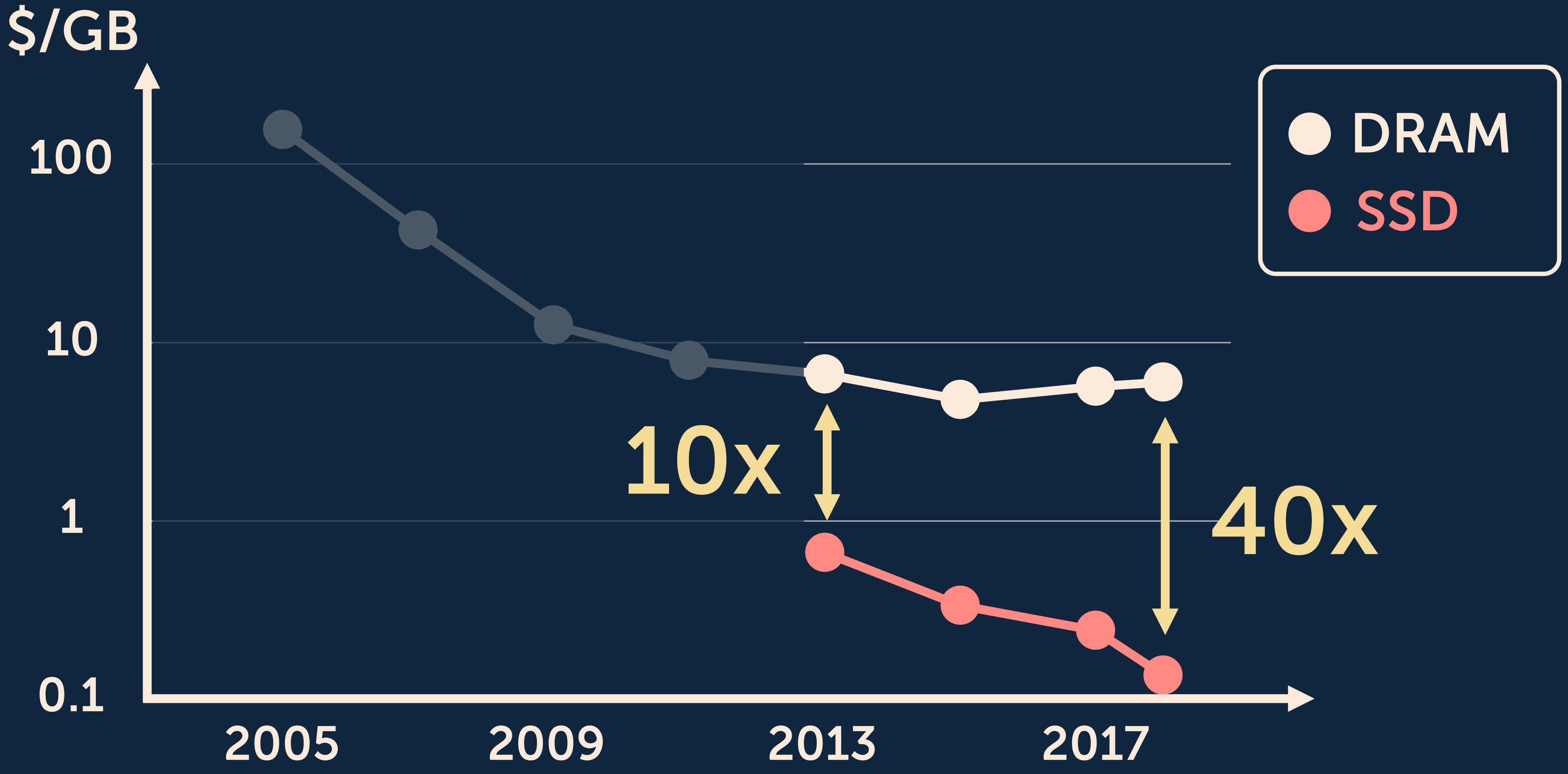
Memory-Efficient Search Trees for Database Management Systems

@huanchenzhang

Carnegie Mellon University

Memory is precious



[Source: https://www.jcmit.net/memoryprice.htm]

Databases face tight memory budgets

An Example mid-tier Amazon EC2 Instance optimized for database workloads

vCPU	Mem(GB)		SSD(GB)
4	30.5	1 : 30	950

RocksDB → 1 : 100

Modern applications demand more

Example: Alibaba’s e-commerce platform on Singles’ Day

- Average response time: < 0.5 ms
- Peak throughput: 70 million txn/s

Working set must fit in memory

Insufficient Memory → Memory-Efficiency → Higher Database Performance

With Less

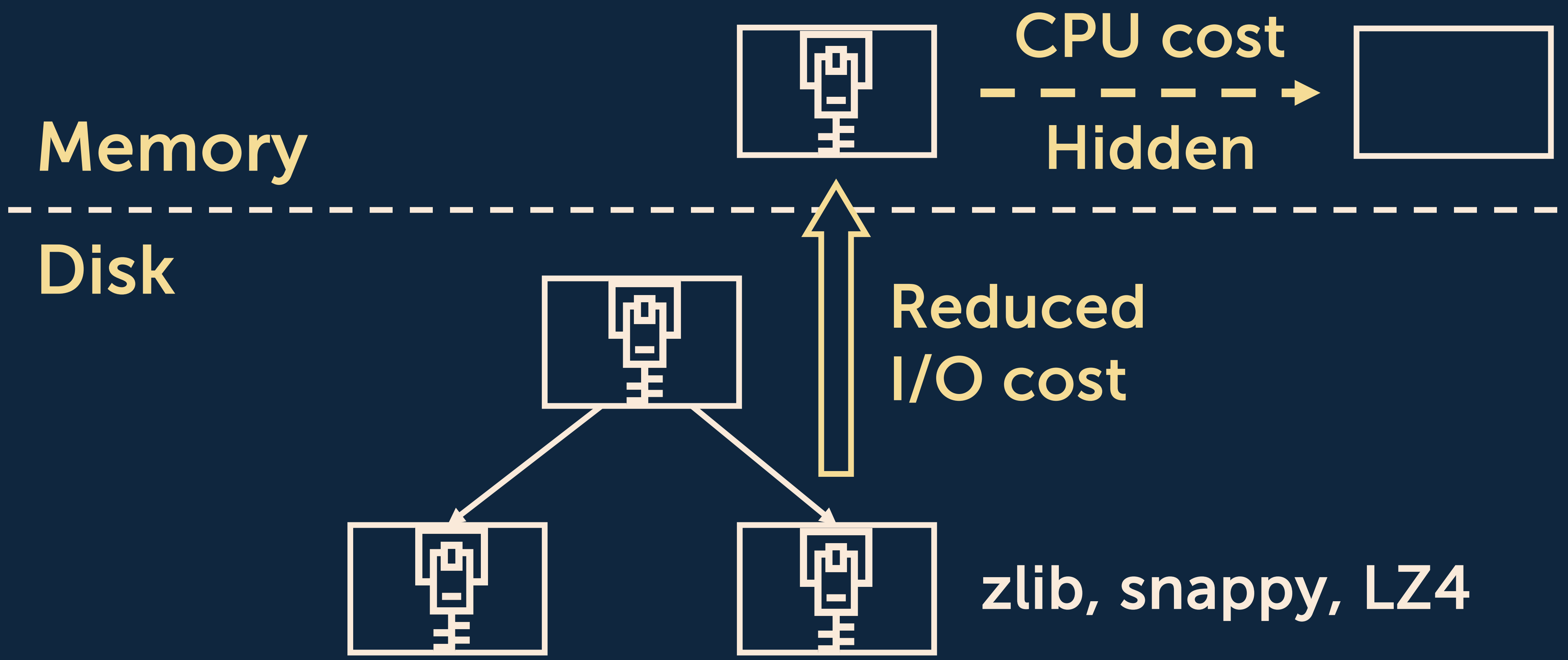
Do More

Search trees consume a lot of memory

Statistics from H-Store

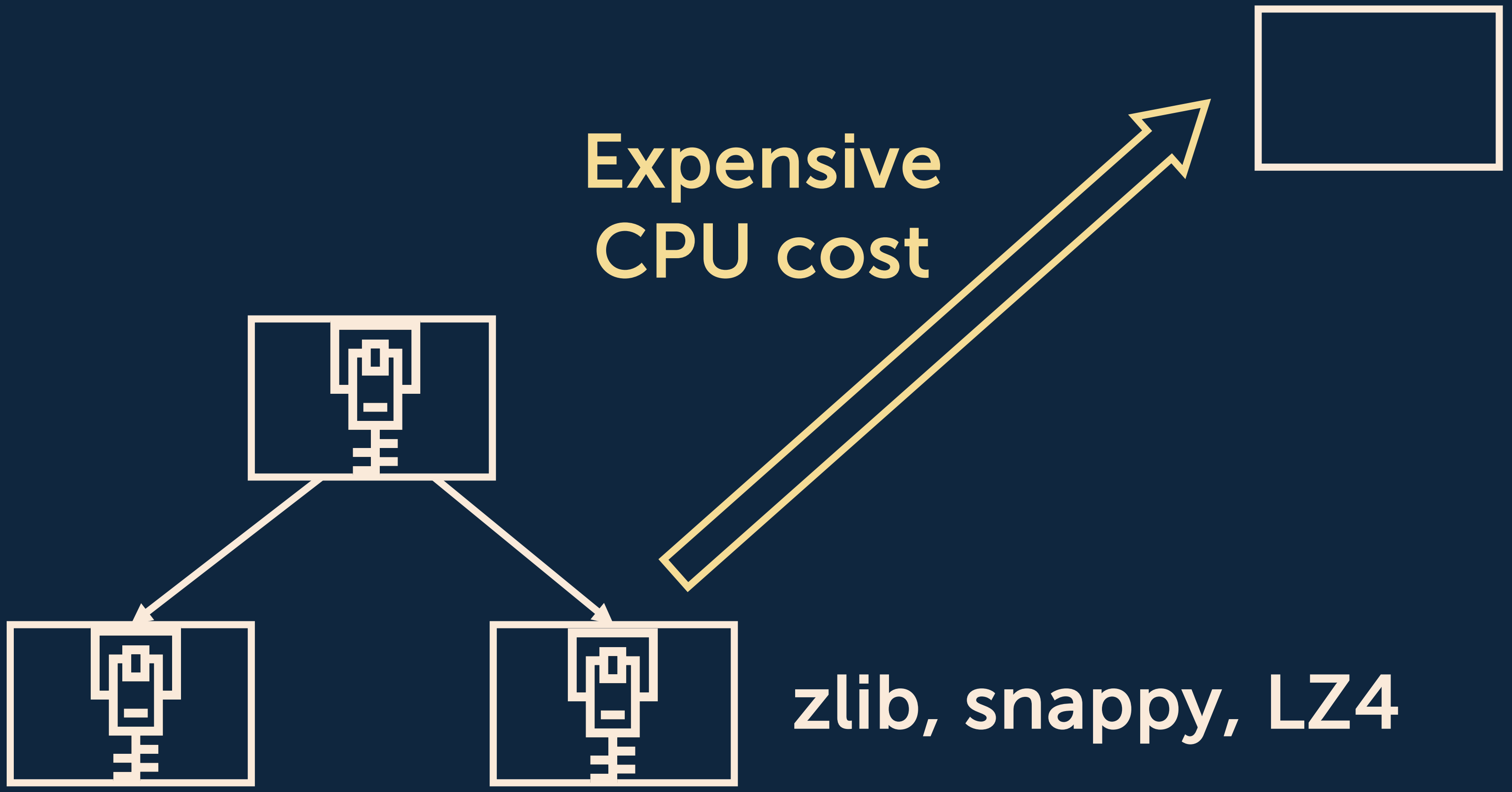
Benchmark	Tree Index Memory
TPC-C	58%
Voter	55%
Articles	34%

Block compression works well on disk

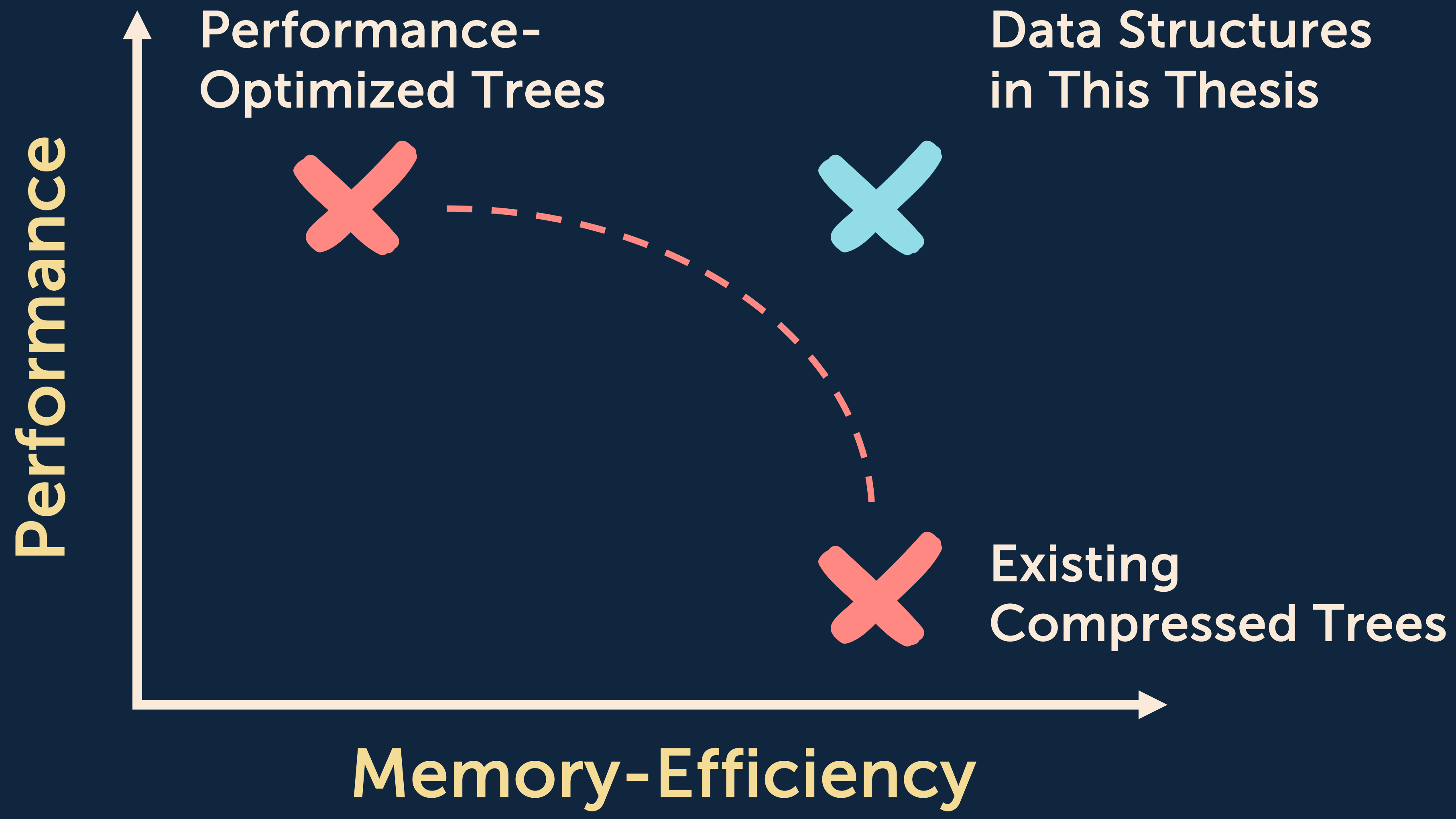


Block compression is slow in memory

Memory



Thesis goal: a Pareto improvement



Thesis Statement:

Compressing in-memory search trees via efficient algorithms and careful engineering improves the performance and resource-efficiency of database management systems.

① Build fast static search trees with maximum structural compression

Memory-Efficiency

② Support dynamic operations with bounded & amortized cost

③ Compress input keys efficiently while preserving their order

Part I

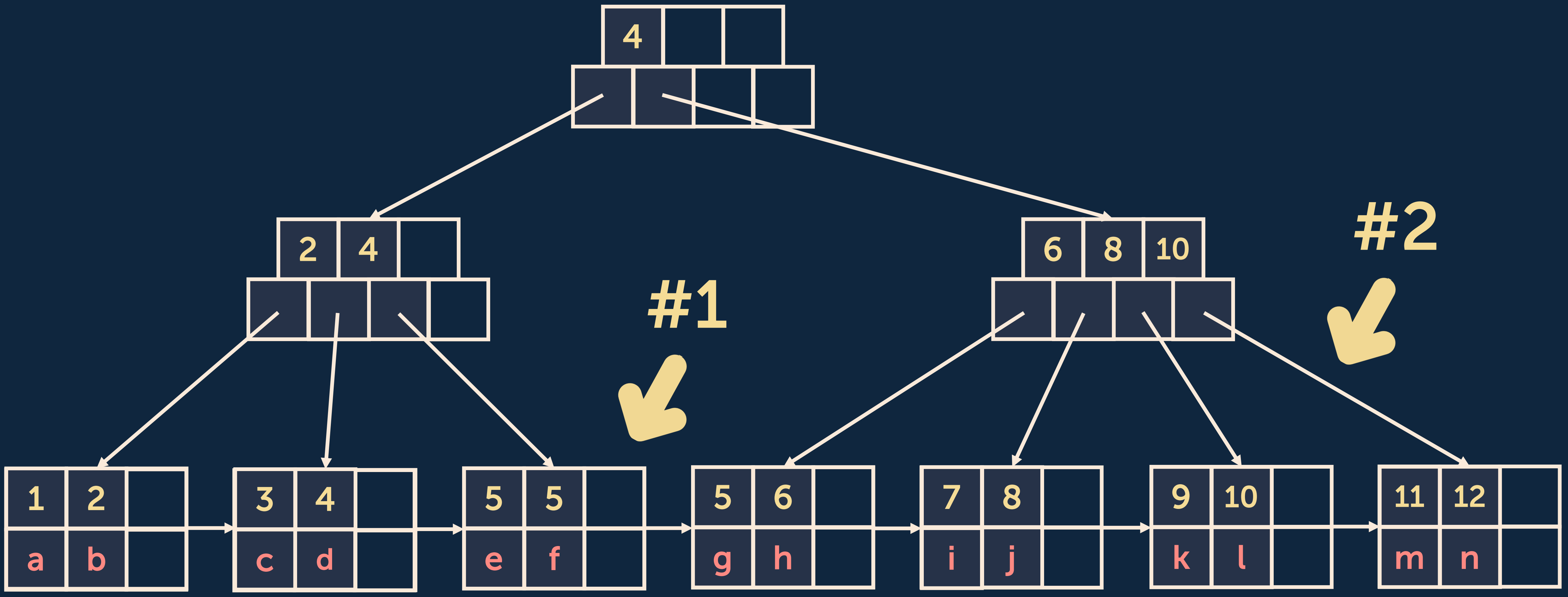
Compressing Static Search Trees

Dynamic-to-Static Rules

Fast Succinct Tries

Succinct Range Filters

Memory overhead in dynamic trees



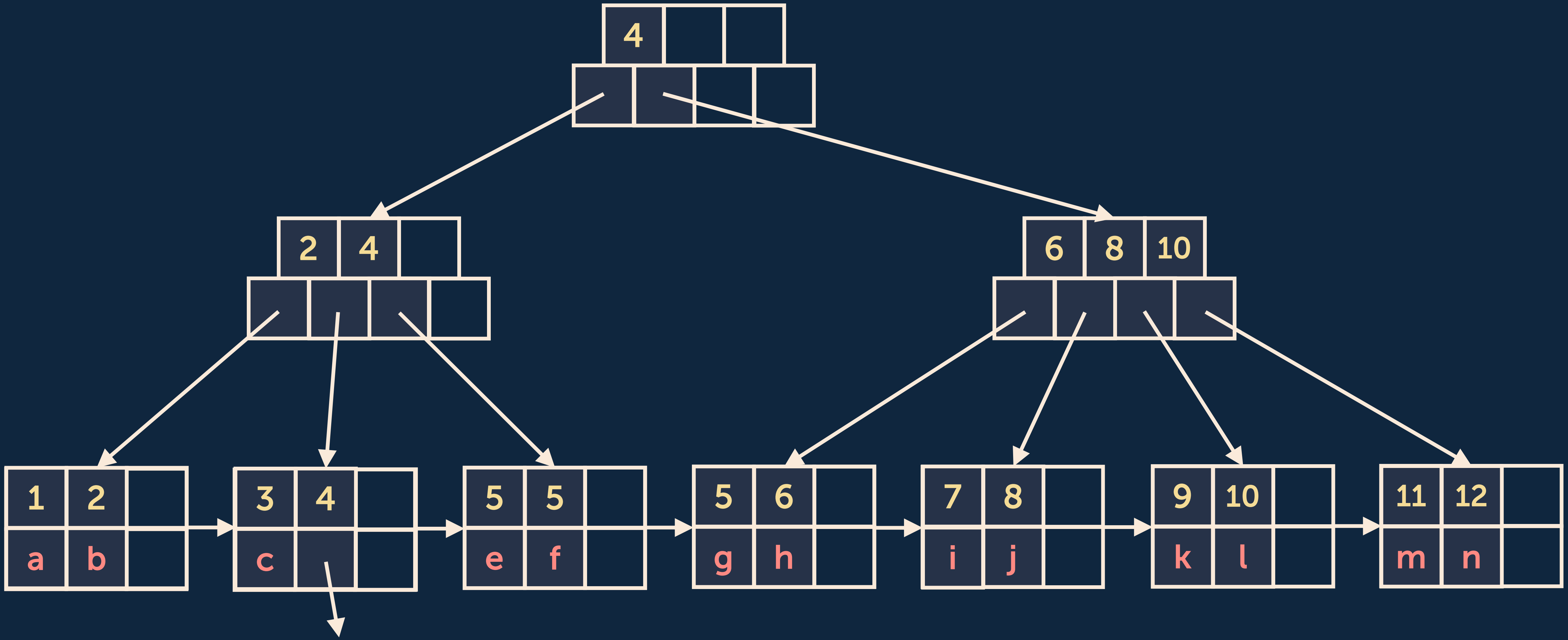
#1 Compaction:

Remove duplicate entries and make every allocated memory block 100% full.

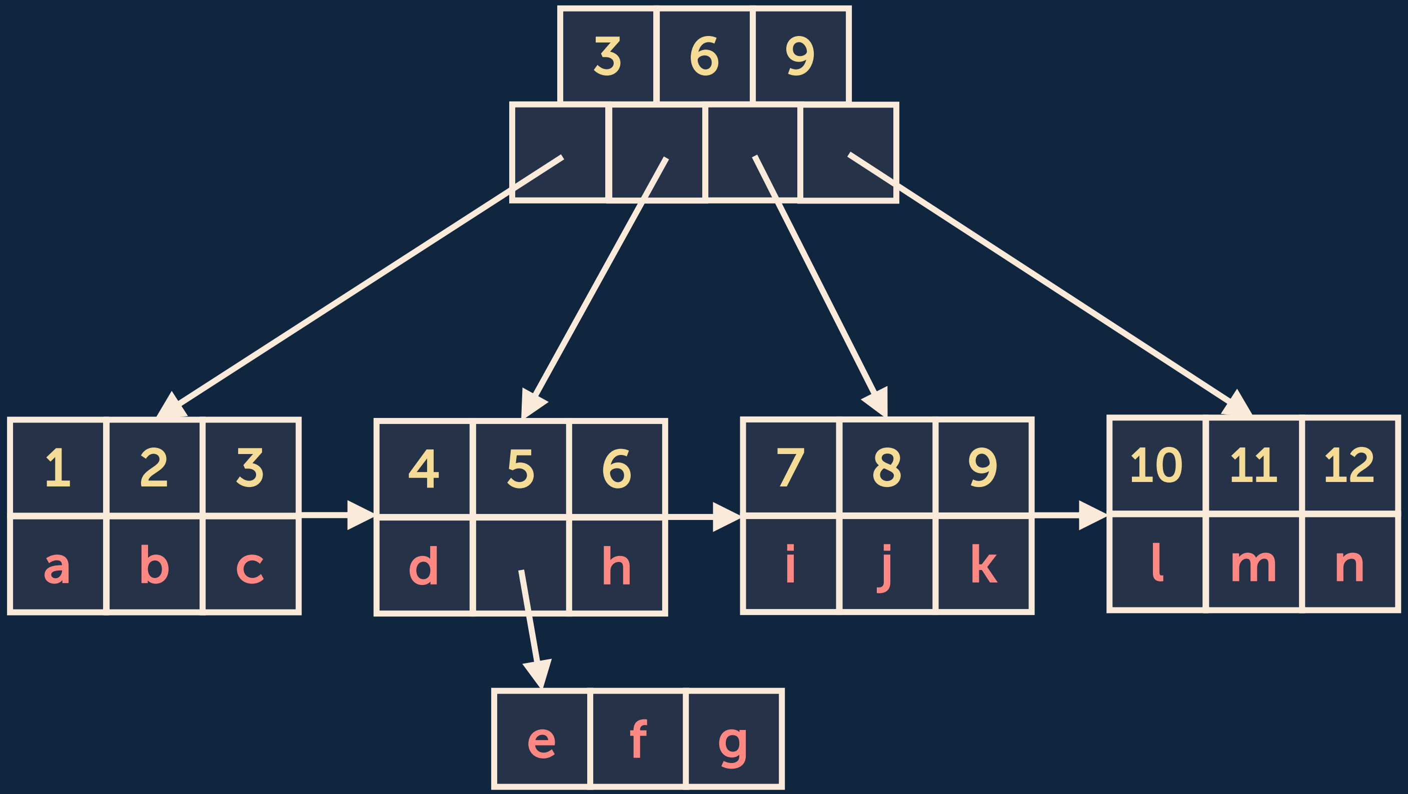
#2 Reduction:

Remove pointers and structures that are unnecessary for efficient read operations.

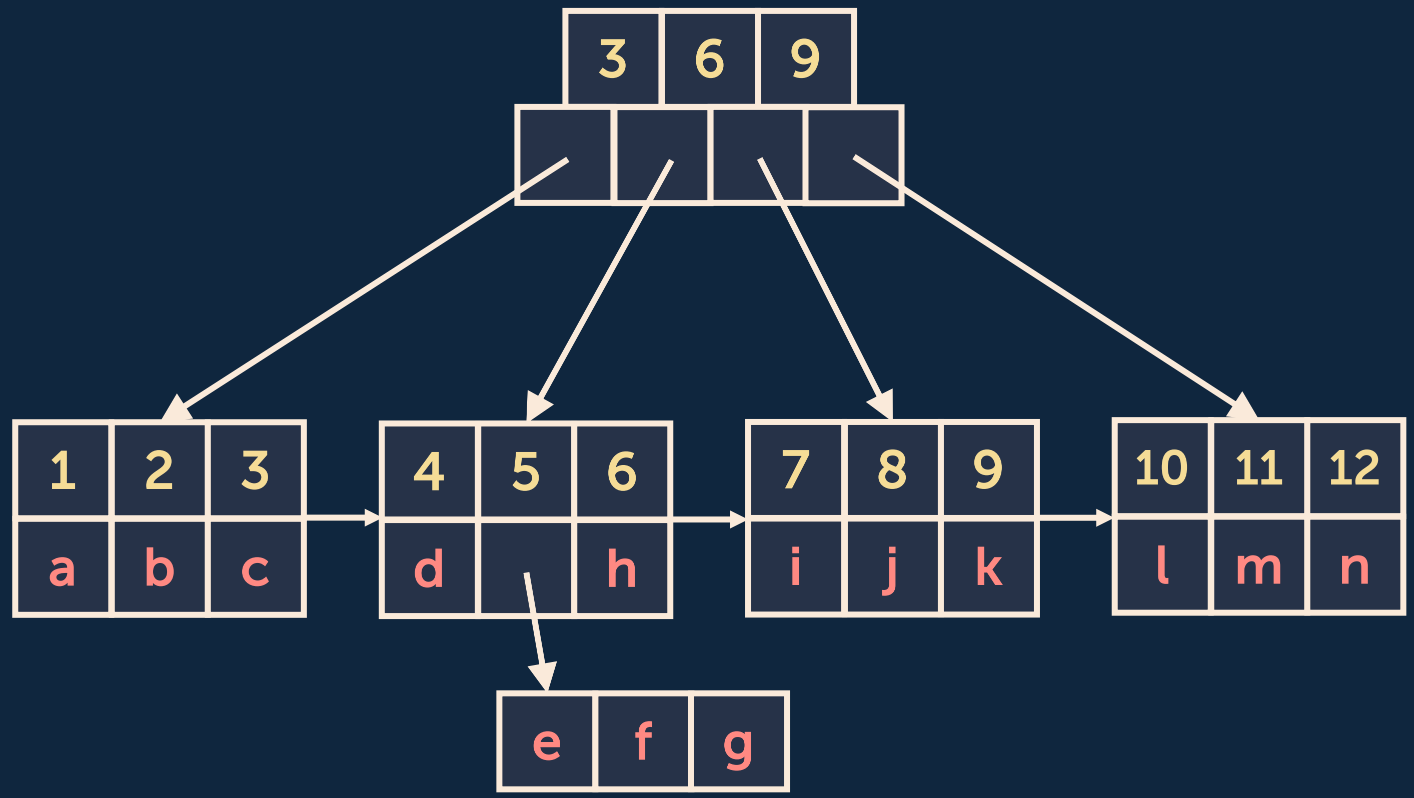
#1 Compaction on B+trees



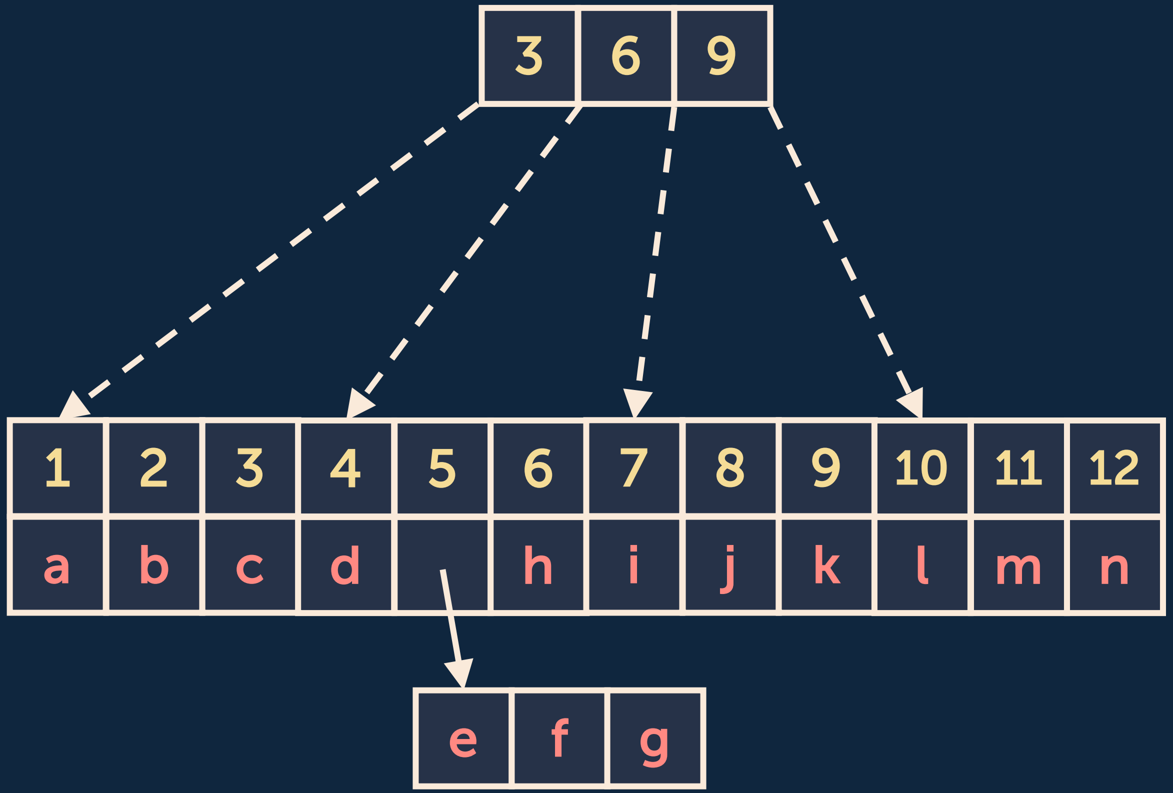
#1 Compaction on B+trees



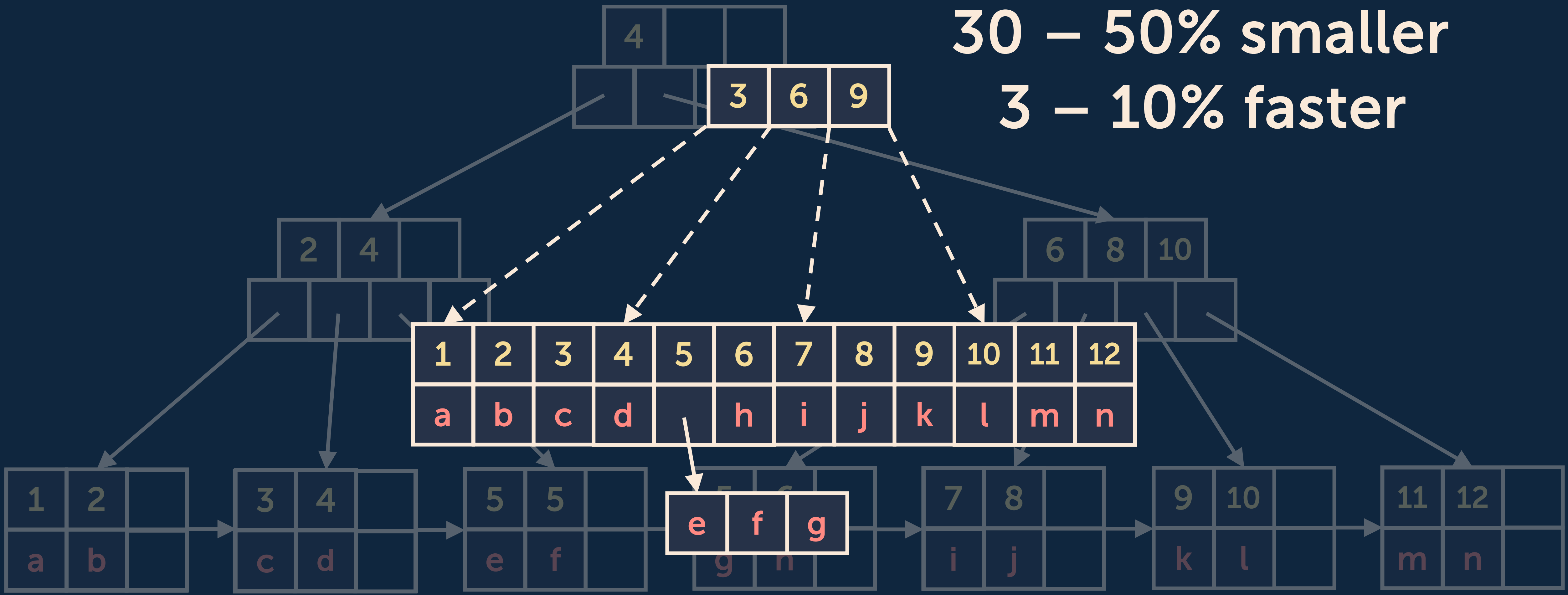
#2 Reduction on B+trees



#2 Reduction on B+trees



Compact B+tree vs. Regular B+tree



30 – 50% smaller

3 – 10% faster

Part I

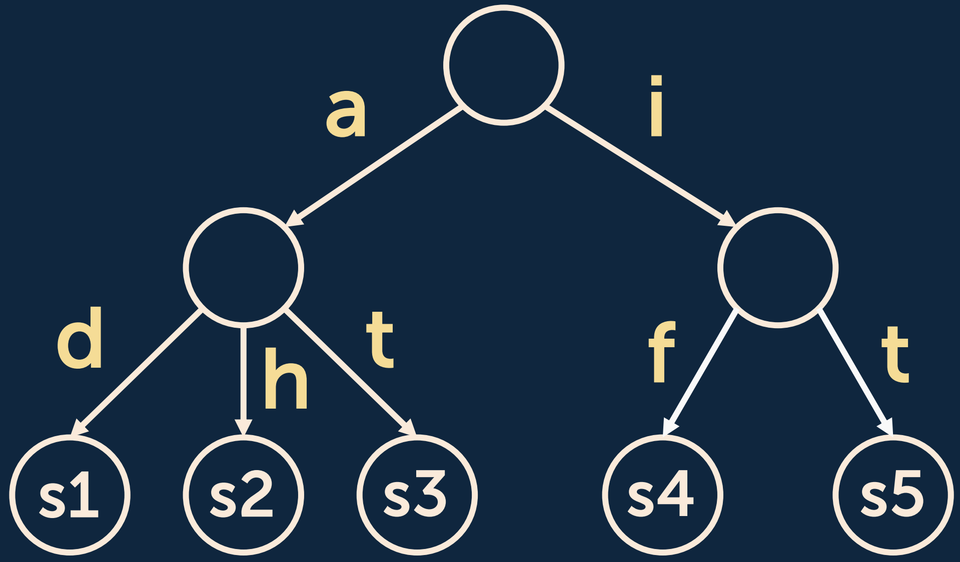
Compressing Static Search Trees

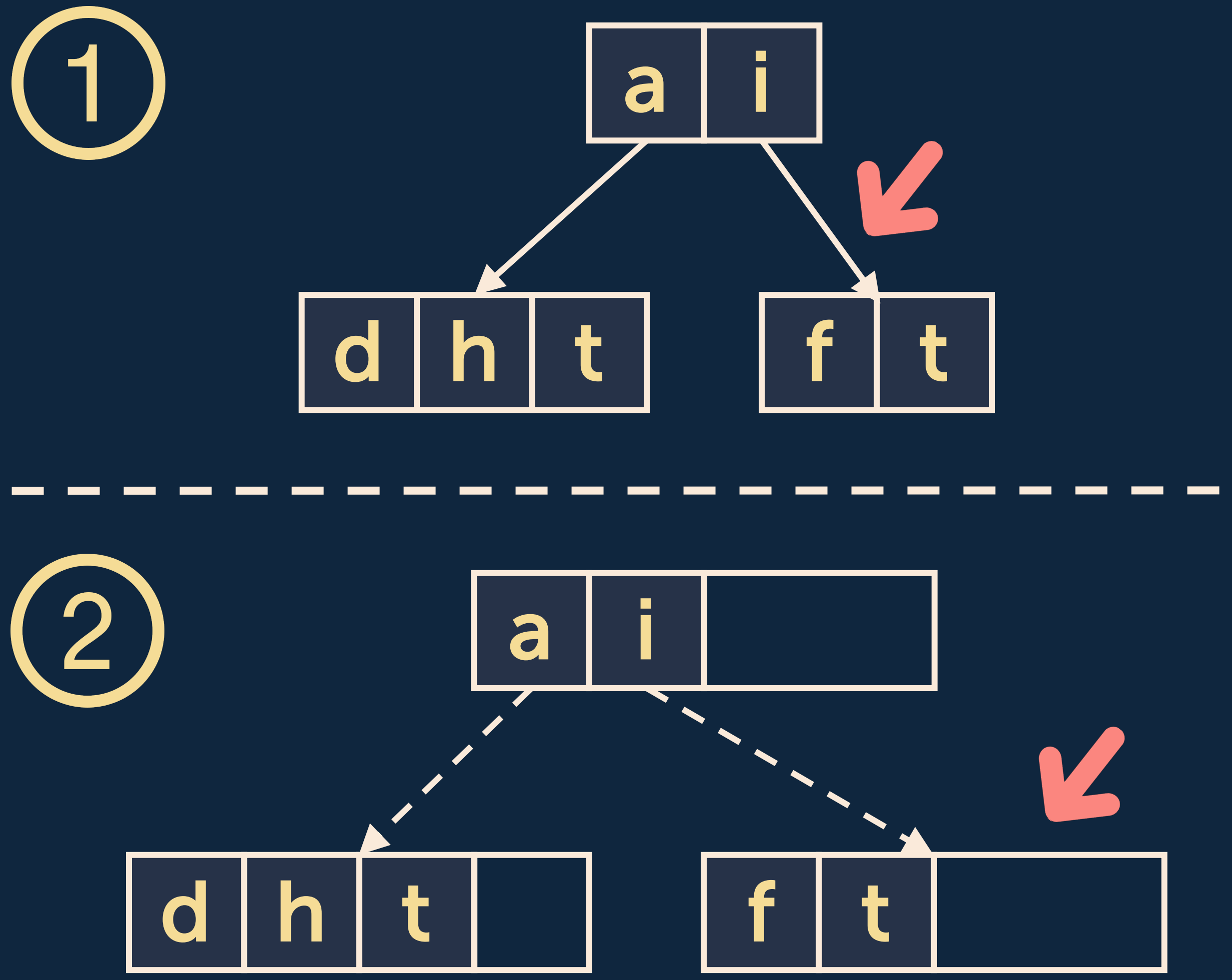
Dynamic-to-Static Rules

Fast Succinct Tries

Succinct Range Filters

Challenges in compressing tries





The information-theoretic lower bound

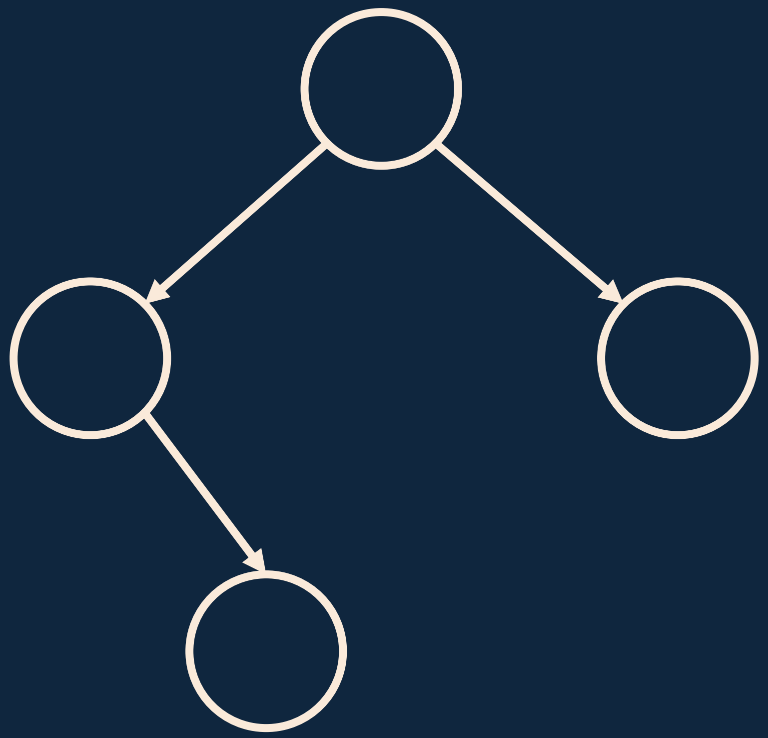
The minimum number of bits needed to distinguish any object in a class

$$|S| = n \rightarrow \log_2 n \text{ bits}$$

$$\left| \begin{array}{l} n\text{-node tries} \\ \text{of degree } \underset{\downarrow}{k} \end{array} \right| = \frac{\binom{kn+1}{n}}{kn+1} \rightarrow \underbrace{n(\log_2 k + \log_2 e)}_{9.4n} \text{ bits}$$

256

Warm-up: succinctly encode a binary tree

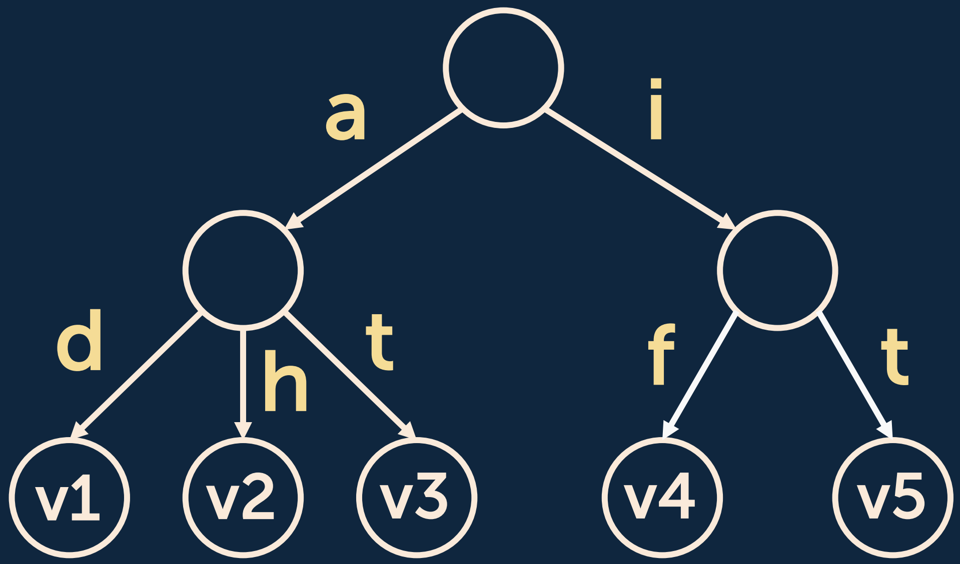


Level-Order Encoding

1 1 0 1 0 0 0 0

2n bits

Our succinct trie representation



Label:	a	i	d	h	t	f	t
Has-Child:	1	1	0	0	0	0	0
Structure:	1	0	1	0	0	1	0
~~Value:~~			~~v1~~	~~v2~~	~~v3~~	~~v4~~	~~v5~~

Limit = 9.4 bits/node

10 bits/node

Rank & Select on bit vectors

rank(bv, i) = # 1's up to position i in bv
select(bv, i) = position of the i-th 1 in bv

0 5 10

bv: 1 1 0 1 1 1 0 1 1 0 0 0 0 0 0

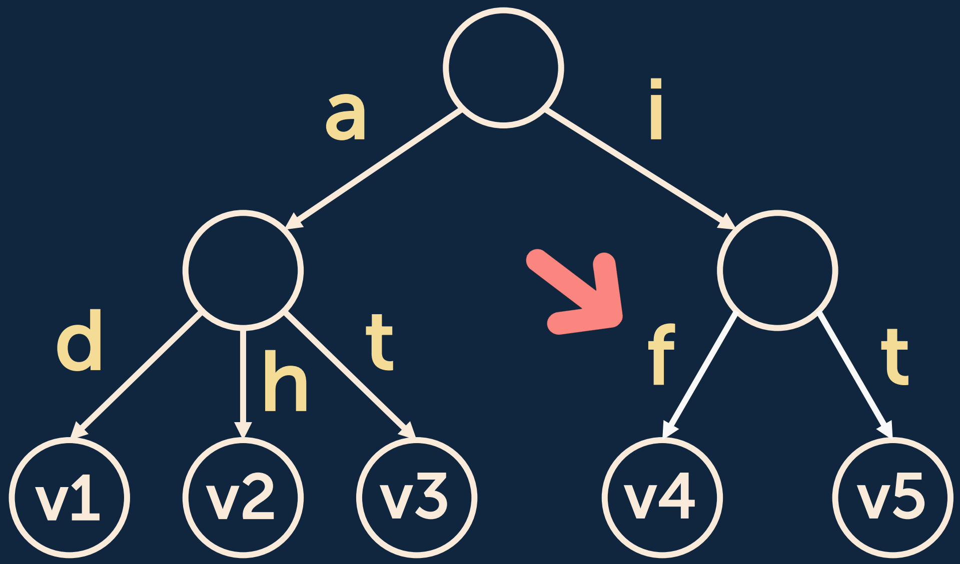
rank(bv, 6) = 5

select(bv, 6) = 7

Time: Constant

Space: 3-10% sizeof(bv)

Search the encoded trie efficiently

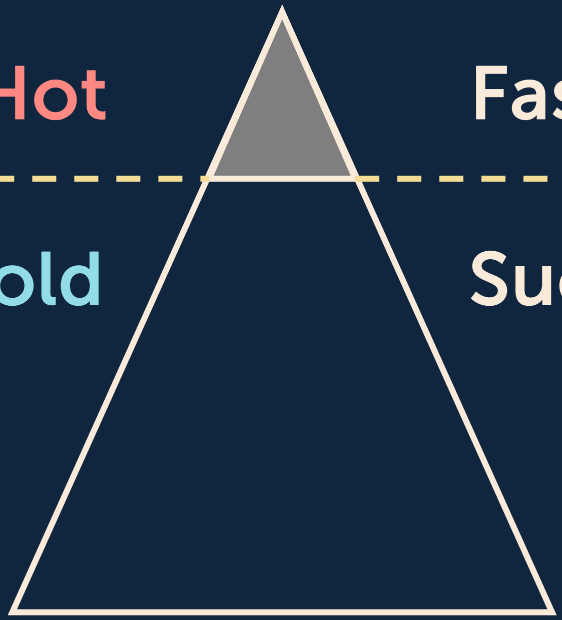


Label:	a	i	d	h	t	$\boxed{f}$	t
Has-Child:	1	1	0	0	0	0	0
Structure:	1	0	1	0	0	1	0
Value:			v1	v2	v3	v4	v5

moveToChild(p) = select(Structure, rank(Has-child, p) + 1)

moveToValue(p) = p - rank(Has-child, p)

Trade small space for performance



Hot — Fast Bitmaps — 1

Cold — Succinct Encoding — 64

Size Ratio

< 1% space overhead

3x speed-up

More performance optimizations

1. Vectorized Label Search

a b c d e f g h i j k l m n o p — SIMD Search

2. Memory Prefetching

Label:	a	i	d	h	t	f	t
Has-Child:	1	1	0	0	0	0	0
Structure:	1	0	1	0	0	1	0

30% speed-up

3. • • •

The Fast Succinct Trie (FST)

Small

≈10 bits per key for 64-bit integers

≈14 bits per key for emails

Fast

≈200 ns per query for 10M 64-bit integers

= state-of-the-art performance-optimized trees

Part I

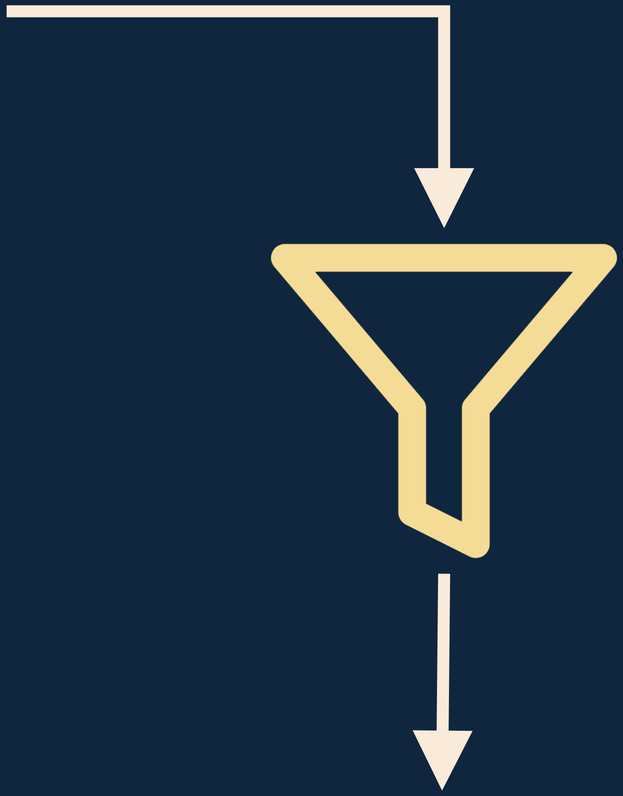
Compressing Static Search Trees

Dynamic-to-Static Rules

Fast Succinct Tries

Succinct Range Filters

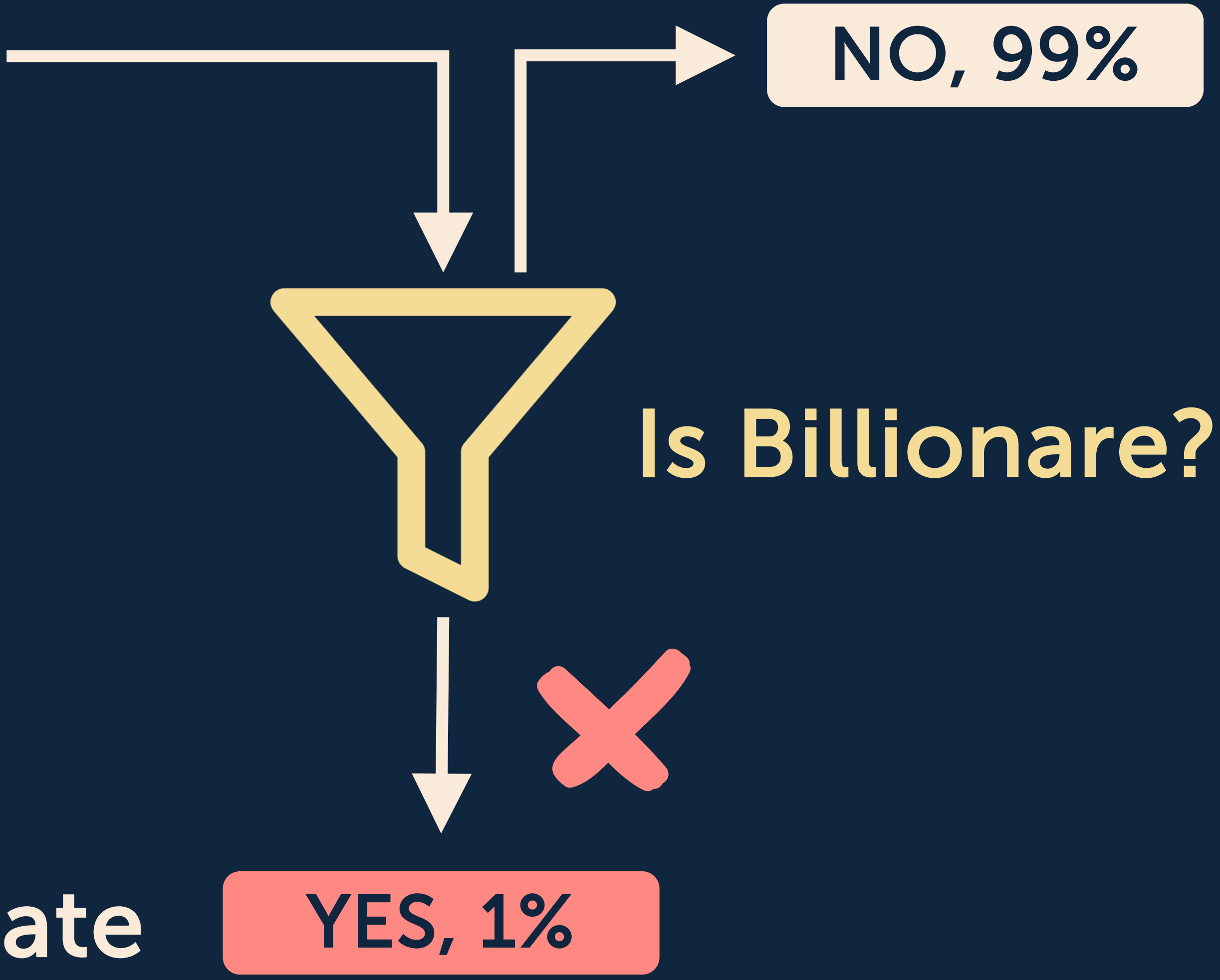
Filters answer approximate membership queries



Is Billionare?

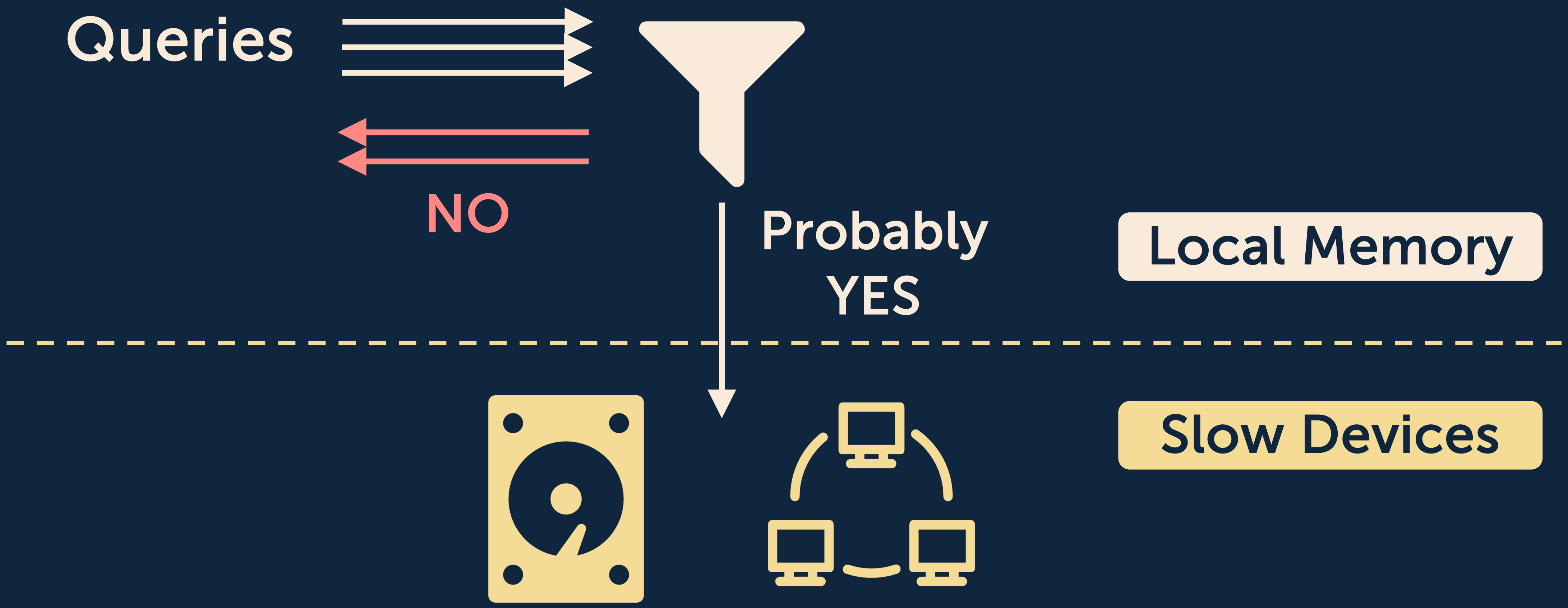
YES, 100%

Filters answer approximate membership queries



False Positive Rate

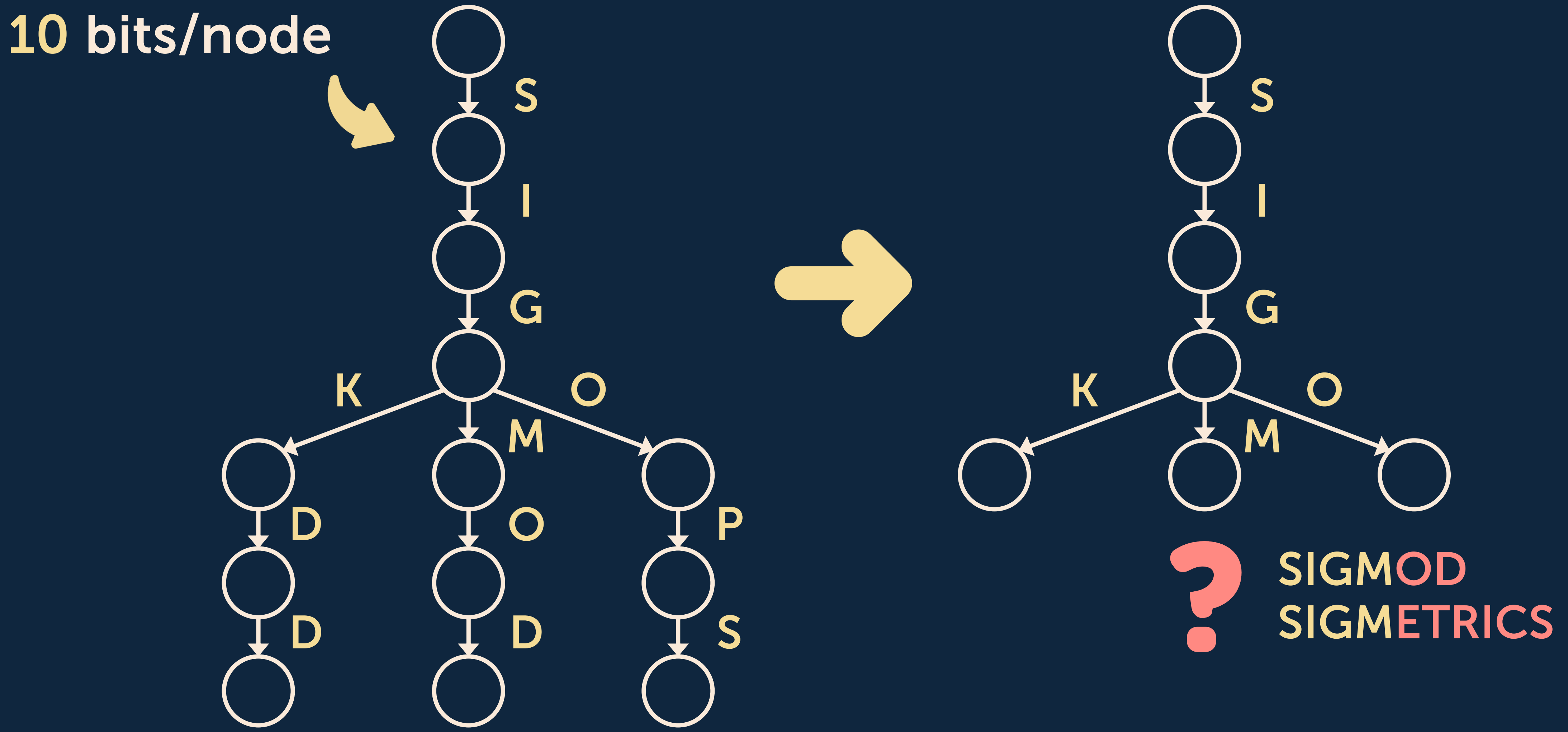
Filters help reduce unnecessary I/Os



Existing filters only support point queries

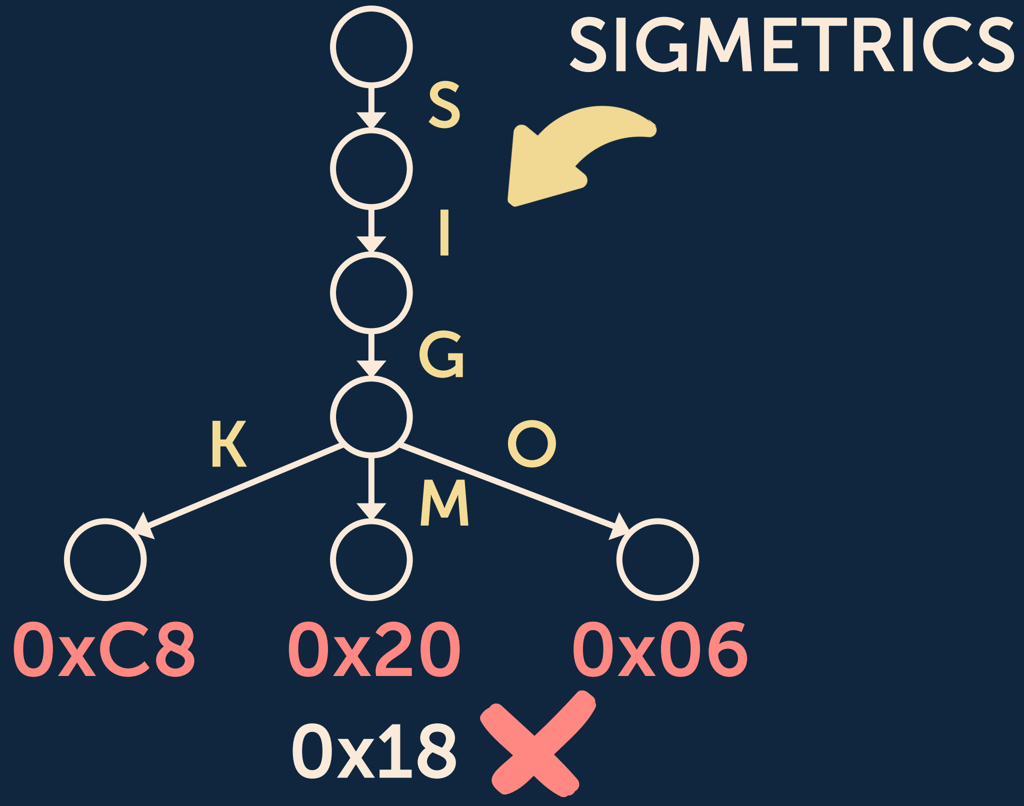
Point Filtering	Range Filtering
Is key 65 in my set?	Are there keys between 60 and 66 in my set?
Bloom Filter (1970) Quotient Filter (2012) Cuckoo Filter (2014)	?

SuRF uses a truncated trie

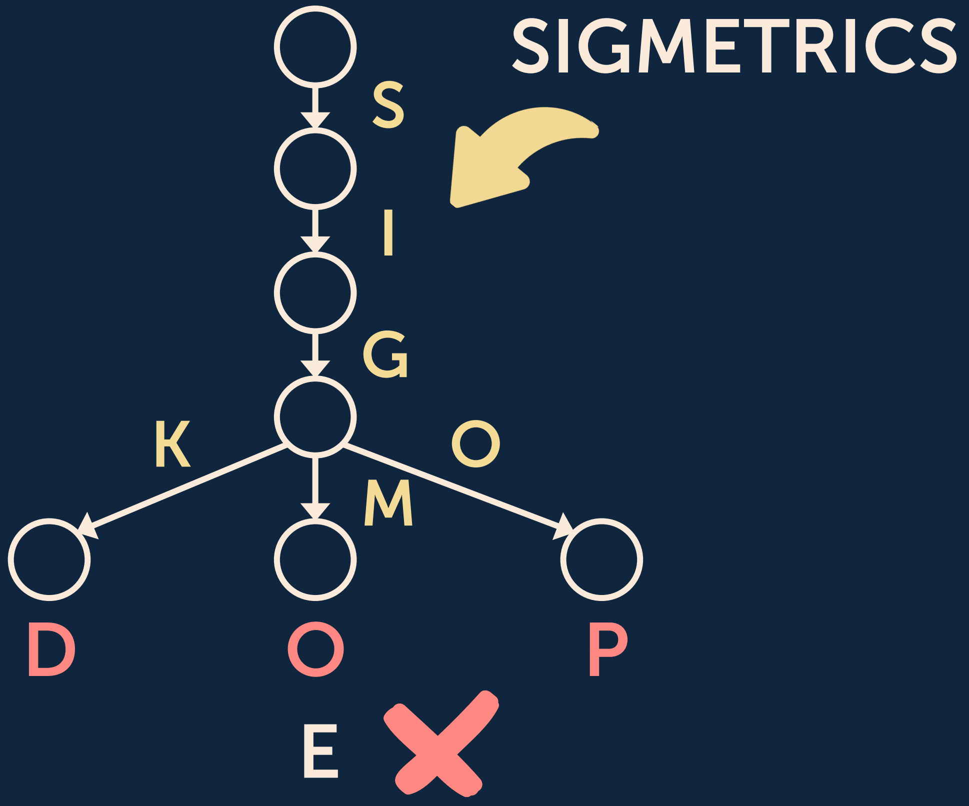


Add suffix bits to reduce false positive rate

Hashed Suffix Bits

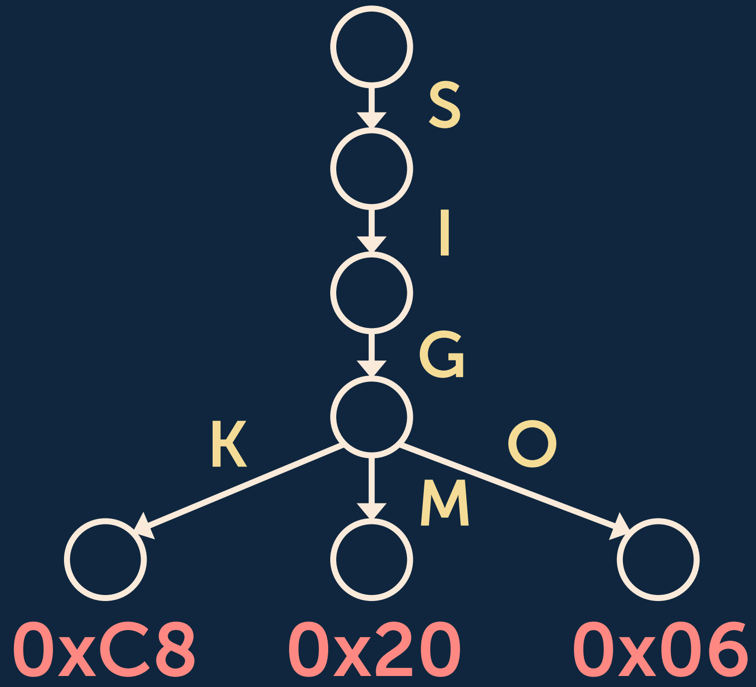


Real Suffix Bits



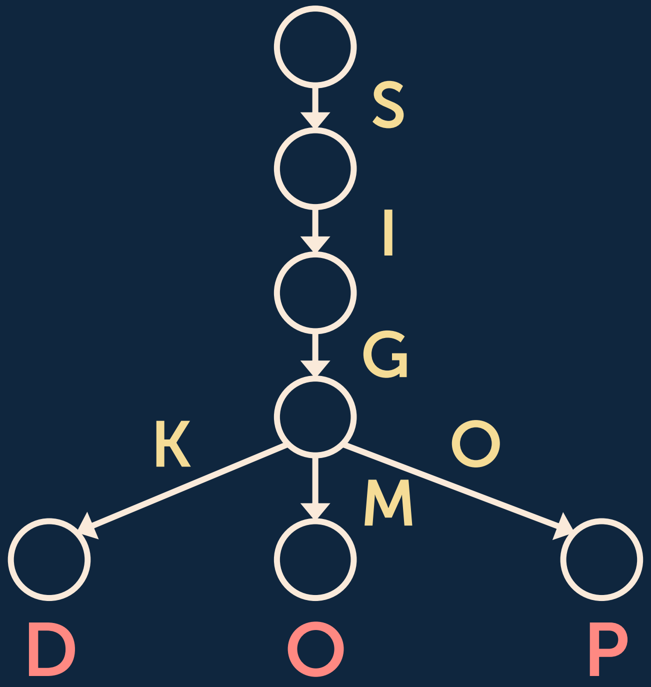
Add suffix bits to reduce false positive rate

Hashed Suffix Bits



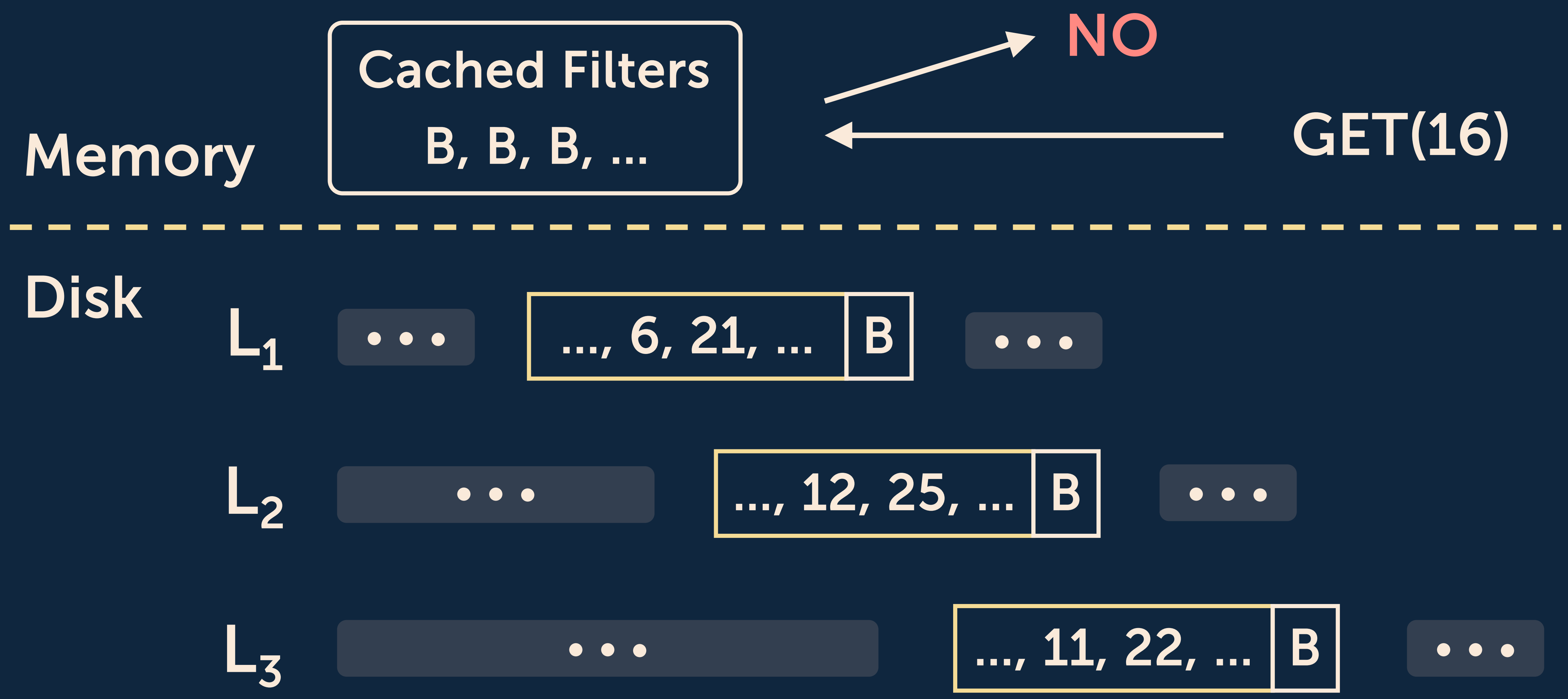
- Each bit reduces FPR by half
- Cannot help range queries

Real Suffix Bits

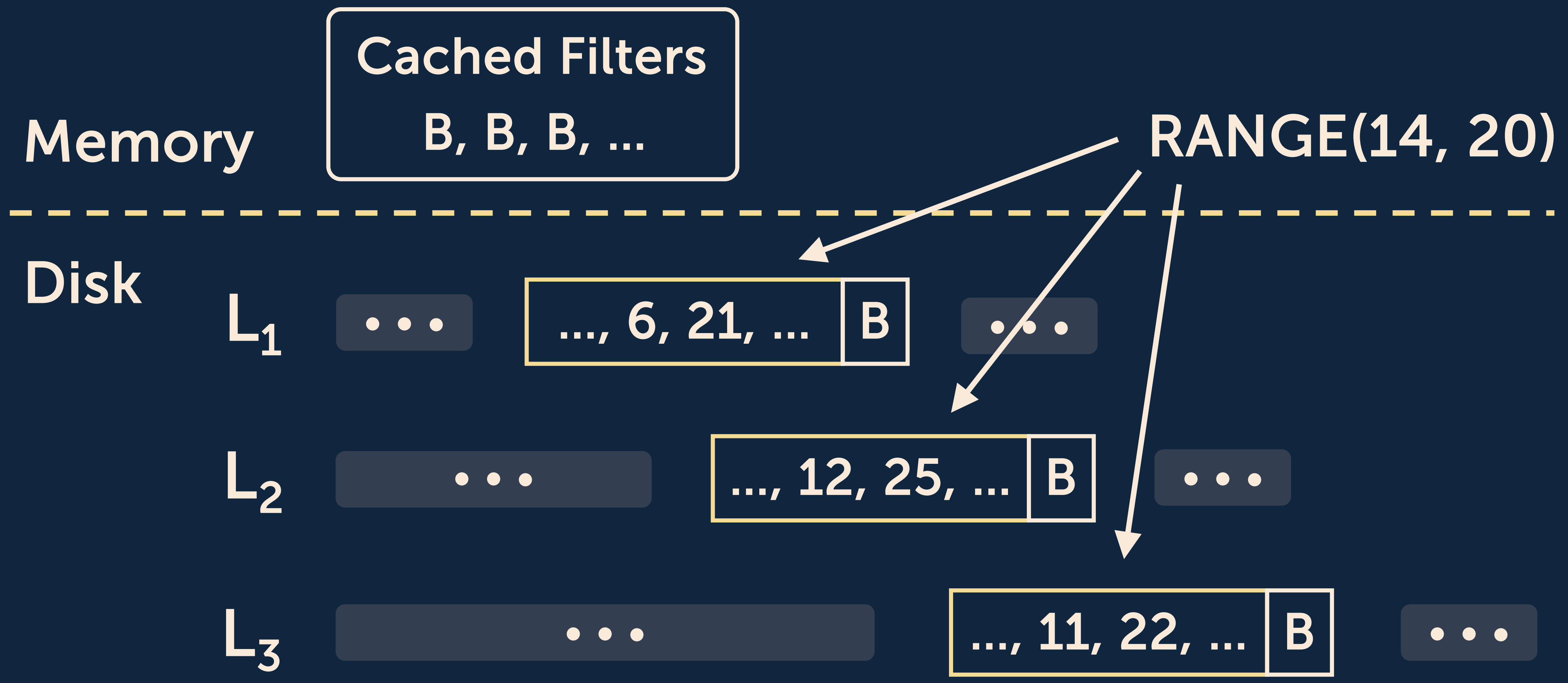


- Benefit point & range queries
- Weaker distinguishability

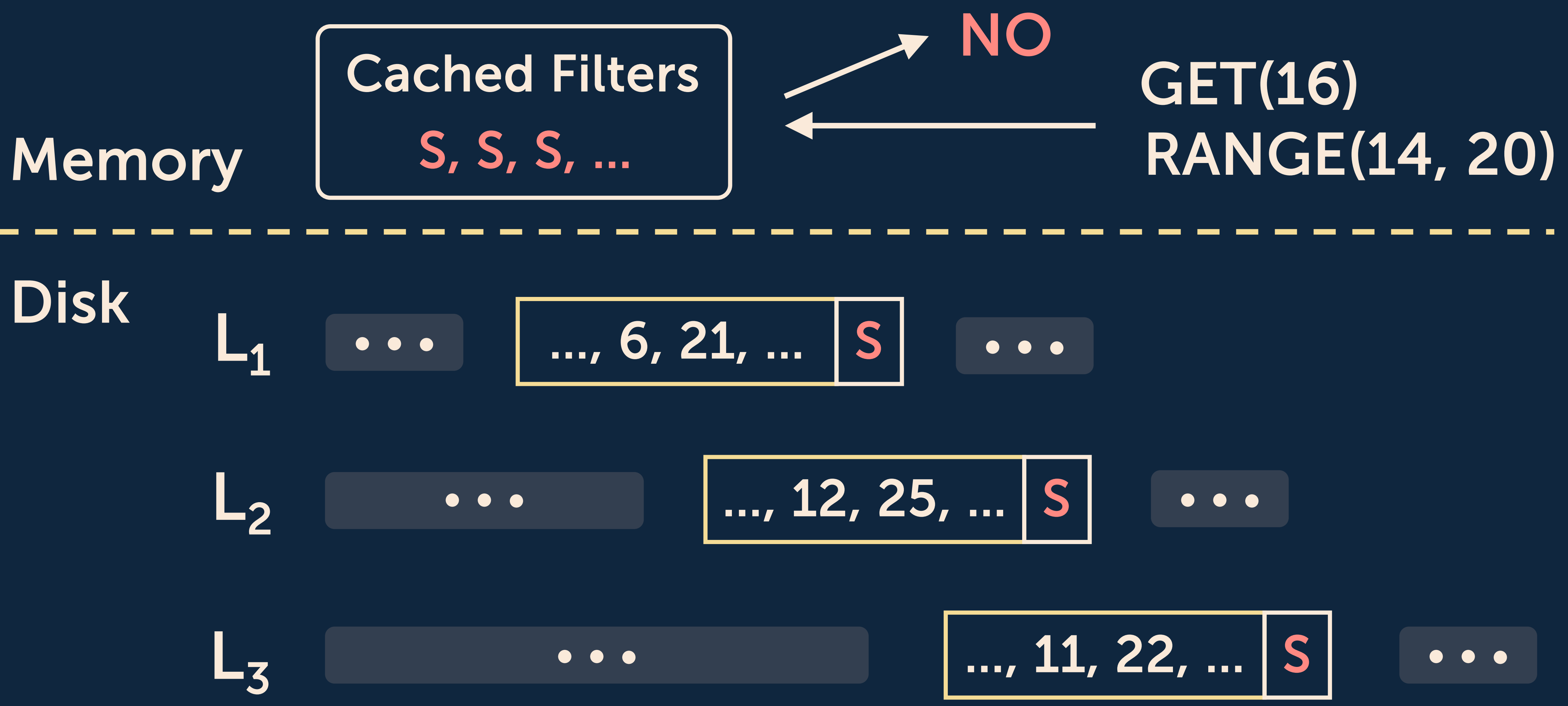
Bloom filters speed up point queries in RocksDB



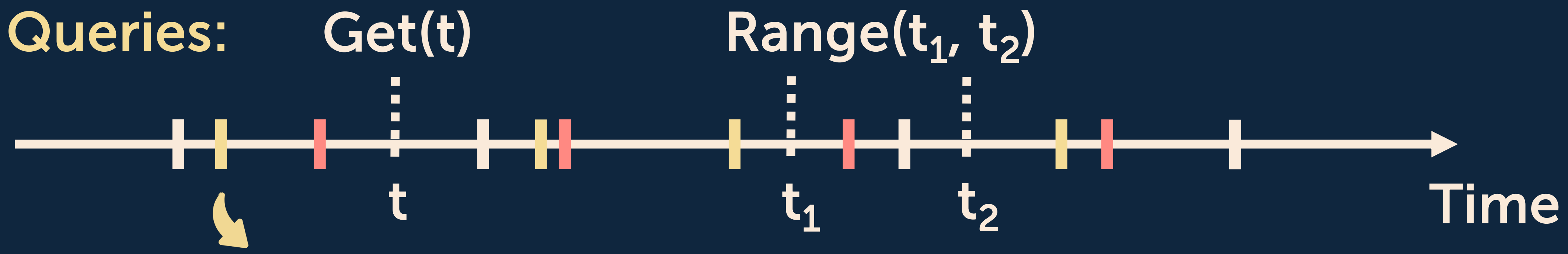
Range queries still incur multiple I/Os



SuRFs save I/Os for both point and range queries



Evaluation setup: a time-series benchmark



Key: 64-bit timestamp + 64-bit sensor ID
Value: 1KB payload

System Config

Dataset: ≈100 GB on SSD

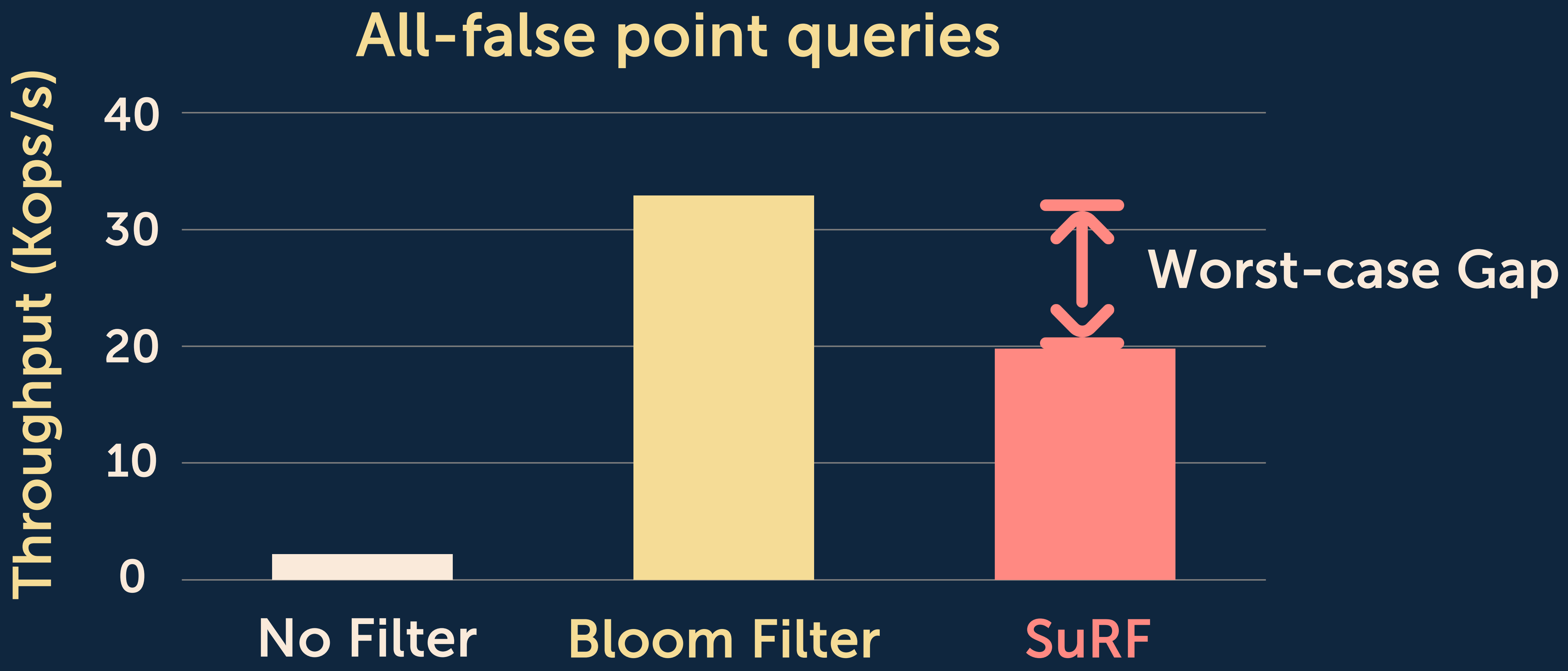
DRAM: 32 GB

Filter Config

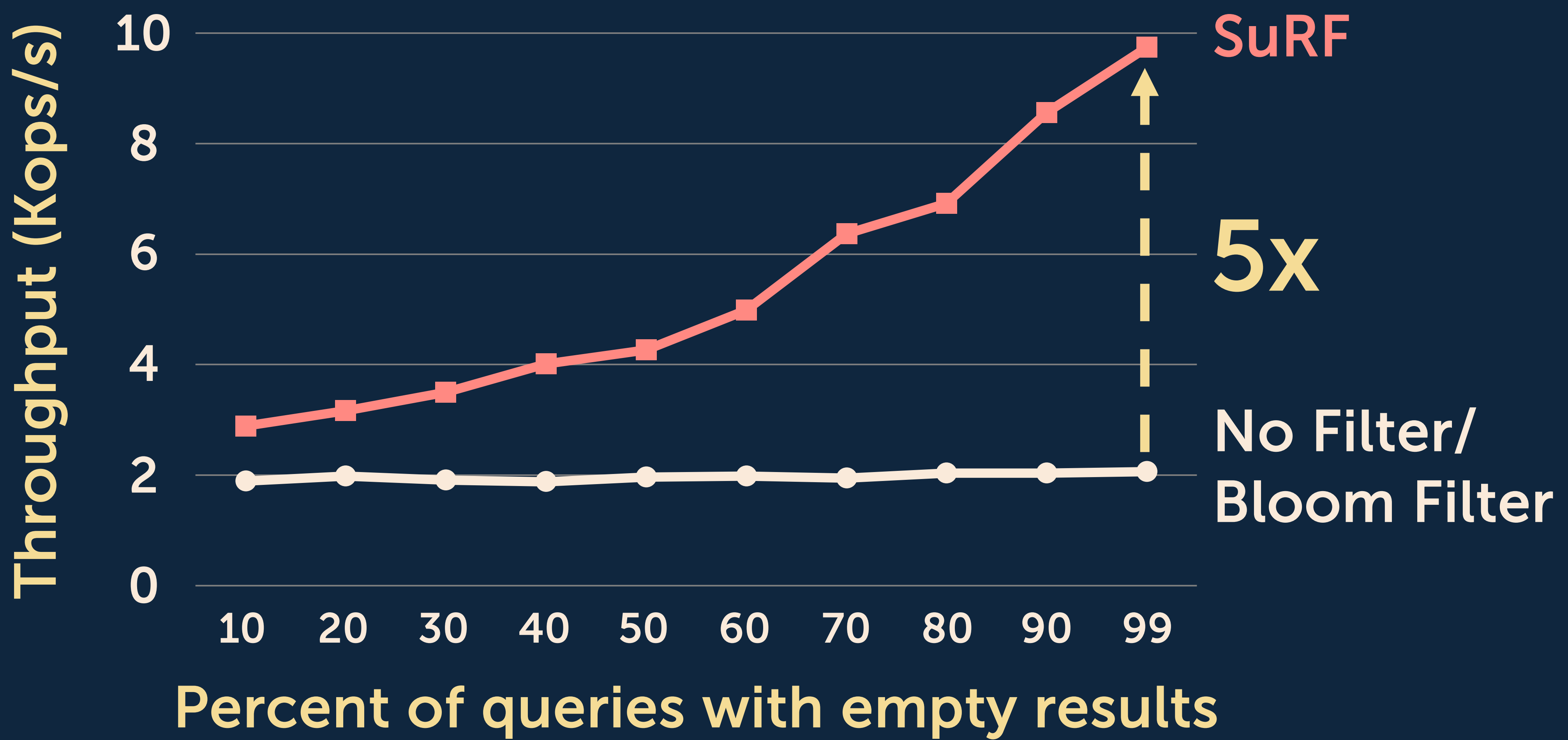
Bloom filter: 14 bits per key

SuRF: 4-bit real suffix

SuRFs act like Bloom filters for point queries



SuRFs speed up range queries



SuRF's impact in academia and industry

Best Paper Award at SIGMOD'18

Being implemented by several major internet companies

6x

1. Build fast static search trees with maximum structural compression

Memory-Efficiency

2. Support dynamic operations with bounded & amortized cost

3. Compress input keys efficiently while preserving their order

Part II

Supporting Dynamic Operations

Hybrid Index

Hybrid Index is a dual stage architecture

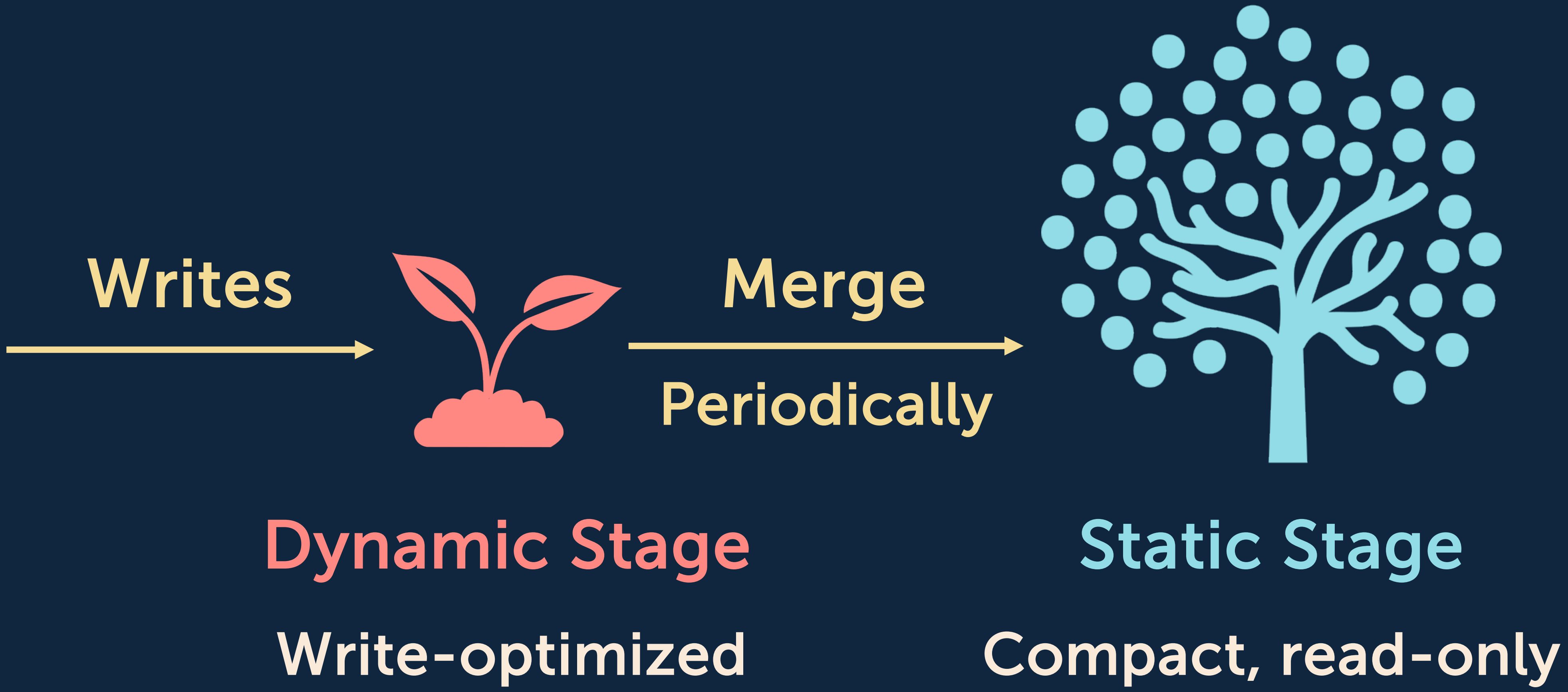
Dynamic Stage

Write-optimized

Static Stage

Compact, read-only

Inserts are batched in the dynamic stage



Reads search both stages in order

1 Read

Dynamic Stage

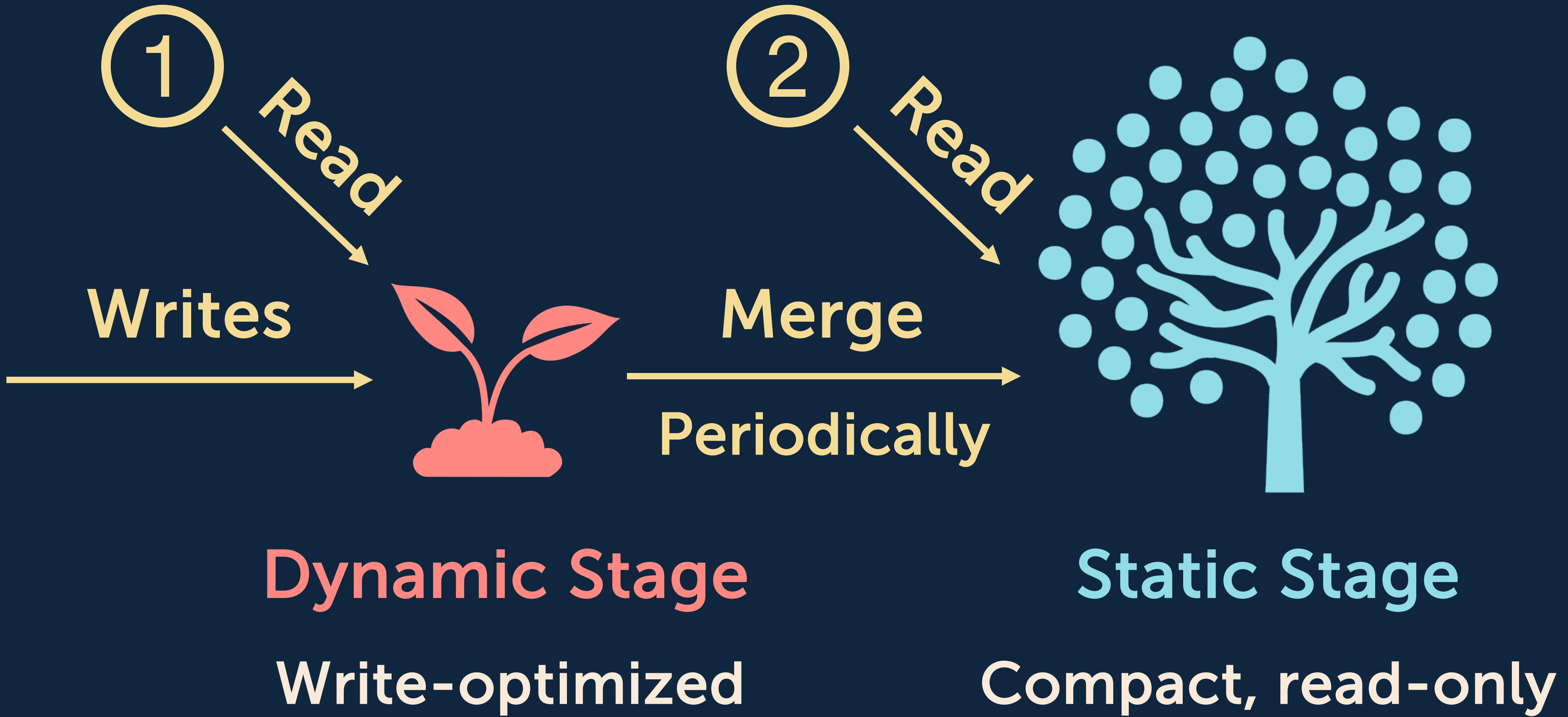
Write-optimized

2 Read

Static Stage

Compact, read-only

Hybrid Index is memory-efficient and skew-aware

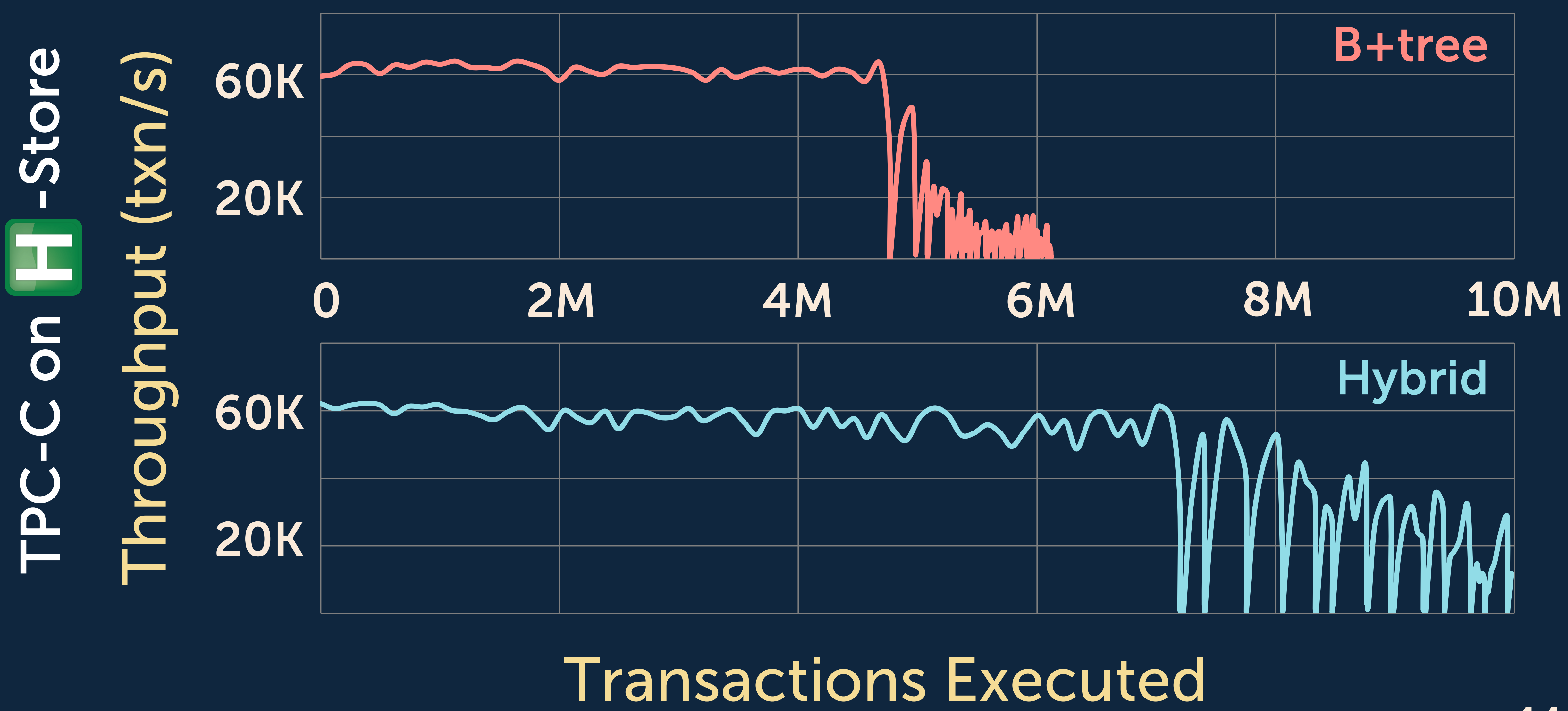


Hybrid Indexes help reduce index memory

Statistics from H-Store

Benchmark	% Memory by Original indexes	% Memory by Hybrid indexes
TPC-C	58%	34%
Voter	55%	39%
Articles	34%	18%

Hybrid Indexes improve the database's capacity



Part III

Compressing Input Keys

High-speed Order-Preserving Encoder

Existing string compression algorithms

① Whole-key Dictionary Compression

Dictionary

Dave	1
Michael	2
Andy	3
Kim	?

Table Column

Dave	⇨	1
Michael		2
Andy		3
Andy		2
Dave		3

Index

- No Compression
- New keys ?

② Huffman Compression

- Does not preserve key ordering

Compression Model: The String Axis

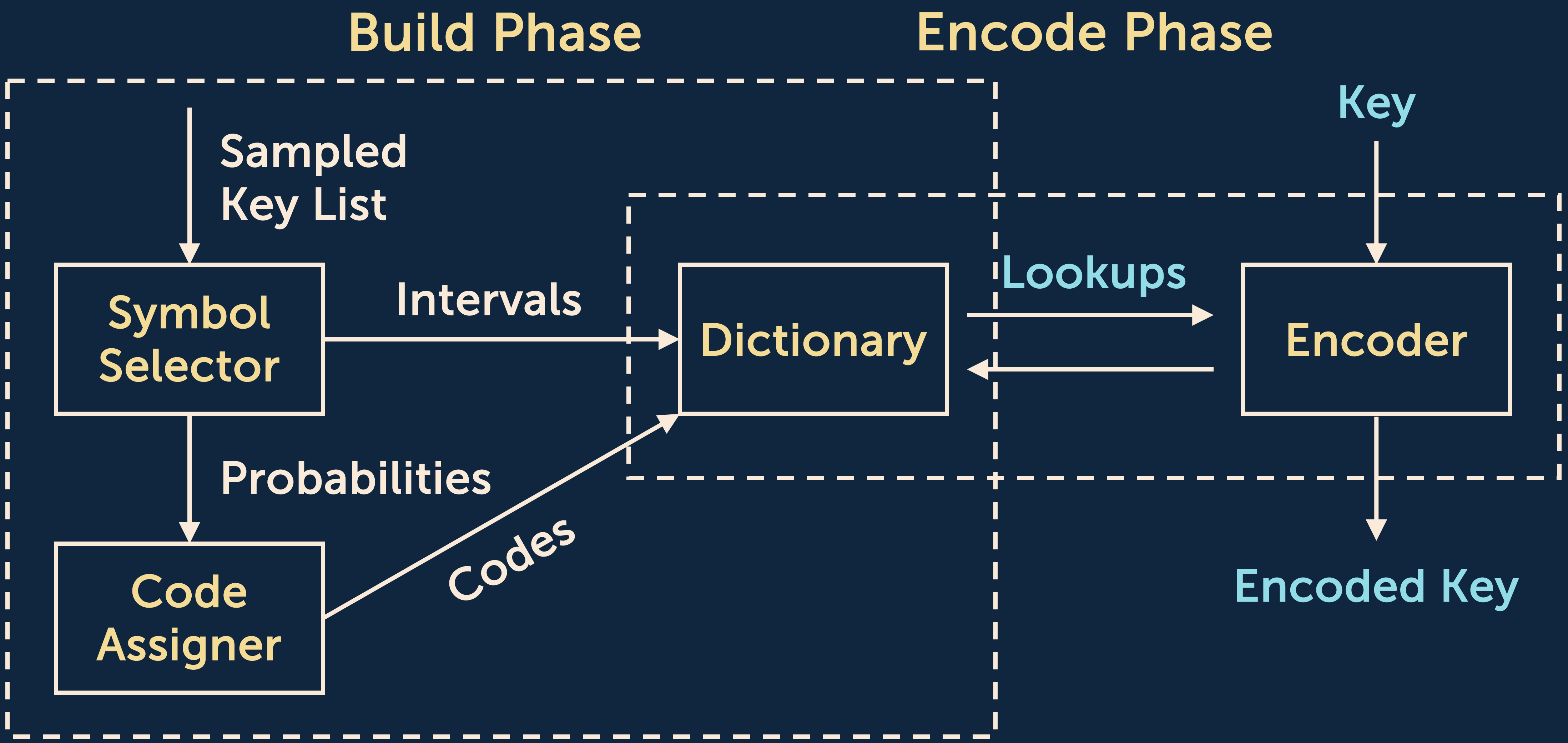
- Dictionary Completeness
- Order-Preserving

Example: amazon → 1 → amazon → 01azon

→ 2 → 01azon → 01100zon

• • •

The HOPE Framework

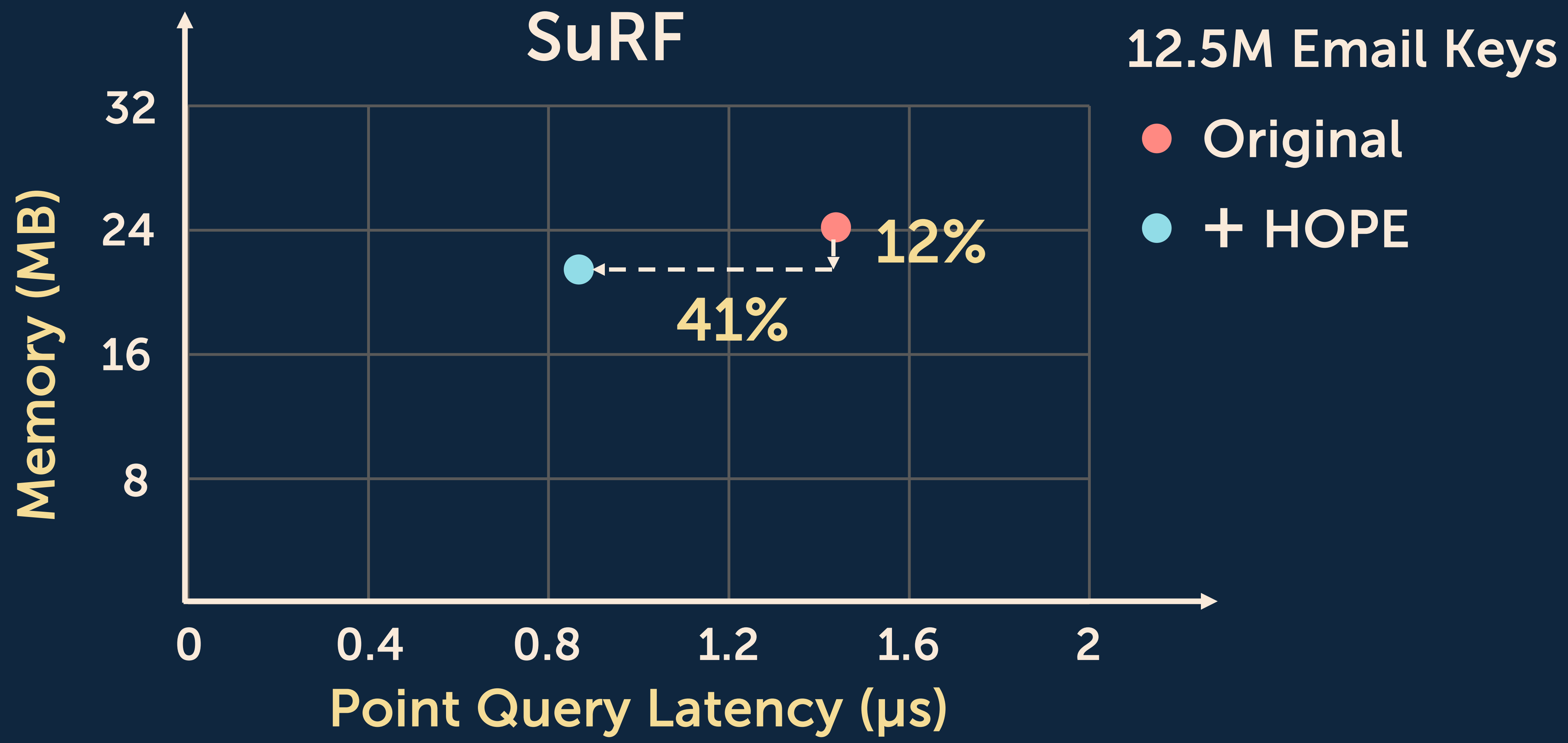


HOPE Evaluation Summary

- B+tree, ART, HOT, SuRF
- Emails, Wikipedia Titles, URLs
- Lookup, Scan, Insert, Update …

30% Smaller + 40% Faster

HOPE is orthogonal to structural compression



1 Build fast static search trees with maximum structural compression

D-to-S Rules, FST, SuRF

Memory-Efficiency

Hybrid Index

HOPE

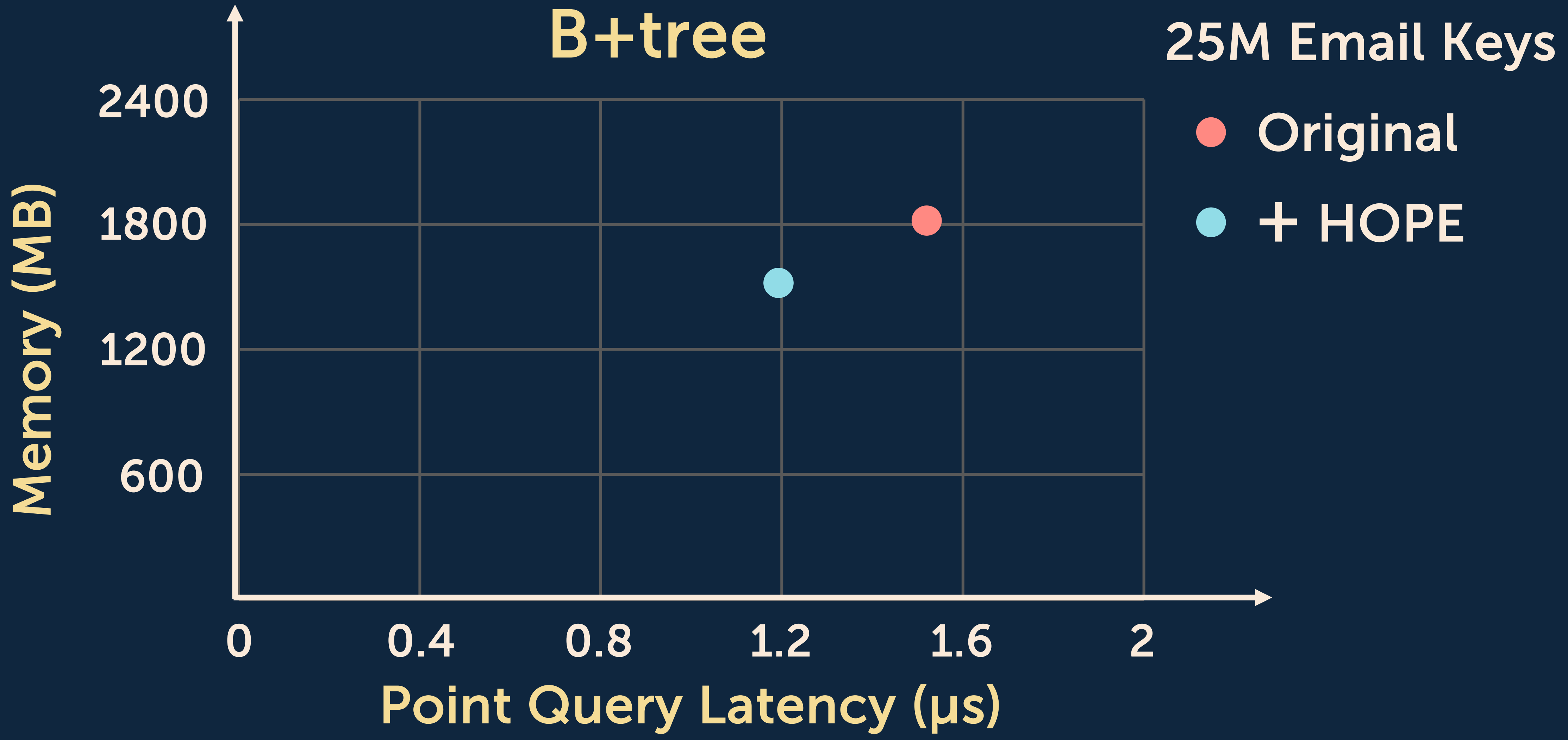
2 Support dynamic operations with bounded & amortized cost

3 Compress input keys efficiently while preserving their order

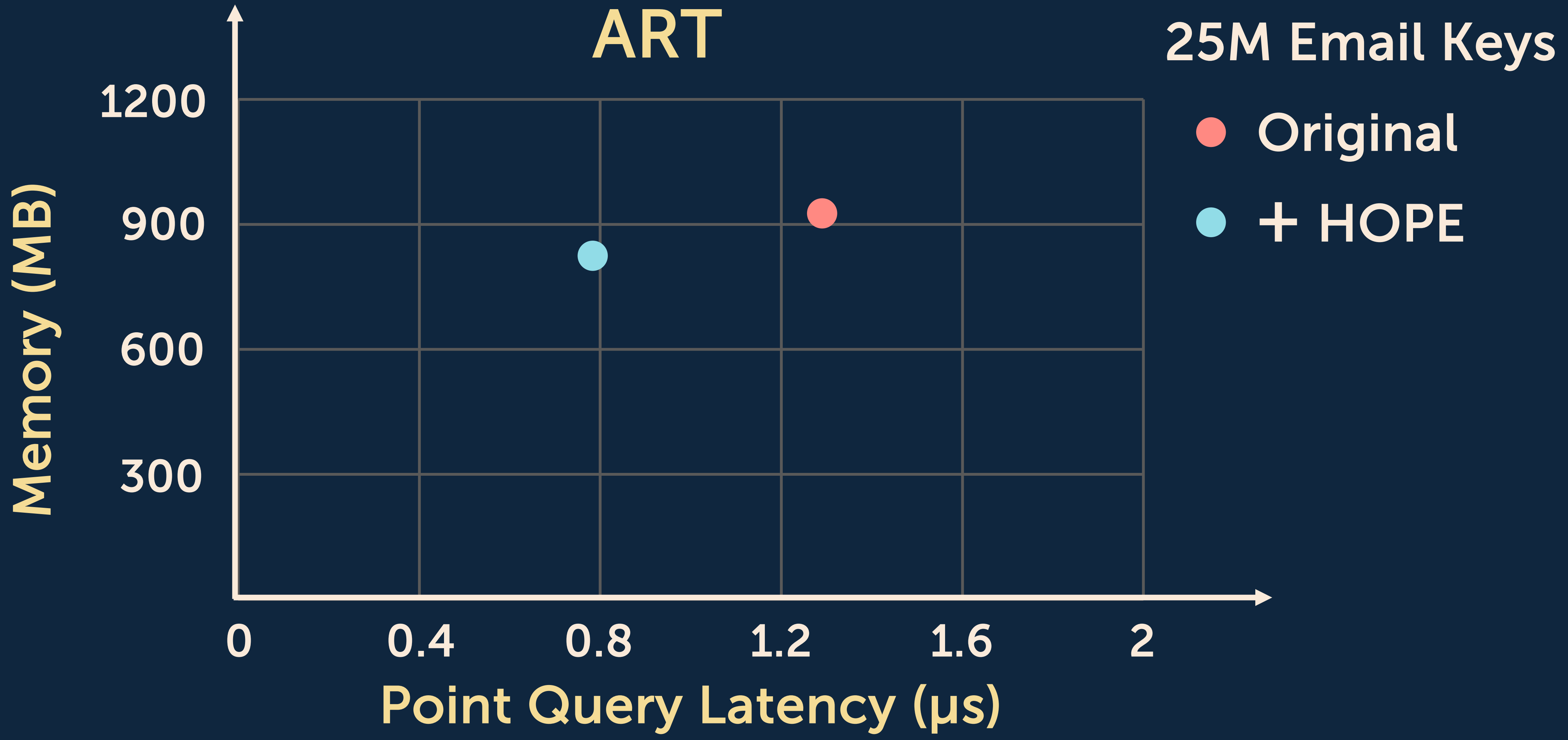
Backup Slides

Concatenation Property Counter-Example

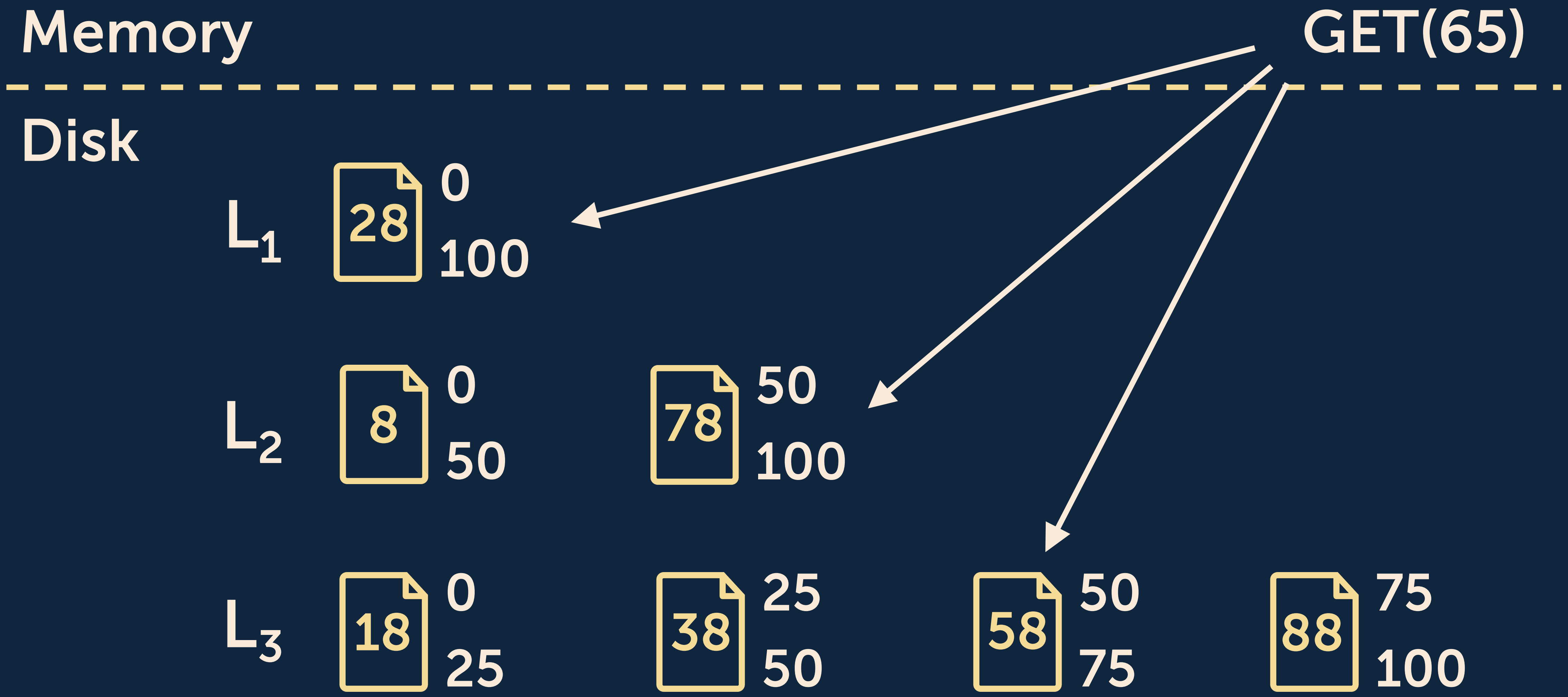
HOPE improves performance & memory-efficiency



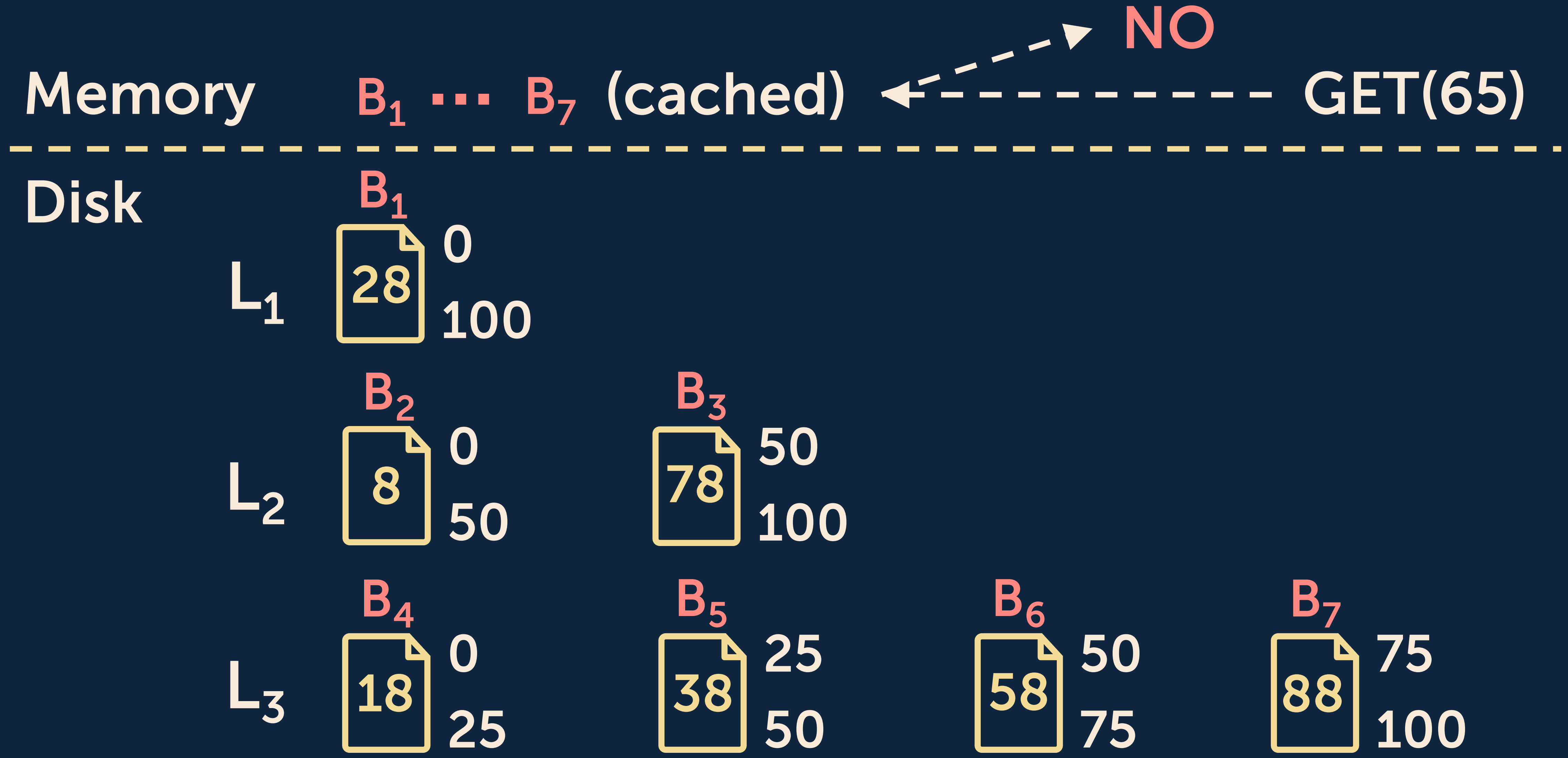
HOPE improves performance & memory-efficiency



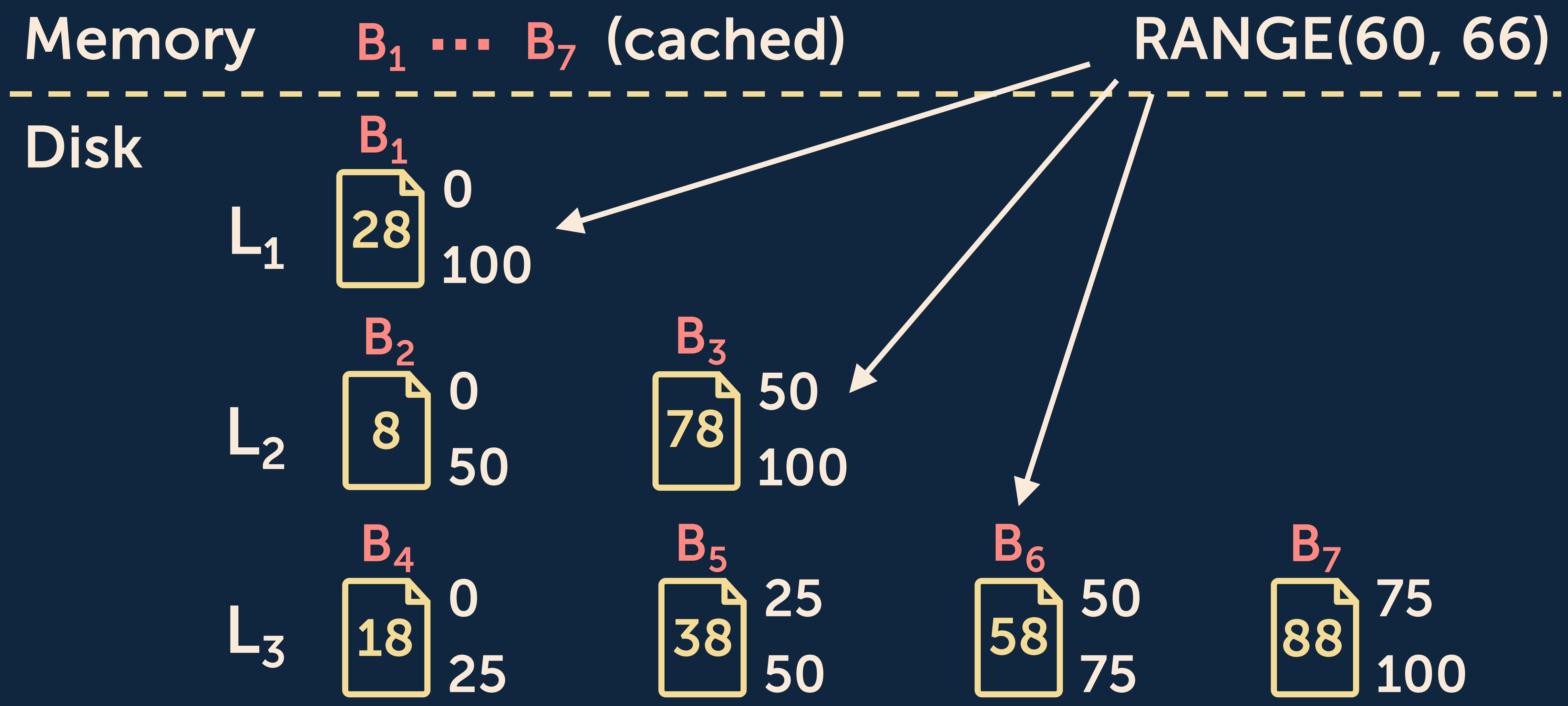
Bloom filters speed up point queries in RocksDB



Bloom filters speed up point queries in RocksDB



Range queries still incur multiple I/Os



SuRFs save I/Os for both point and range queries