

# Exploiting Intel Optane Persistent Memory for Full Text Search

Shoaib Akram  
ANU, Canberra  
[shoaib.akram@anu.edu.au](mailto:shoaib.akram@anu.edu.au)



Australian  
National  
University

# Full text search is ubiquitous

Web search



Retail



Social media



# Search = Indexing + Query eval

Indexing builds an inverted index

word1 → document-list  
word2 → document-list

Query evaluation searches for words

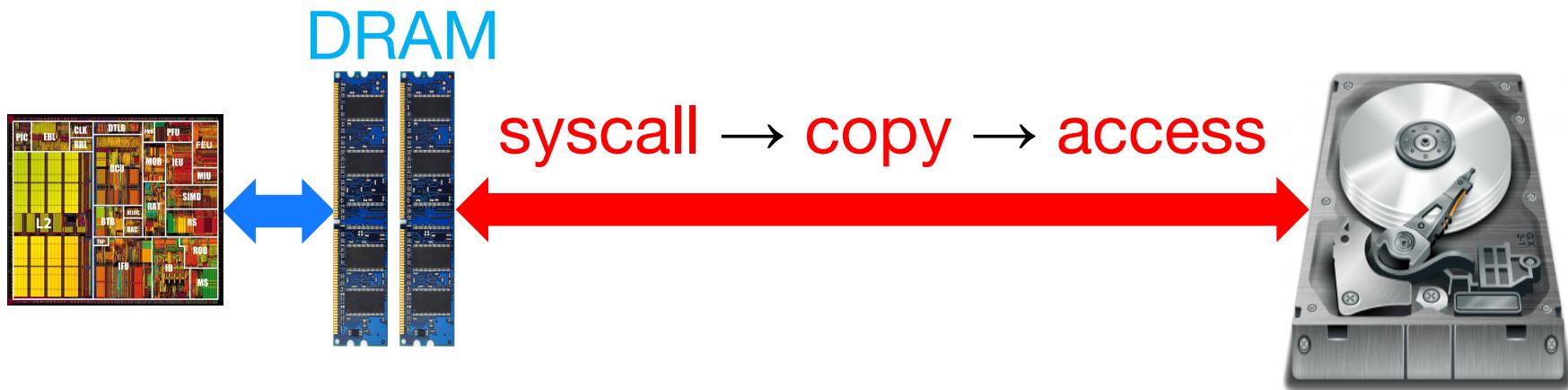


Indexing speed increasingly critical



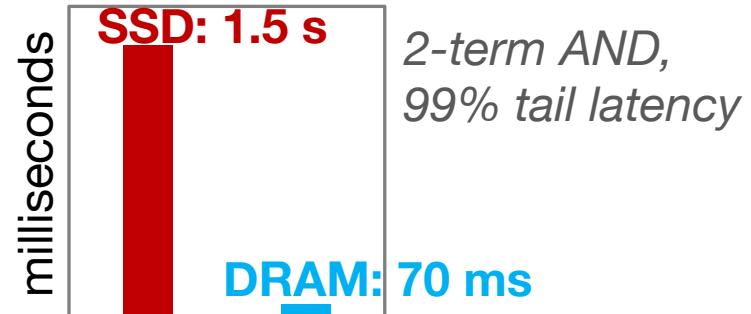
# Challenge: I/O intensity

Writing & merging partial **indices** on storage takes up **40%** of exec time



# Challenge: DRAM capacity

NVMe SSD violates **real time** response constraint



- :( Data growth outpaces DRAM scaling
  - Data volume → 2X
  - DRAM GB/\$ → 20%

# Today: Give up **real time**, or give up **cost efficiency**

Looking forward

Reduce I/O overhead

Find a **fresh memory** scaling roadmap

# Persistent memory (PM)

4X denser than DRAM

Load/store access

Non-volatile



# Contribution: PM Search Engine

Exploiting PM for building/storing indices

- Memory, storage, universal roles
- Fine-grained crash consistent recovery

Extensive PM evaluation vs DRAM/SSD

- Indexing perf, scalability, bottlenecks
- Tail latency of query workloads

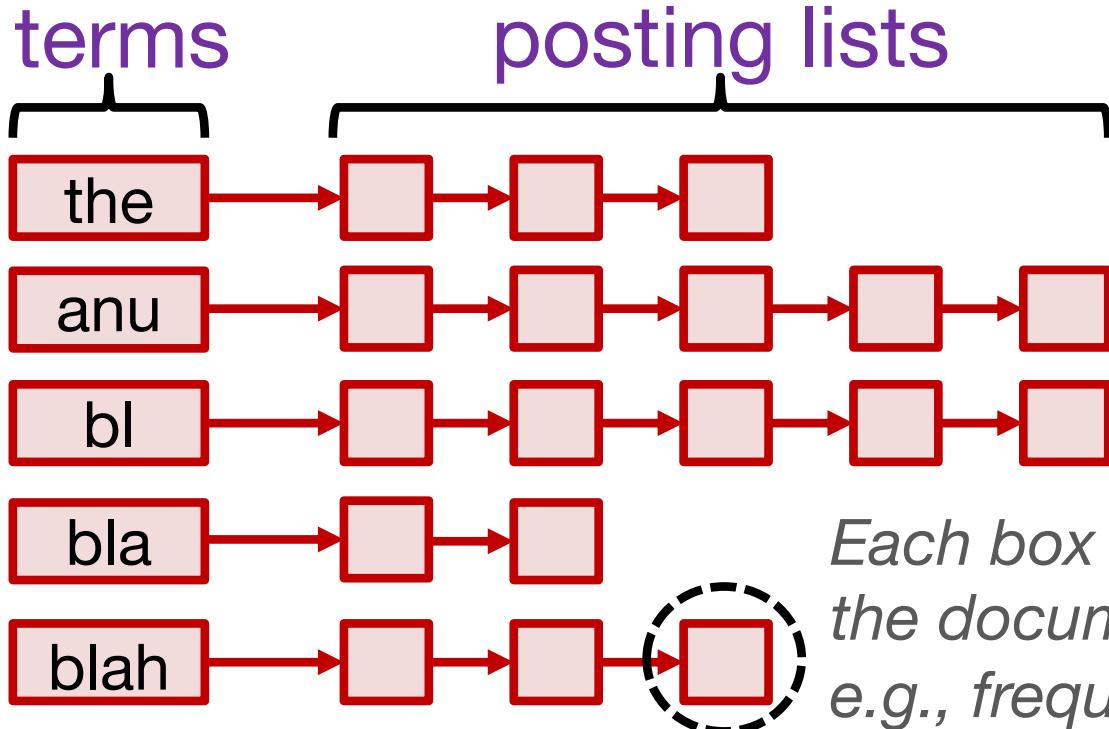
# Rest of the talk

Building an index

Exploiting PM

Evaluation

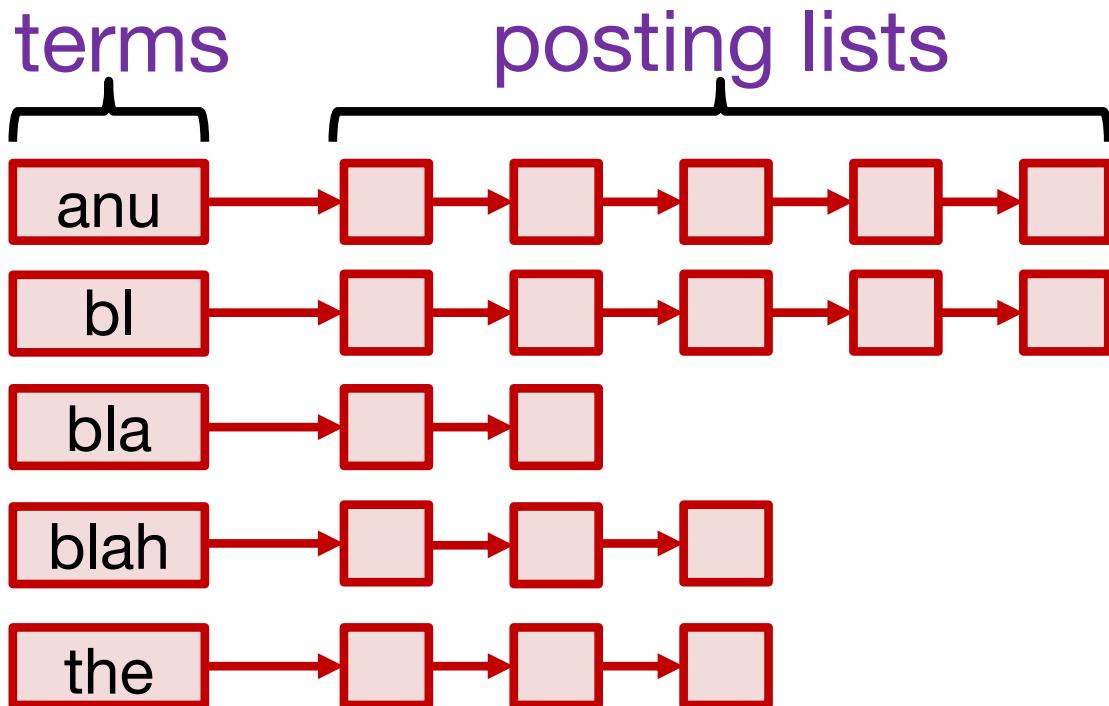
# Step 1: Building the hash table



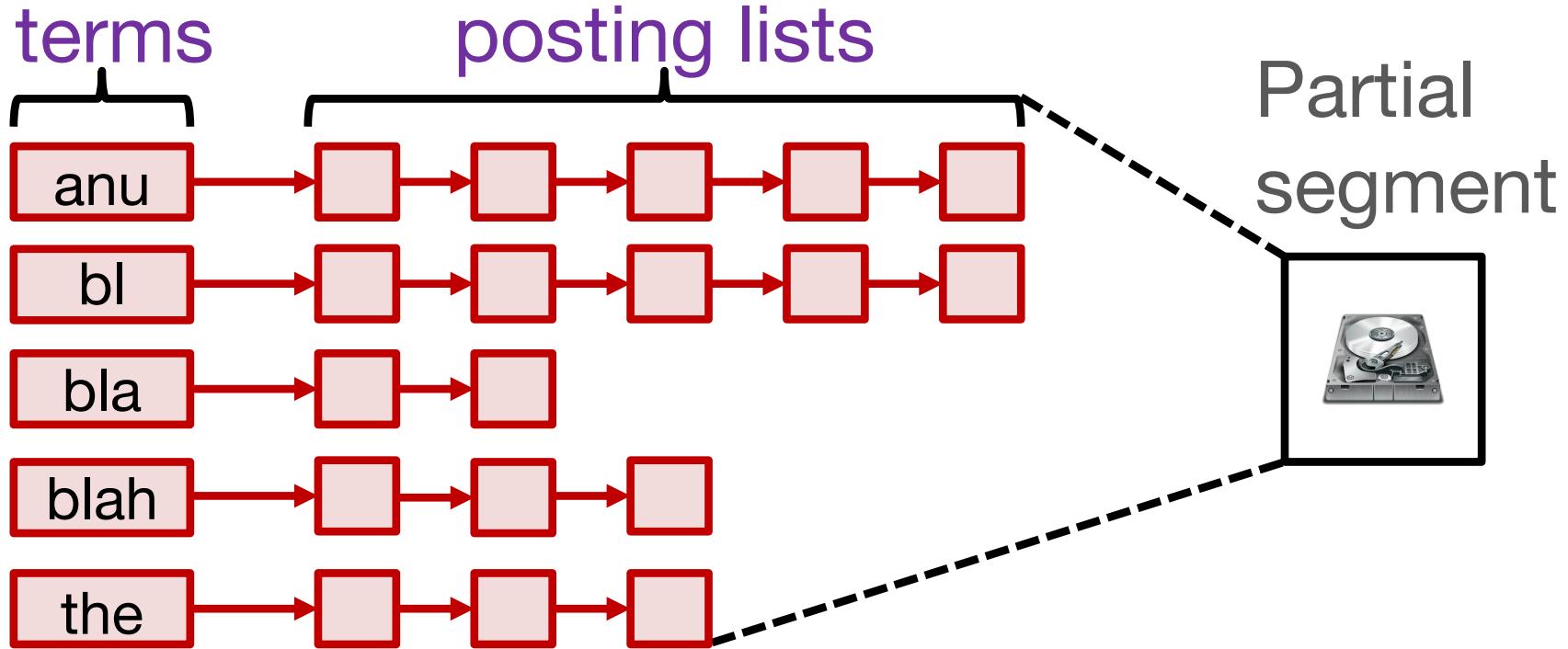
When the table  
is full → Step 2

*Each box is a posting. It contains the document id plus meta-data, e.g., frequency and position of terms*

# Step 2: Sorting the hash table



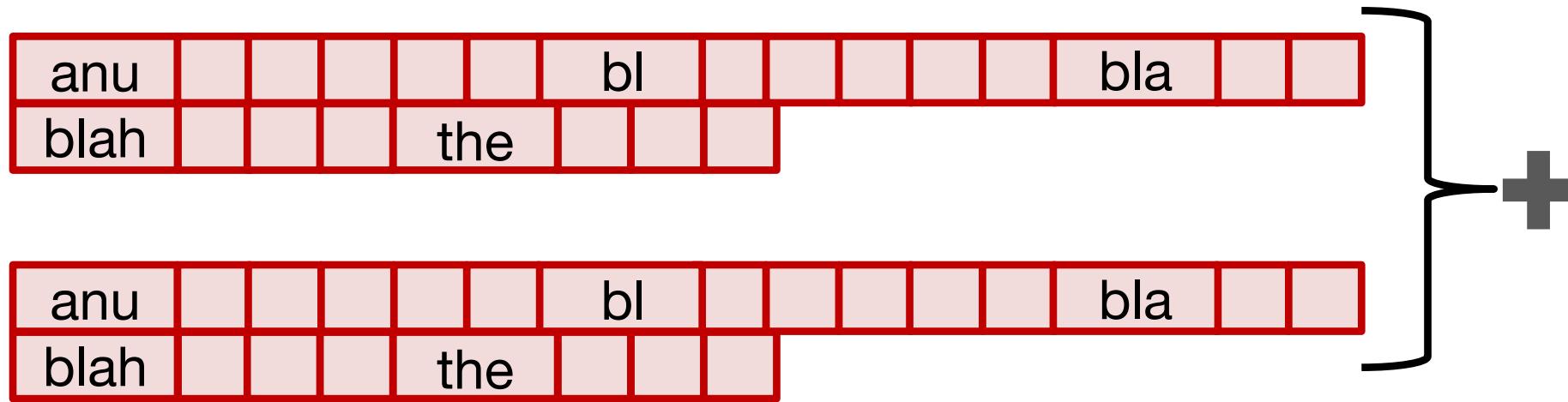
# Step 3: Flushing the hash table



Flushing results in large amounts of sequential I/O

# Step 4: Merging segments

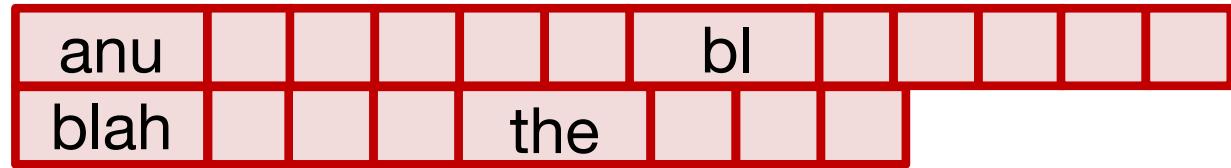
Merging segments is crucial for fast **query evaluation**



Merging results in **large amounts of read/write I/O**

# Index = Segment + Dictionary

term	offset
anu	0
bl	6



Segment: Sequentially sorted postings on storage

Dictionary: To find posting lists in segments, indexers use a key-value store, such as, Berkeley DB

# Different ways to exploit PM

Hash table, DRAM → PM

Partial segments, SSD → PM

Merged segments, SSD → PM

Dictionary, SSD → PM

# PM configurations for indexing

Name of Configuration	Placement of Table, Postings, and Dictionary				Role of Optane PM
	H Table	Partial St	Merged St	Dict	
stock	DRAM	SSD	SSD	SSD	none
table-pm	PM	SSD	SSD	SSD	main memory
pm-only	PM	PM	PM	PM	universal
hybrid	DRAM	PM	PM	PM	storage
hybrid+	DRAM	PM	PM	SSD	storage

# PM configurations for indexing

Name of Configuration	Placement of Table, Postings, and Dictionary				Role of Optane PM
	H Table	Partial St	Merged St	Dict	
stock	DRAM	SSD	SSD	SSD	none
table-pm	PM	SSD	SSD	SSD	main memory
pm-only	PM	PM	PM	PM	universal
hybrid	DRAM	PM	PM	PM	storage
hybrid+	DRAM	PM	PM	SSD	storage

# PM configurations for indexing

Name of Configuration	Placement of Table, Postings, and Dictionary				Role of Optane PM
	H Table	Partial St	Merged St	Dict	
stock	DRAM	SSD	SSD	SSD	none
table-pm	PM	SSD	SSD	SSD	main memory
pm-only	PM	PM	PM	PM	universal
hybrid	DRAM	PM	PM	PM	storage
hybrid+	DRAM	PM	PM	SSD	storage

# Crash consistent indexing

Crash consistent segment flushing

- Use `pmem_persist(segment)`
- Track progress (docIds)

Crash consistent merging

- Tracking progress is tricky
- Details of “logging” in the paper

# Baseline Engine

Psearchy

## MOSBENCH

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, Nickolai Zeldovich

[mosbench@pdos](mailto:mosbench@pdos)

MOSBENCH is a set of application benchmarks designed to measure scalability of operating systems. It consists of applications that previous work has shown not to scale well on Linux and applications that are designed for parallel execution and are kernel intensive. The applications and workloads are chosen to stress important parts of many kernel components.

Native, fast, and flexible

Easily integrated with Intel PMDK

# Indexing Methodology

## Dataset and measurement

- Wikipedia English (**DRAM**)
- Execution time
- 1 GB HT per core, up to 32 cores

## PM setup

- Interleaved, local, EXT4+**DAX**
- pmemkv dictionary [github.com/pmem/pmemkv](https://github.com/pmem/pmemkv)

# Experimental Platform

Our in-house server with DRAM, PM, & SSD

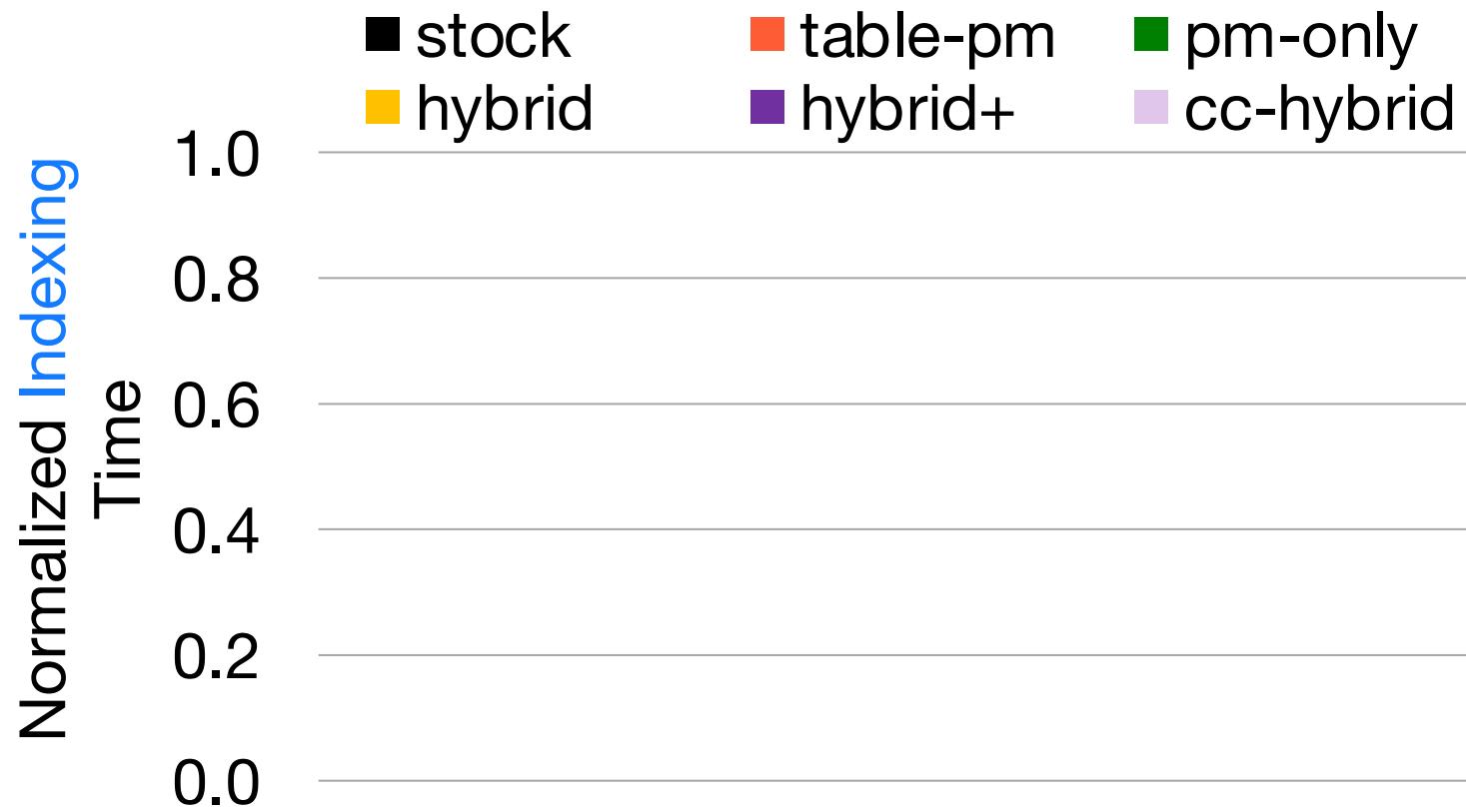
2 TB PM

0.5 TB DRAM

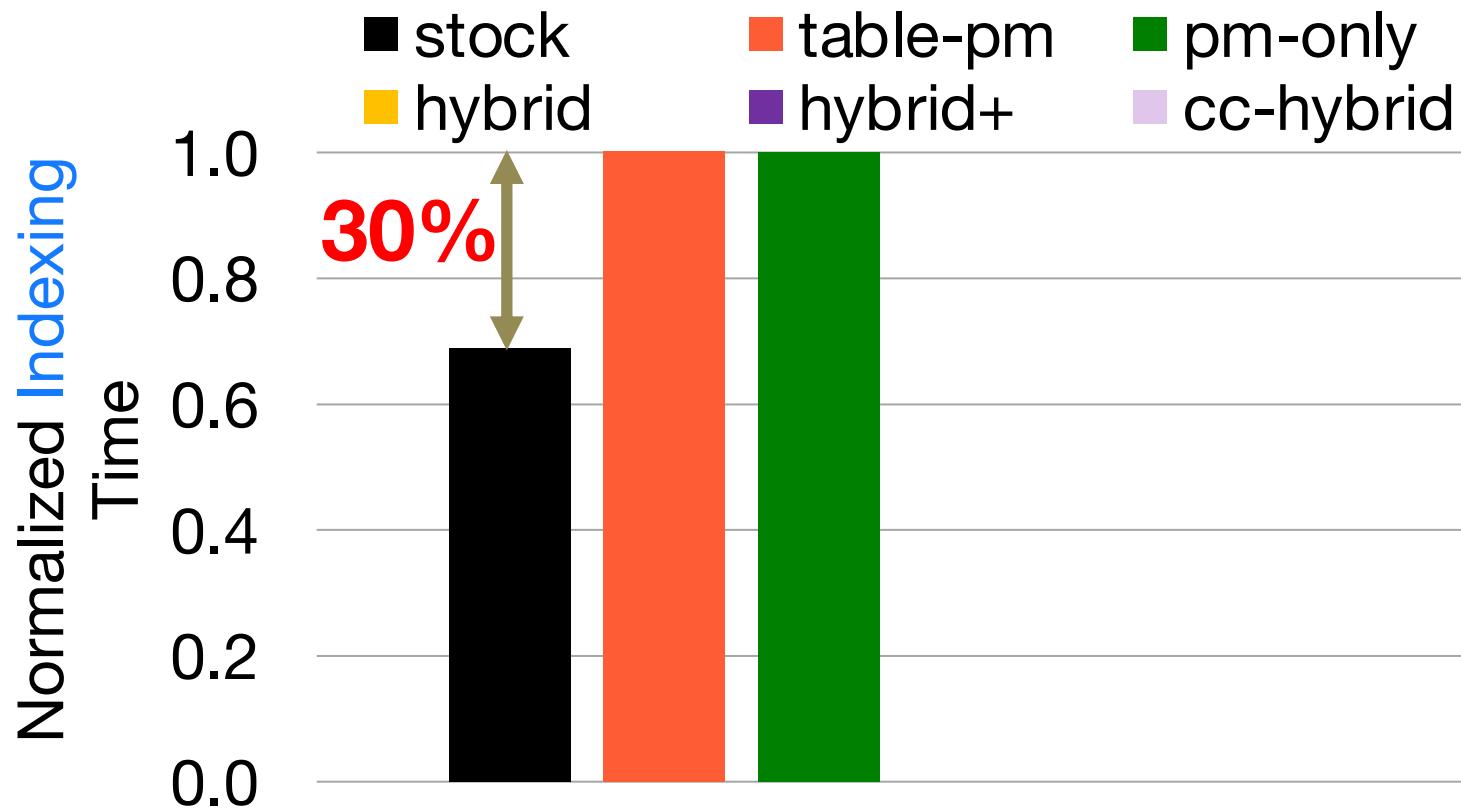
1.5 TB NVMe Optane SSD



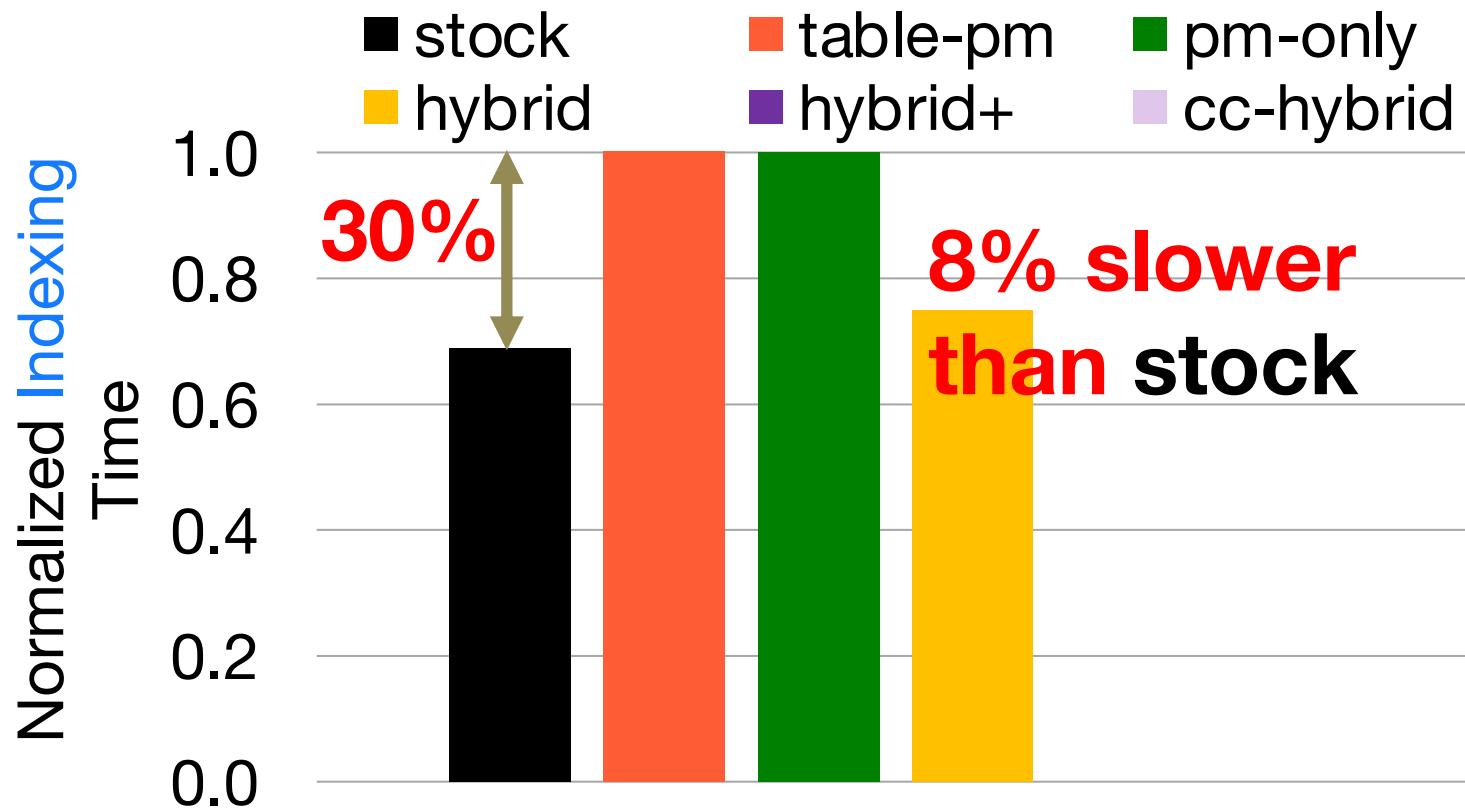
# Indexing perf with one core



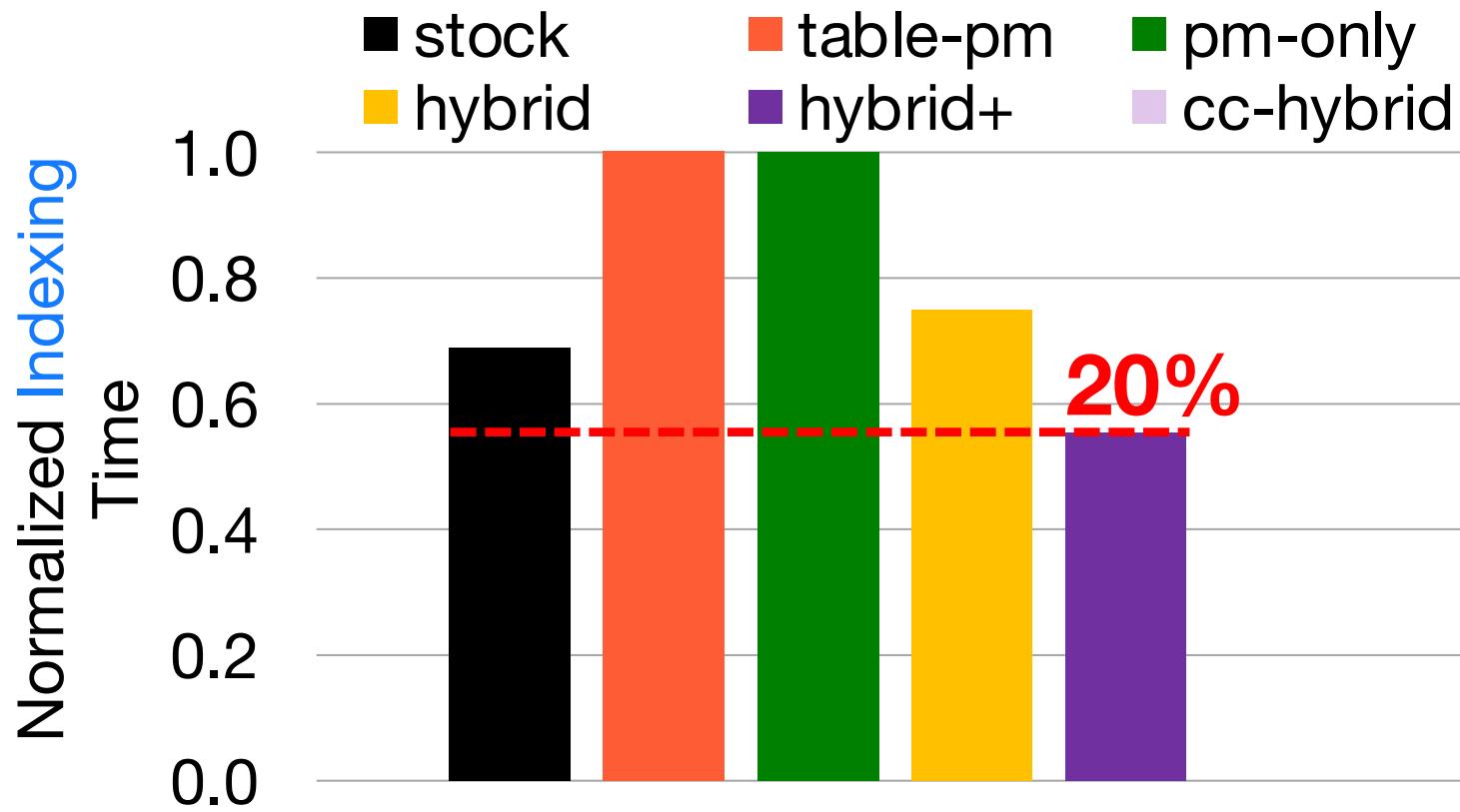
# PM as main/only is 30% slower



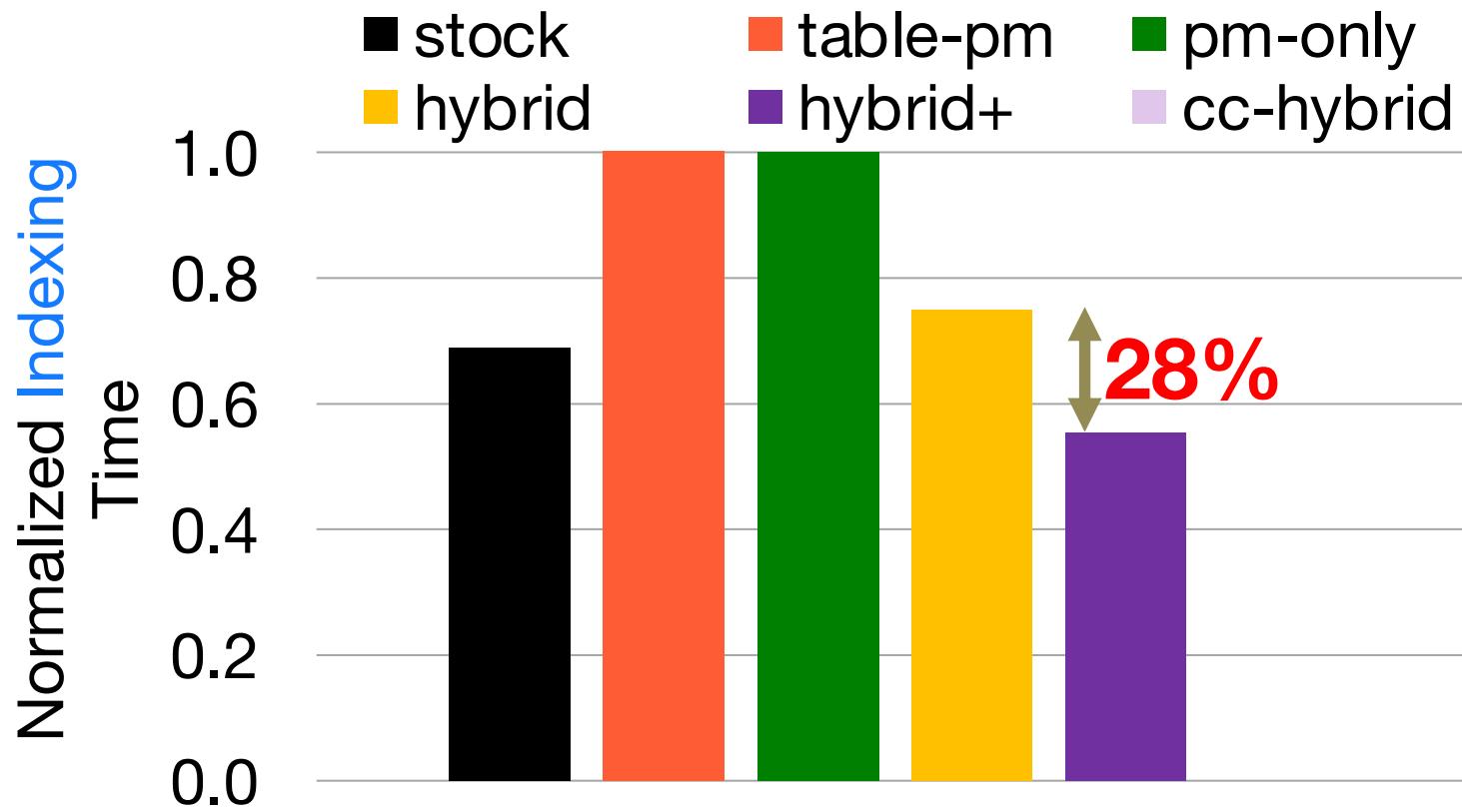
# Hybrid is 8% slower than stock



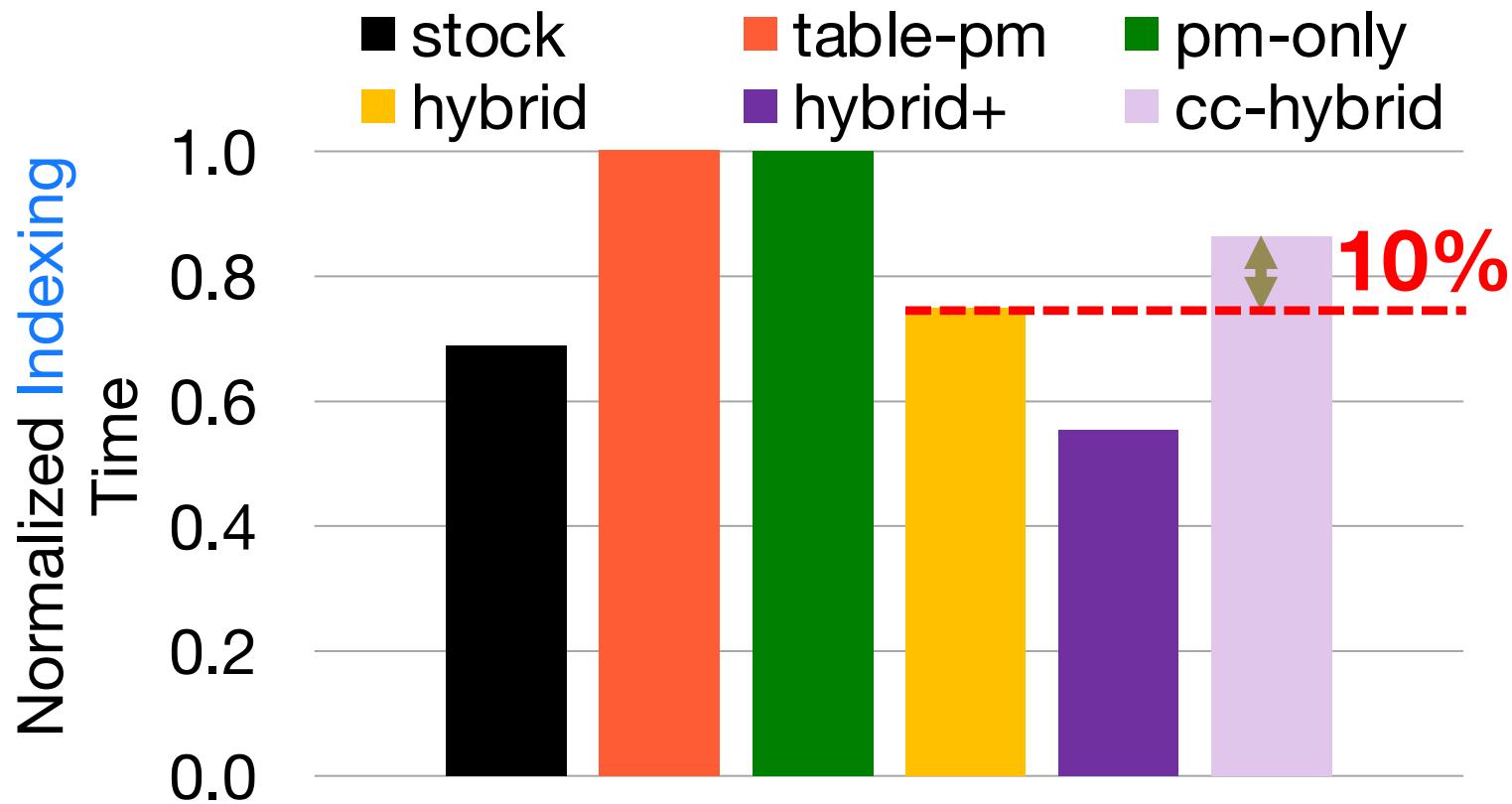
# Hybrid+ is best, 20% over stock



# Hybrid+ is best, pmkv costs 28%

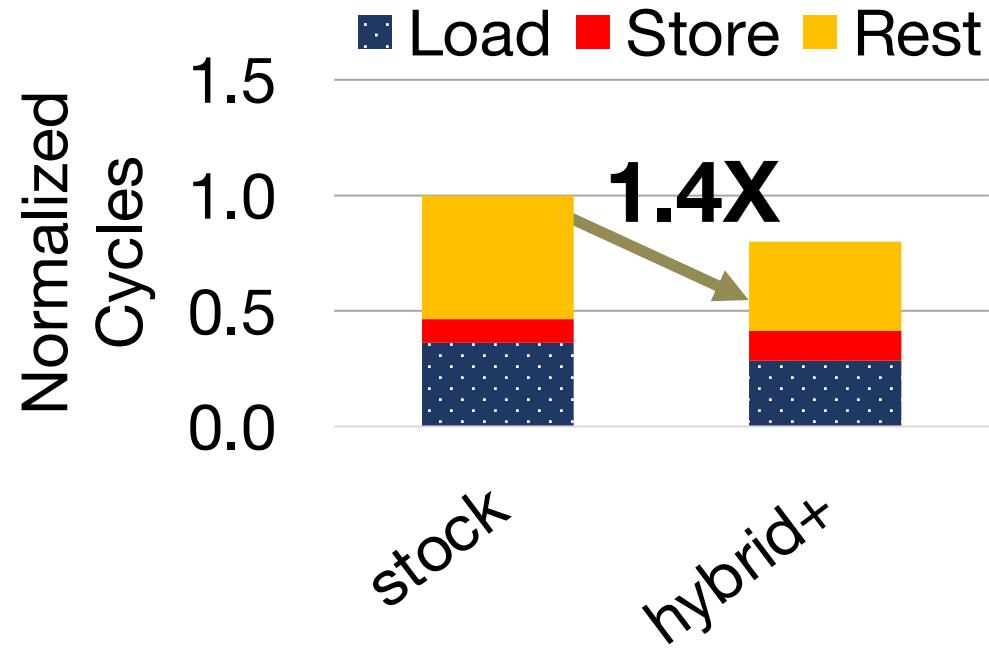


# Crash consistency costs 10%

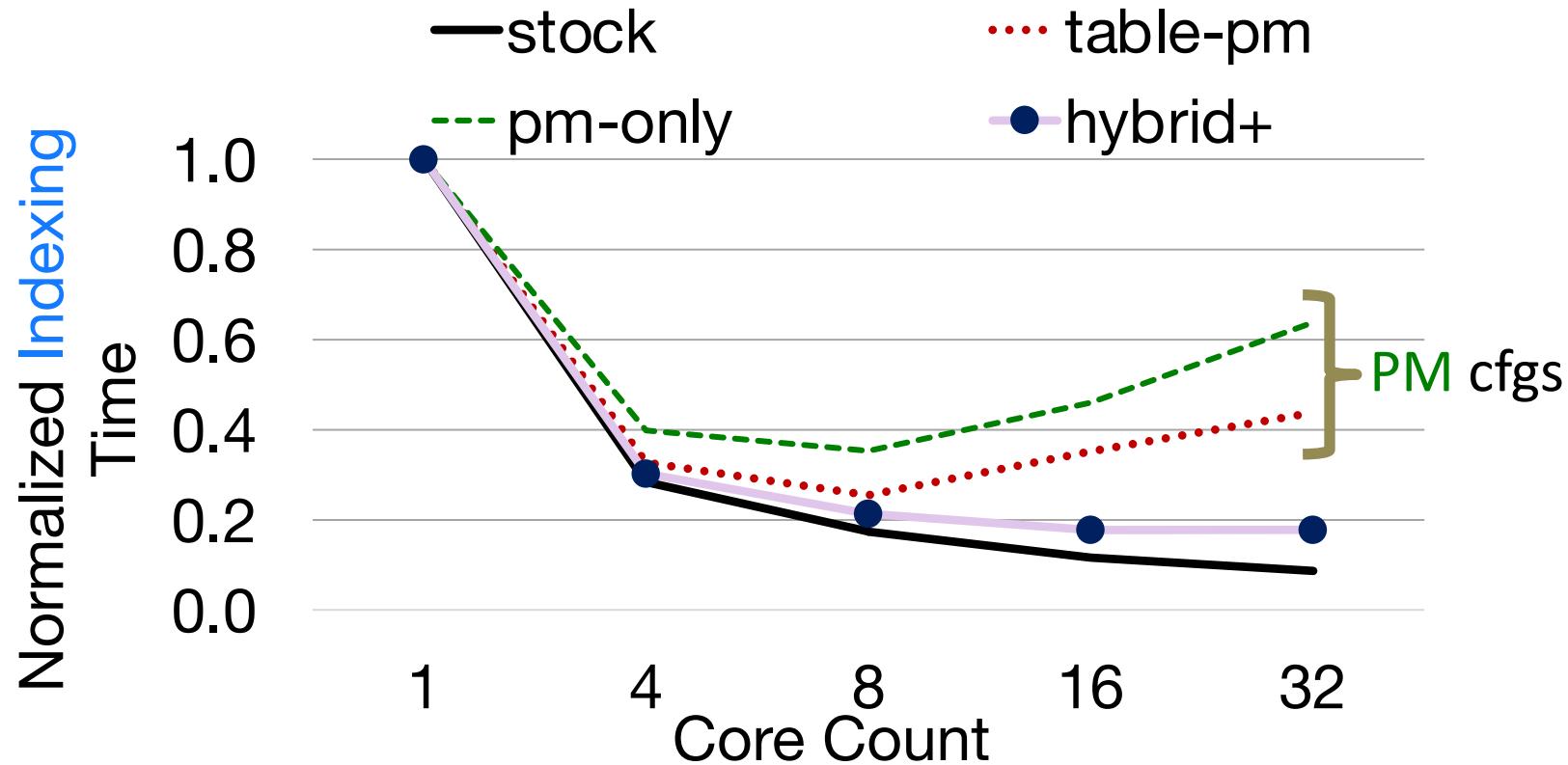


# syscall → mmap is mainly why hybrid+ beats stock

Use perf counters to observe Load/Store stalls the multicore incurs

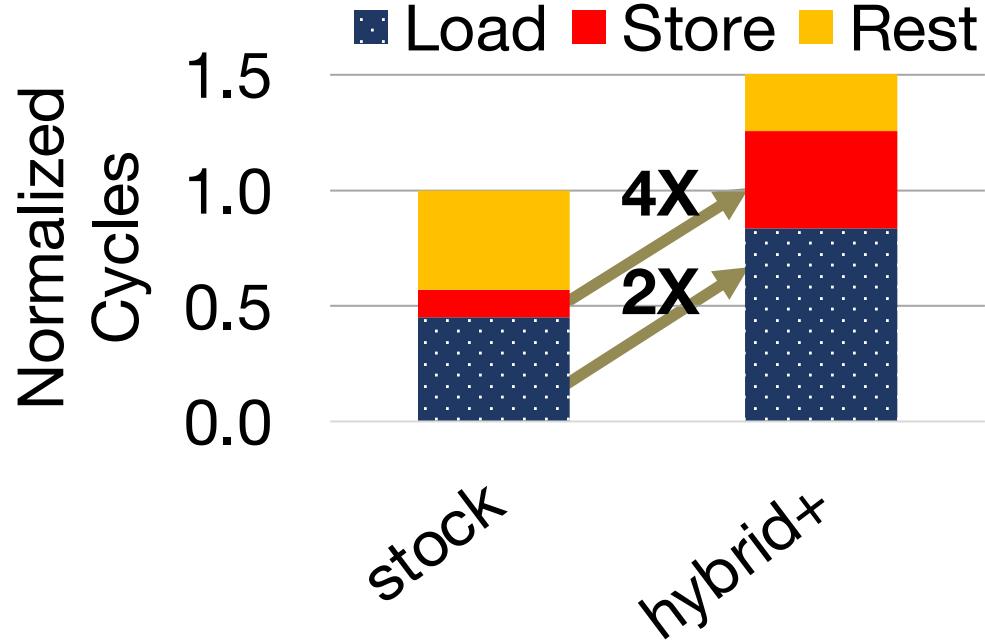


# Indexing scalability



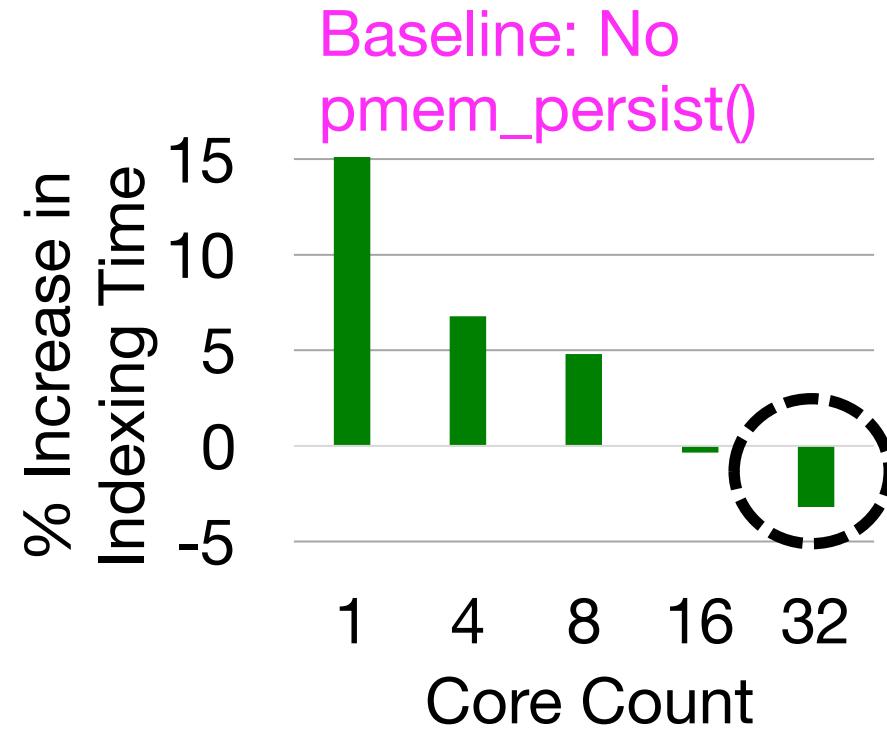
# Hybrid+ incurs an increase in memory stalls (32 cores)

Use perf counters to observe Load/Store stalls the multicore incurs



# Crash consistent indexing with 32 cores improves perf

32 cores: Invalidated cache lines become replacement candidates, improving LLC hit rate



# Query Evaluation Methodology

Tail latency of **100K** concurrent queries

- 1 term
- AND 2 terms

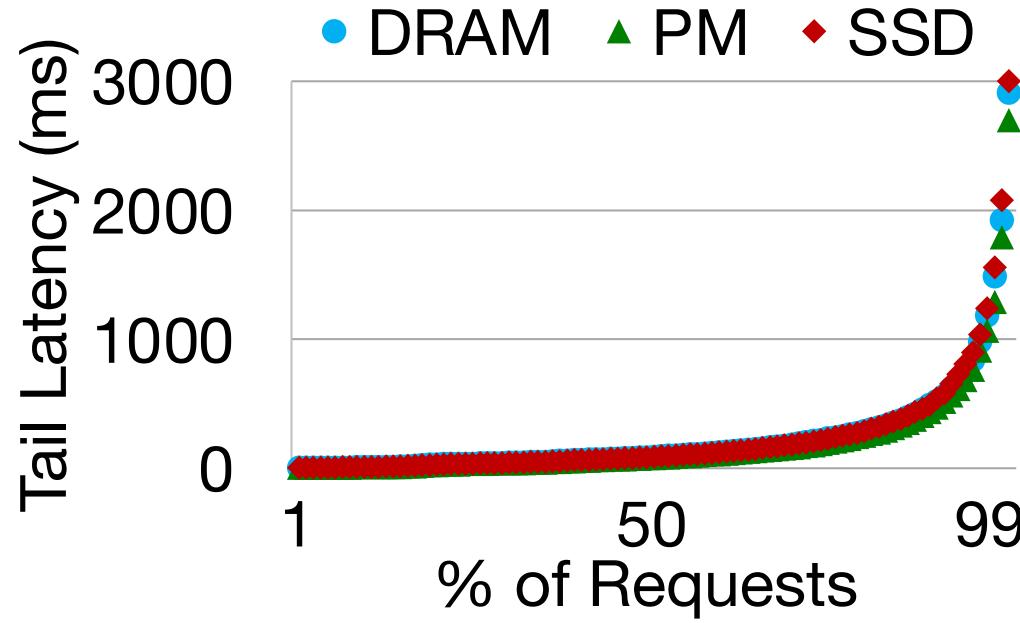
See **paper** for details

- Term selection, variation, ranking

# Tail latency of single-term queries

**DRAM = PM = SSD**

Accessing a single posting list results in a sequential access pattern

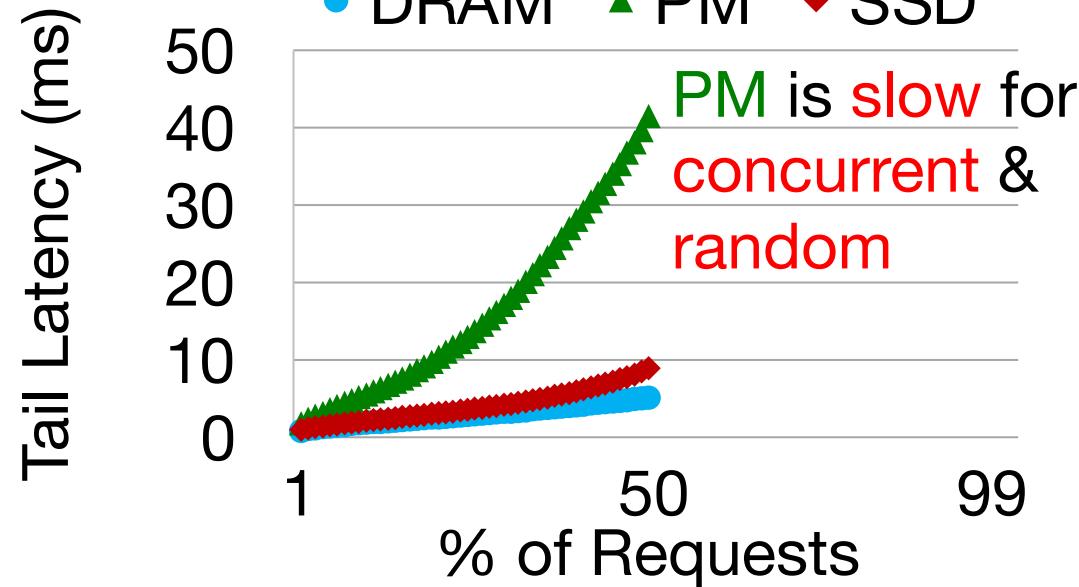


# Tail latency of 2-term AND

Region 1: DRAM < SSD < PM

50% Shortest queries

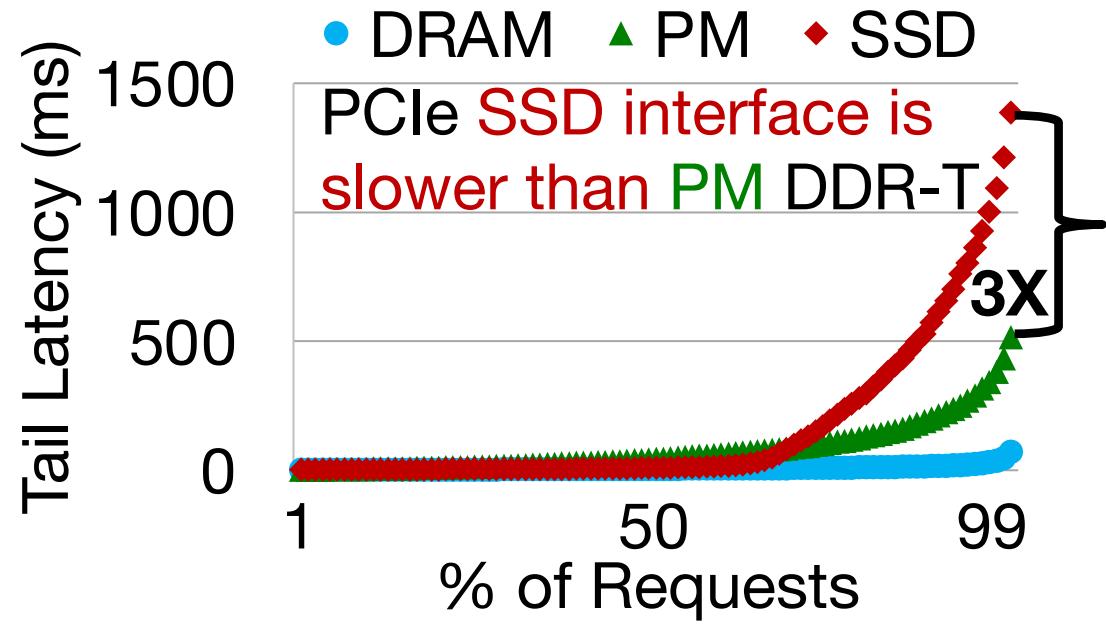
Advancing two lists  
leads to random  
accesses



# Tail latency of 2-term AND

## Region 2: DRAM < PM < SSD

50% Longest queries  
These queries access  
the SSD media



# More analysis in the paper

Indexing: updates

Query eval: access patterns

Breakdowns: sort vs merge, load vs store

pmemkv: volatile map, binding

Other: OS caching impacts

# Key Takeaways

PM do not scale well for write I/O bound indexing

PM shines for the latency-critical query evaluation

# Contribution: PM Search Engine

Exploiting PM for building/storing indices

- Memory, storage, universal roles
- Fine-grained crash consistent recovery

Extensive PM evaluation vs DRAM/SSD

- Indexing perf, scalability, bottlenecks
- Tail latency of query workloads