Support **Vector Inserts** on **SSD** meets this demand

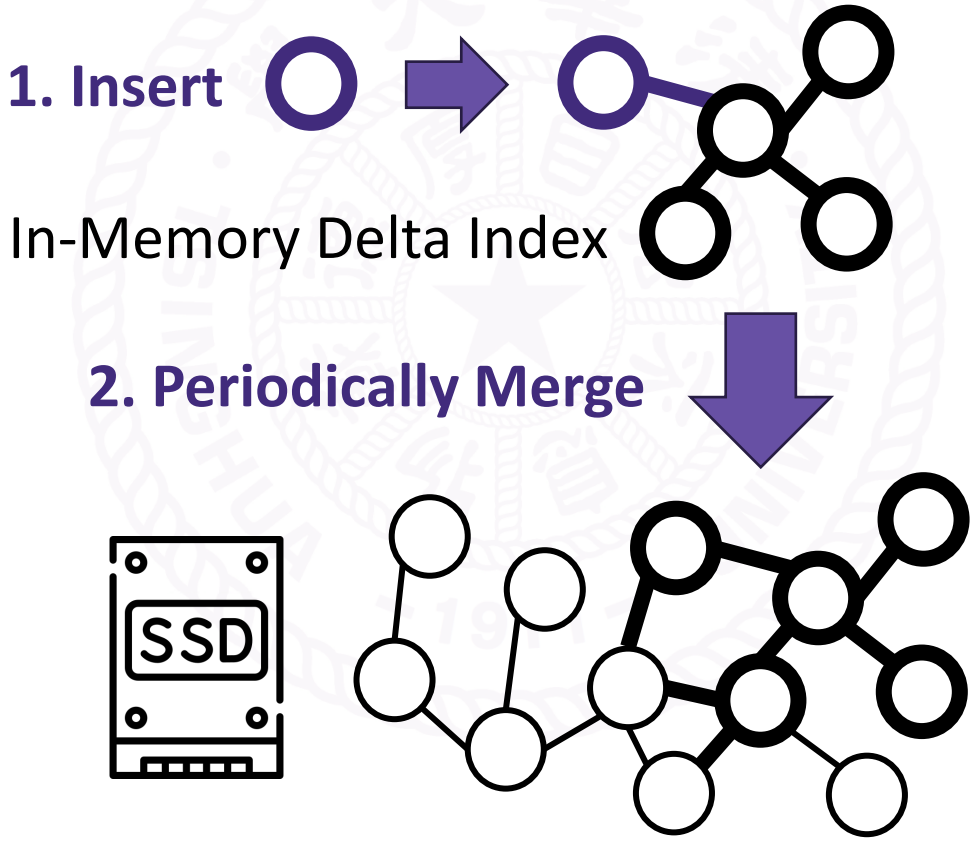
Existing Solution for Vector Inserts

Solution #1: Periodically Rebuild the Index

Index	#Cores	Time
DiskANN	32	**2 days**
SPANN	45	**4 days**

Costly for datasets with billions of vectors

Solution #2: Buffered Insert



Existing Solution for Vector Inserts

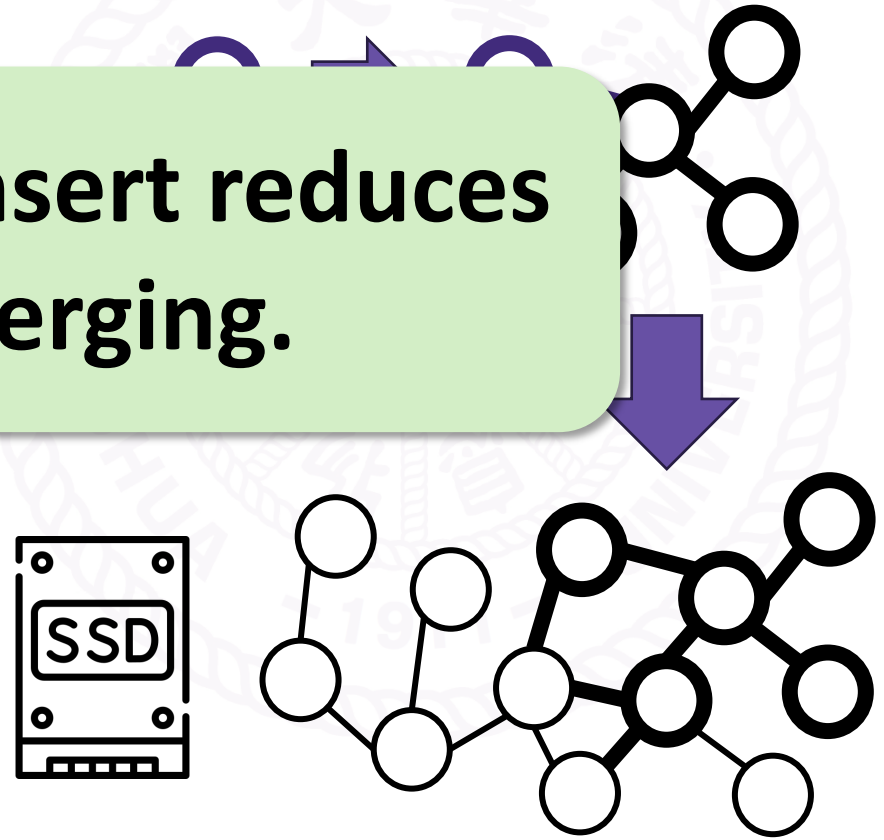
Solution #1:
Periodically Rebuild the Index

Index		
DiskAN		
SPANN	45	4 days

Costly for datasets with billions of vectors

Intuitively, buffered insert reduces SSD writes by merging.

Solution #2:
Buffered Insert

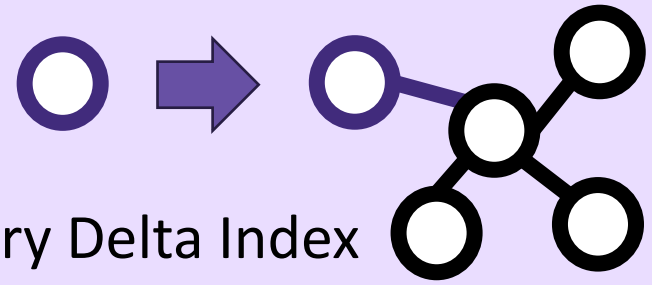


On-Disk Full Index

Limitations of Buffered Insert

Its tradeoff lies in three aspects:

1. Insert

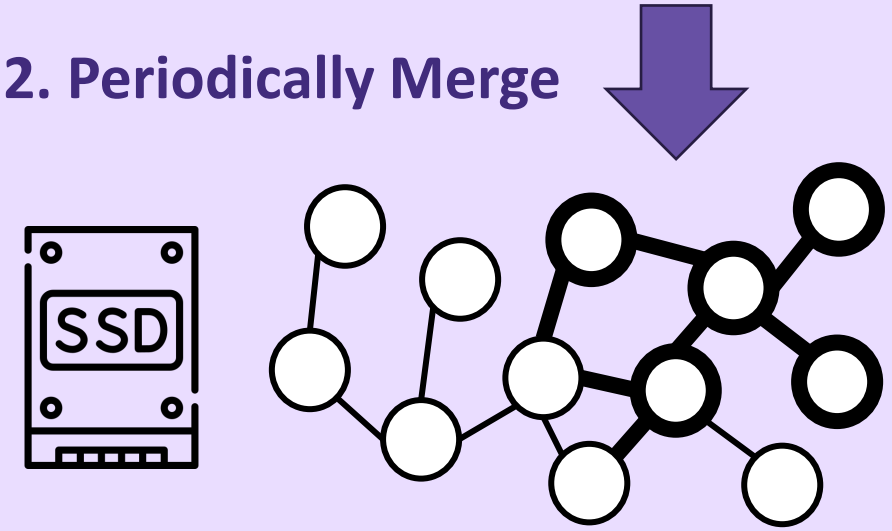


In-Memory Delta Index

#1: Memory overheads:

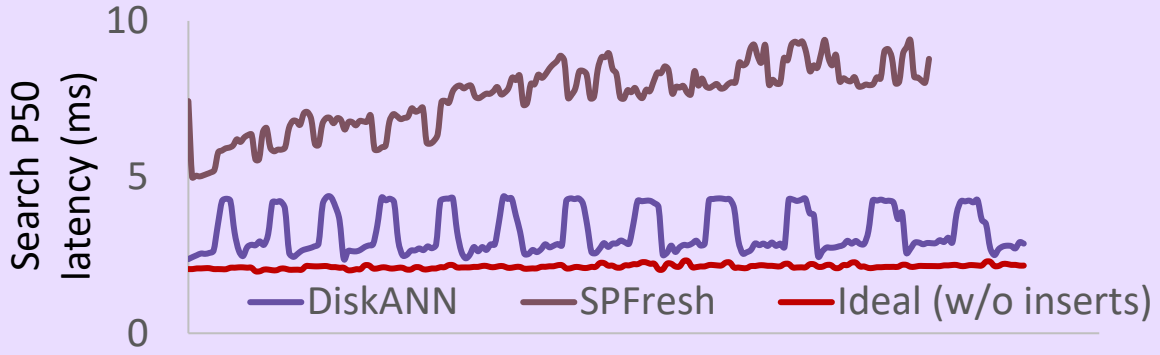
100GB for 30M vectors (**3% of 1B**)

2. Periodically Merge



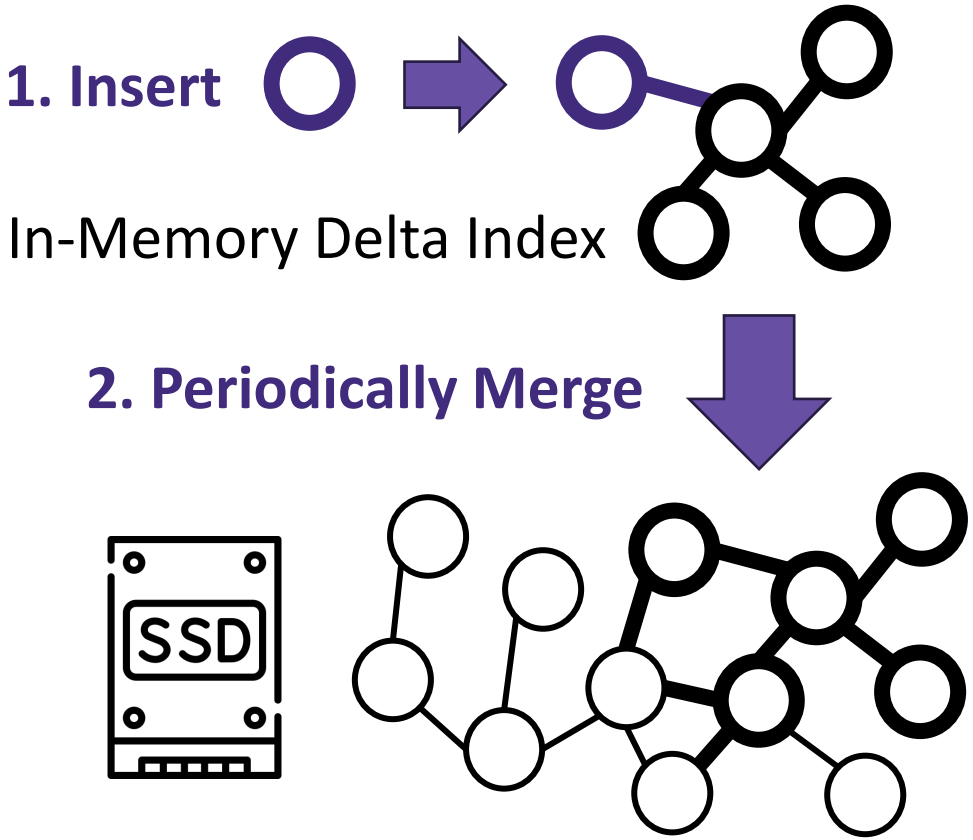
#2: Interfere with frontend search:

Search latency **fluctuates to ~2X**



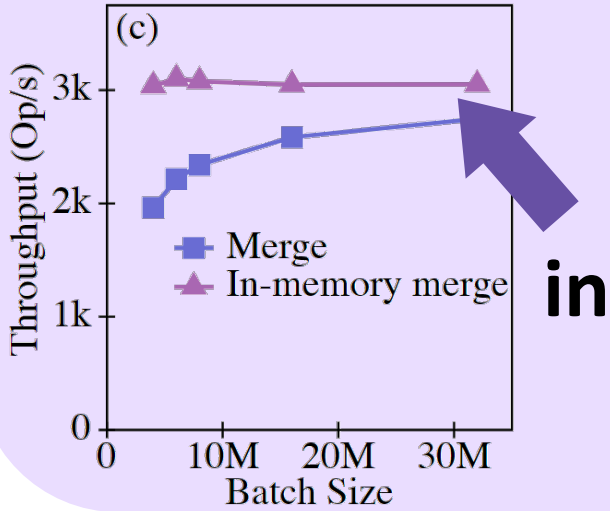
Limitations of Buffered Insert

Its tradeoff lies in three aspects:



#3: The merge algorithm:

Insert **one by one** in memory, then batch apply graph updates to SSD



Bottlenecked by inserting one by one

Limitations of Buffered Insert

Its tradeoff lies in three aspects:

1. In

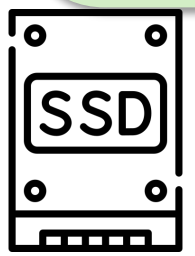
In-M

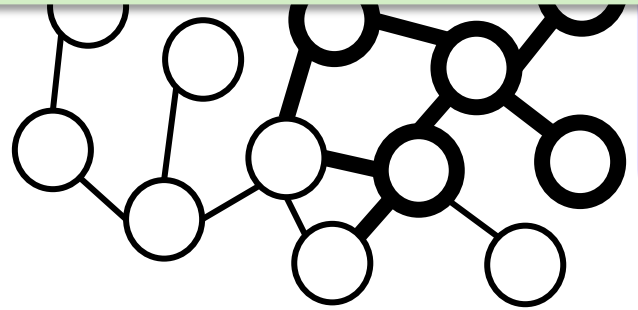
2

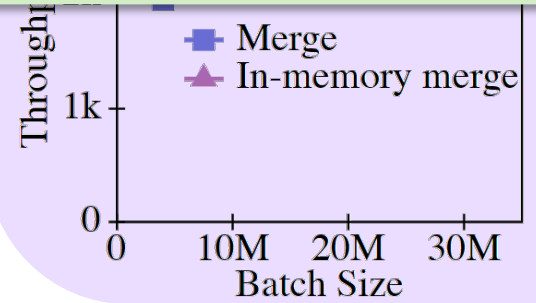
Can we directly insert vectors to SSD, namely

Direct Insert?

Pros: no delta index; amortized merge costs







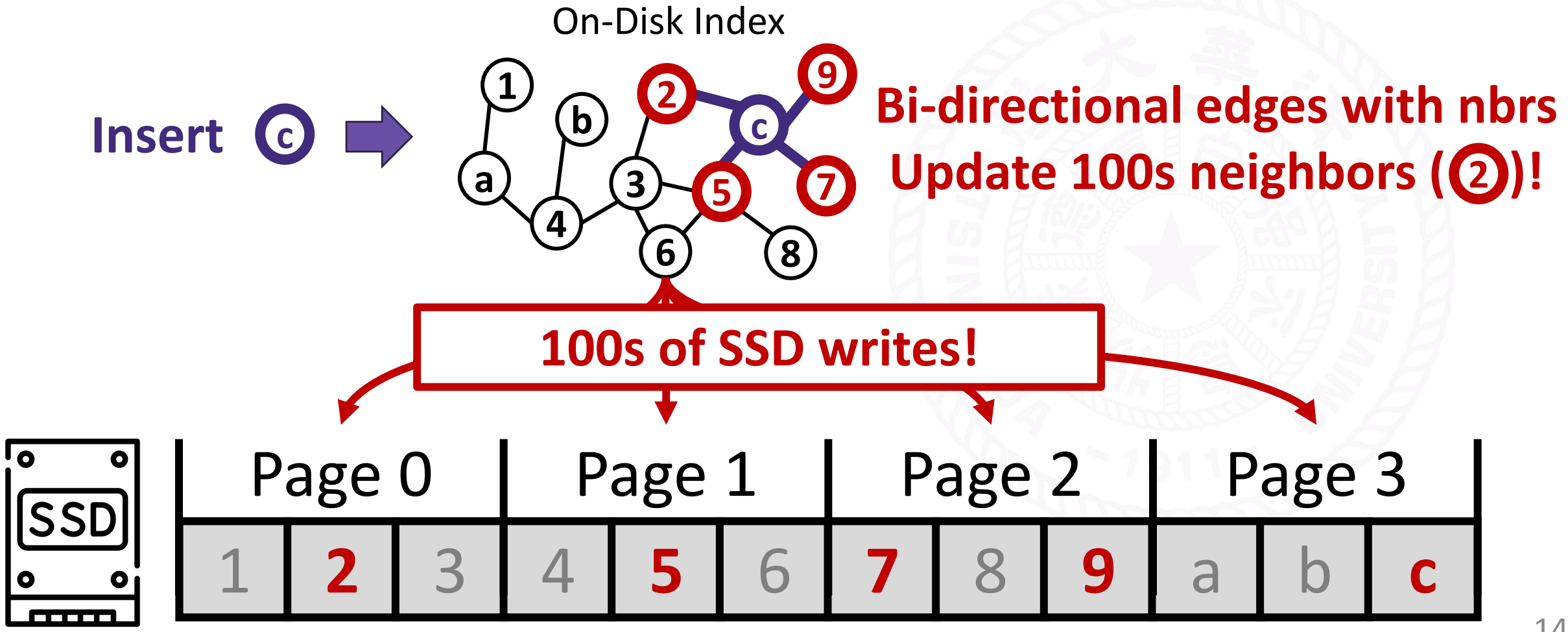
n

D

inserting one by one

Challenge #1: SSD write overheads

Neighbor record updates cannot be merged…

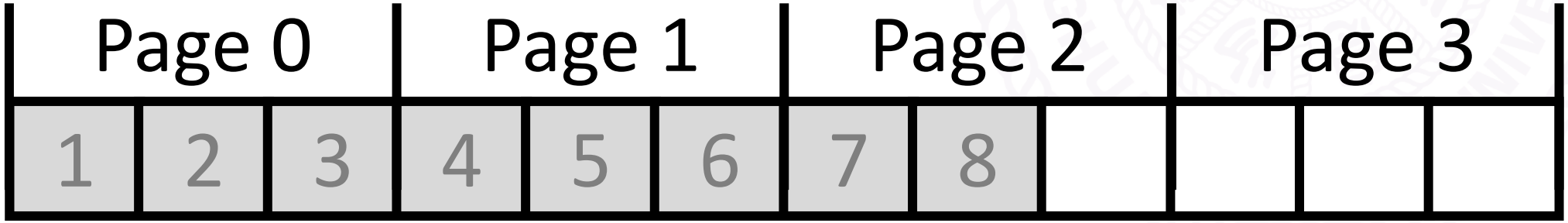


GC-Free Update Combining

Records in the graph are of fixed size, because…

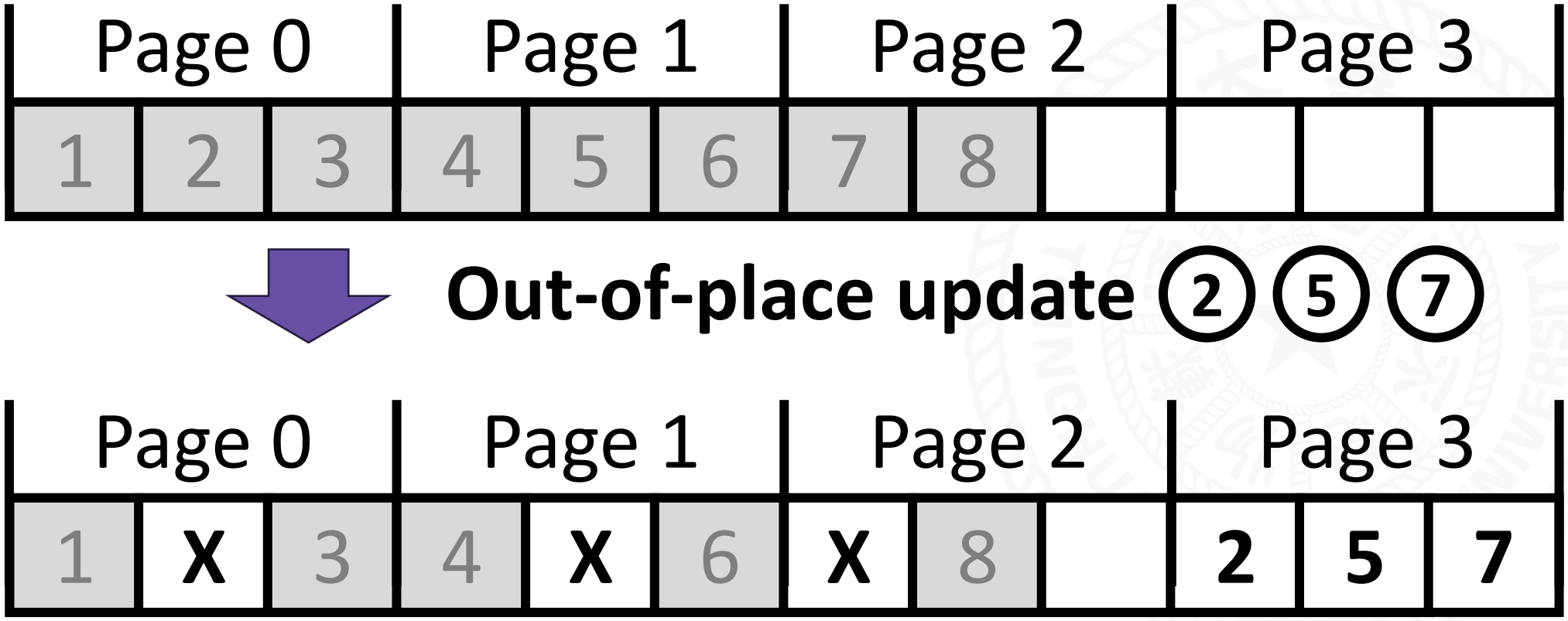
Record for Vector 1

- Vector 1 itself (**fixed dimension** → fixed size)
- Vector 1's nbrs (**max count: R** → fixed size)



GC-Free Update Combining

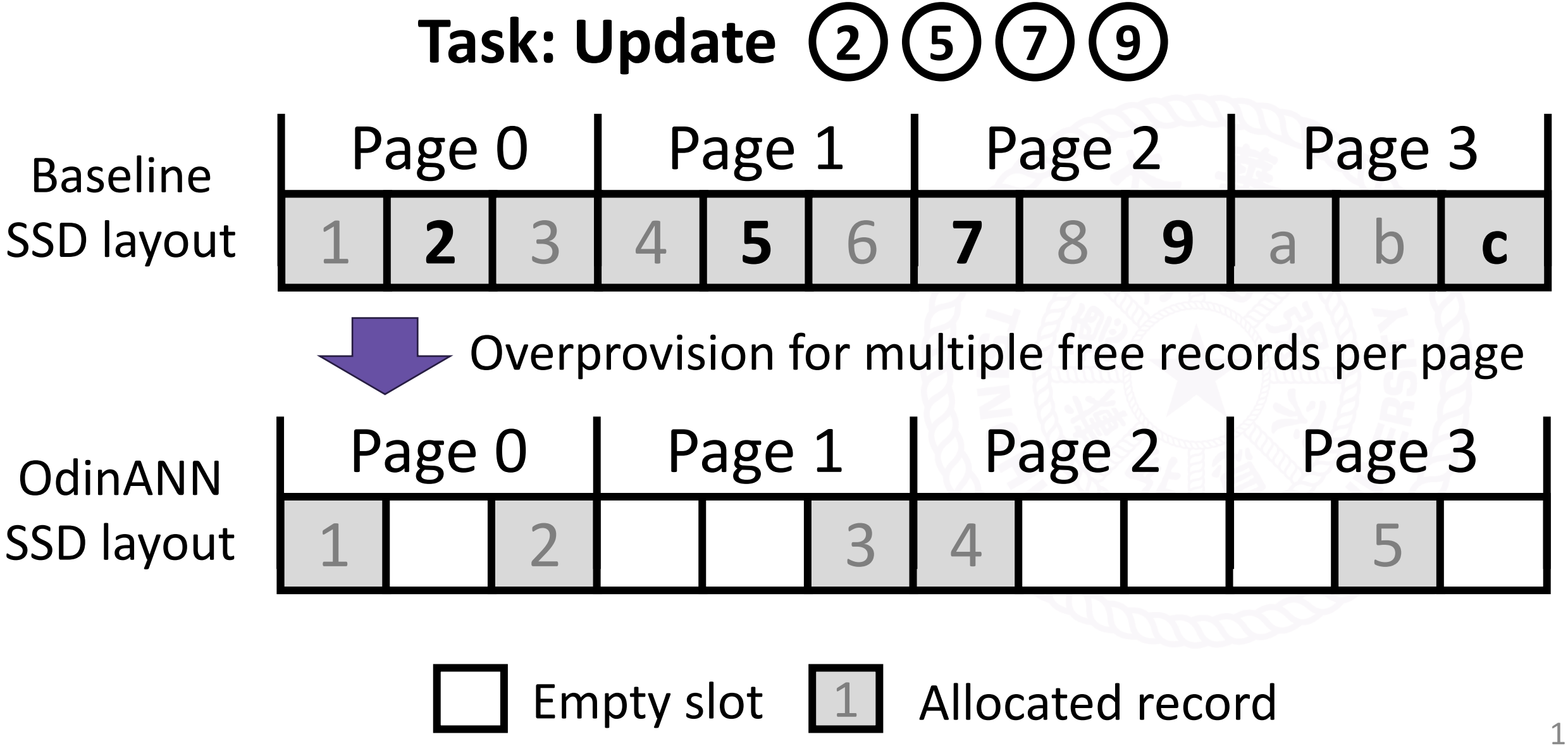
Records in the graph are of fixed size, so…



The old records can be **directly reused** as empty slots!

GC-Free Update Combining

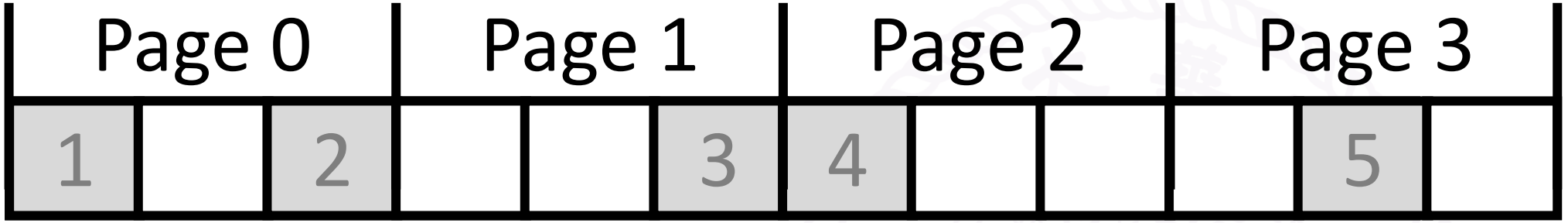
Task: Update ② ⑤ ⑦ ⑨



GC-Free Update Combining

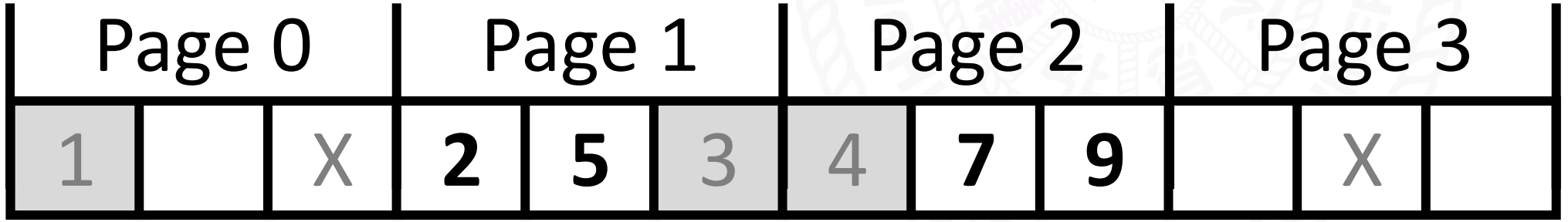
Task: Update ② ⑤ ⑦ ⑨

OdinANN SSD layout



Update the records out-of-place, recycle old ones

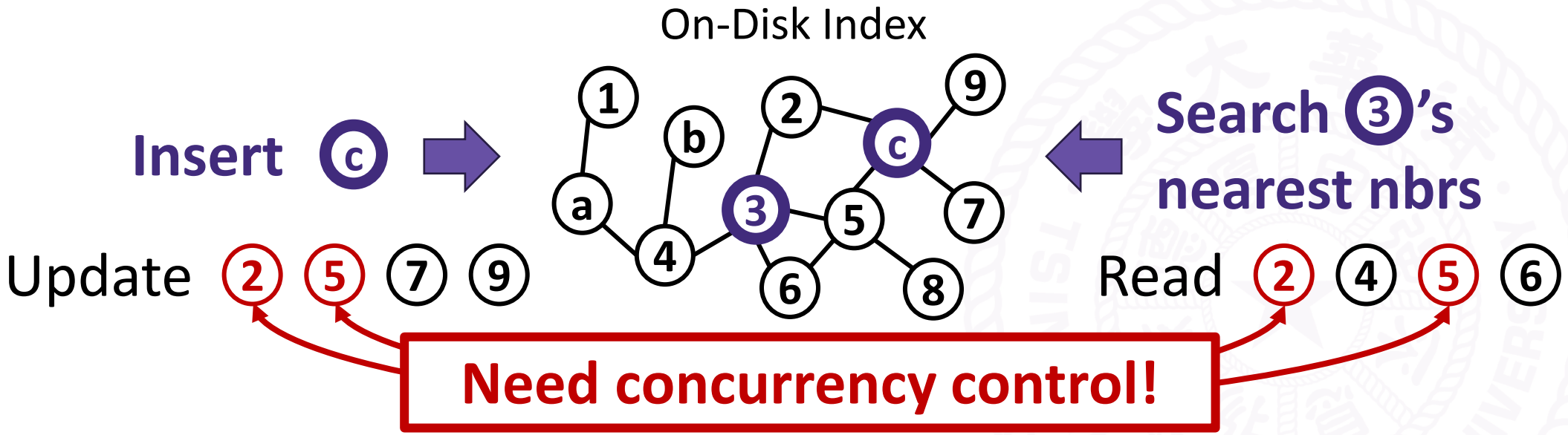
OdinANN After update



3 page writes → **2** page writes

Challenge #2: Concurrency Control

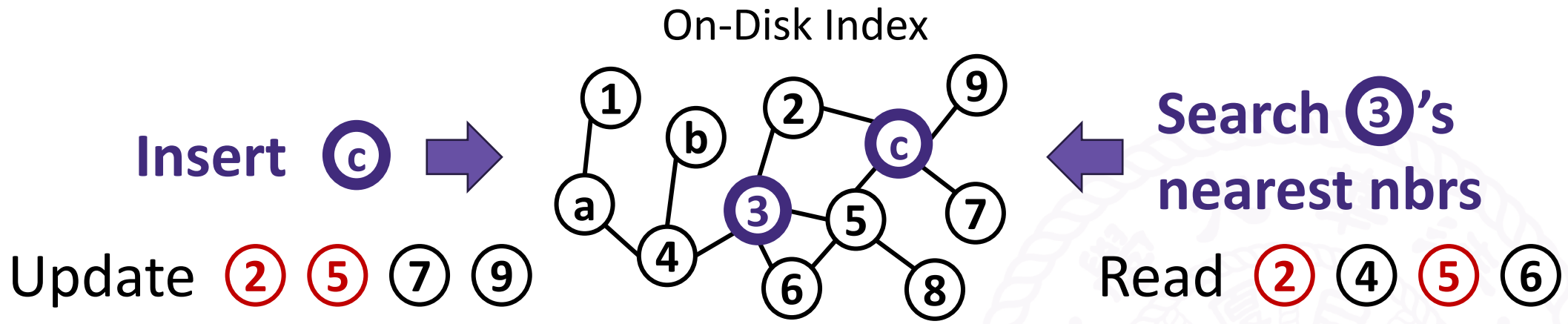
Insert and search may access the same records…



Trivial idea: { 2 5 7 9 } for this insert

But 10ms (each insert takes) critical section…

Approximate Concurrency Control



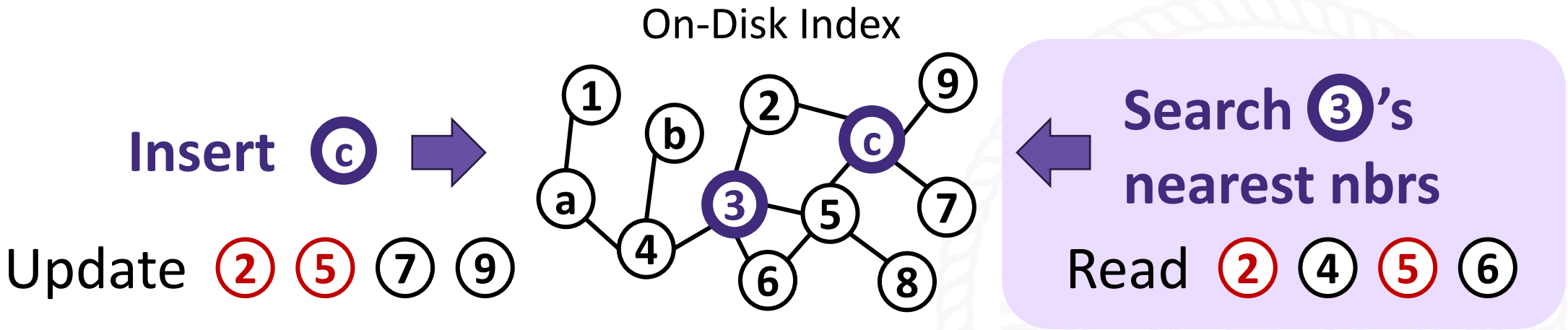
Trivial Op-level lock: 🔒{ 2 5 7 9 } **during insert**

🔒{ 2 4 5 6 } **during search**

Idea: Search & insert are **approximate**, so Op-level lock can be **relaxed**

Approximate Concurrency Control

Method: Only ensure correct records



Search steps: Op-level lock → **Record-level lock**

Lock & Read ② → **Unlock** ② → Lock & Read ④ → **Unlock** ④

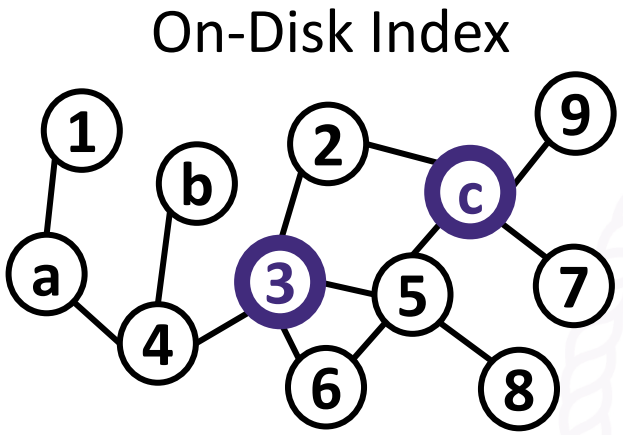
Lock & Read ⑤ → **Unlock** ⑤ → Lock & Read ⑥ → **Unlock** ⑥

Approximate Concurrency Control

Method: Only ensure correct records

Insert ⓒ

Update ② ⑤ ⑦ ⑨



Search ③'s nearest nbrs

Read ② ④ ⑤ ⑥

Insert steps: Avoid update loss by **validation**

Find ⓒ's neighbors ② ⑤ ⑦ ⑨ by **Search** → **Lock** them

→ **Validate** by re-reading them → **Update** and unlock

Put Them Altogether: OdinANN

A large-scale, updatable vector search system

- ***Direct Insert*** to reduce interference with frontend search
- Two techniques for efficiency:
  - ***Update Combining*** to reduce write amplification
  - ***Approximate CC*** to shorten critical section

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)

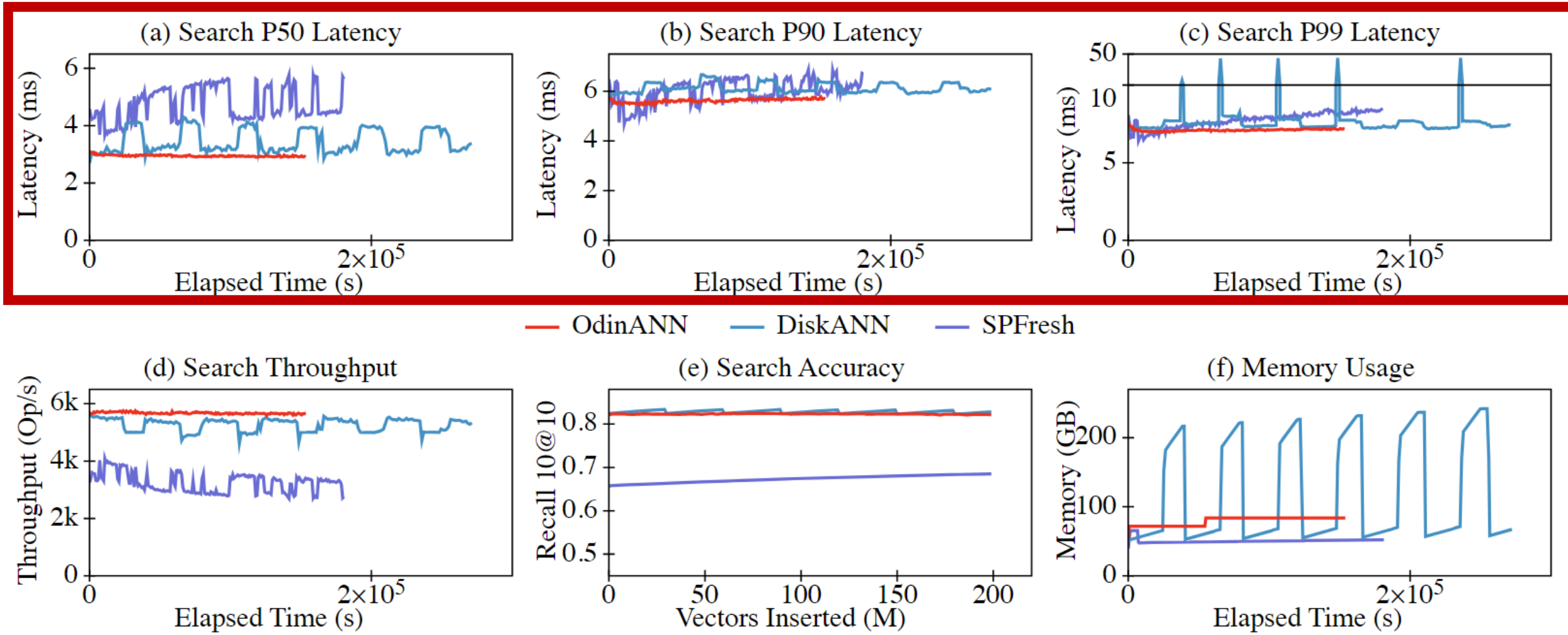
Compared with on-SSD updatable vector search systems:

- **Graph-based:** FreshDiskANN [CoRR'21]
- **Cluster-based:** SPFresh [SOSP'23]

Vector datasets:

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

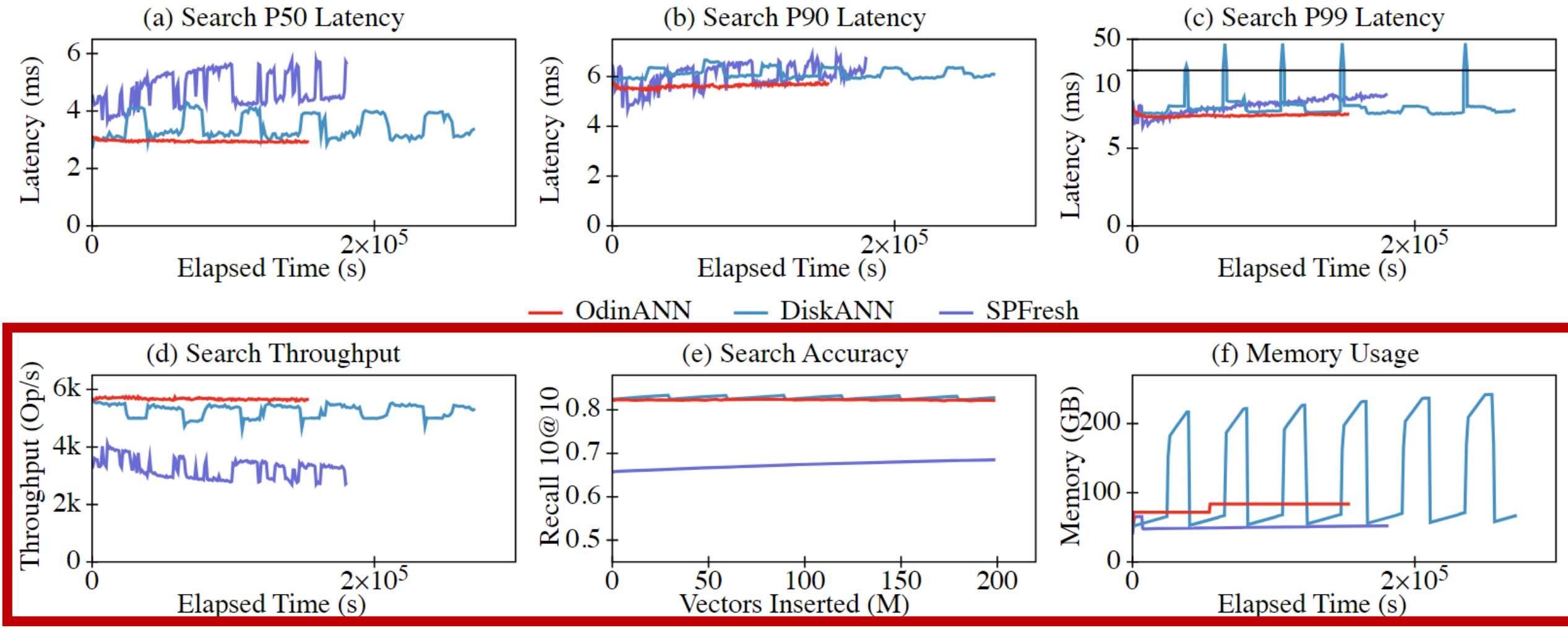
Overall Performance: Billion-Scale Insert-Search



With insert QPS = 1000 and search QPS > 5000, OdinANN has:

Stable search latency compared to DiskANN and SPFresh

Overall Performance: Billion-Scale Insert-Search



Stable search throughput, unsacrificed accuracy, and less memory usage (90GB for billion-scale) compared to DiskANN

Thank you! Questions are welcome.

PipeANN: a **low-latency**, **large-scale**, and **updatable** vector store

- Low latency at large scale: **PipeSearch (OSDI '25)**
- Efficient updates at large scale: **Direct insert (this work)**

PipeANN achieves:

- ***< 1ms*** search latency
- ***20K QPS*** search throughput
- ***< 90GB*** memory usage

} When searching or updating ***billions (TBs) of*** vectors

Artifact: github.com/thustorage/PipeANN

Contact: gh23@mails.tsinghua.edu.cn

image: Flaticon.com ☺