SuRF: PRACTICAL RANGE FILTERING WITH FAST SUCCINCT TRIES

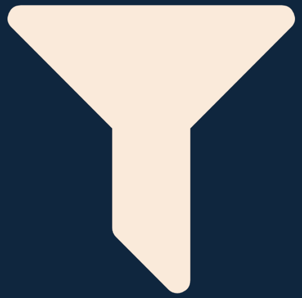
Huanchen Zhang

Hyeontaek Lim, Viktor Leis, David G. Andersen

Michael Kaminsky, Kimberly Keeton, Andrew Pavlo

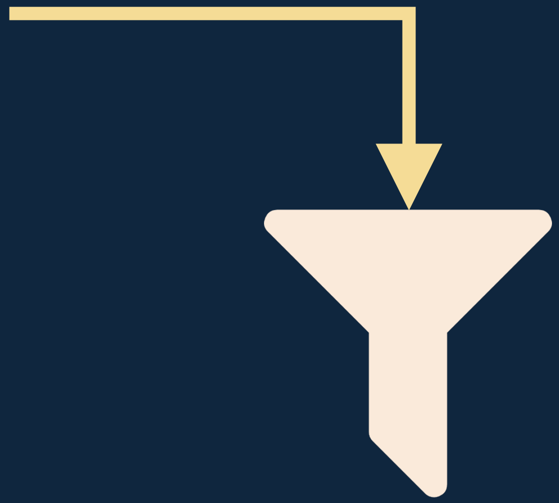
Filters answer approximate membership queries

Filters answer approximate membership queries



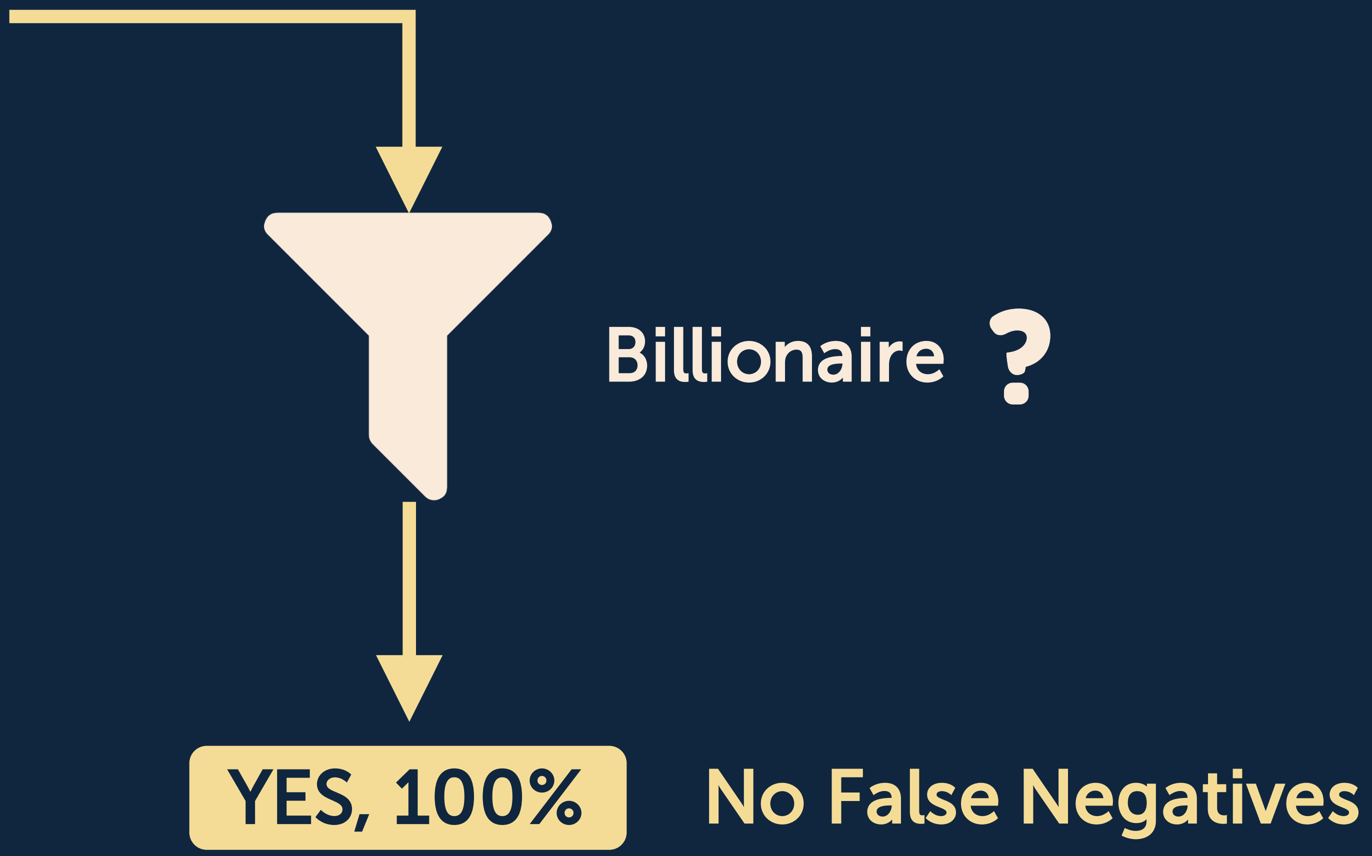
Billionaire ?

Filters answer approximate membership queries

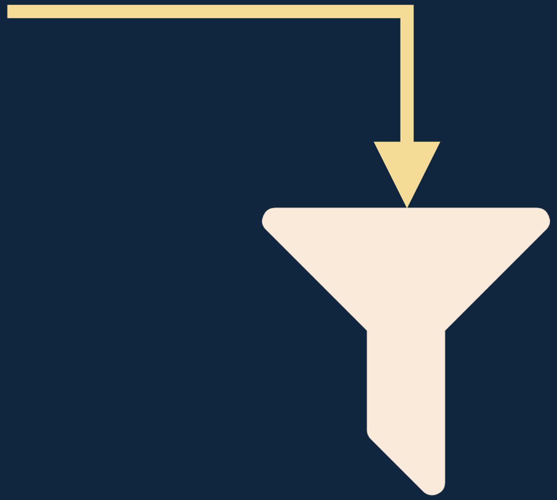


Billionaire ?

Filters answer approximate membership queries

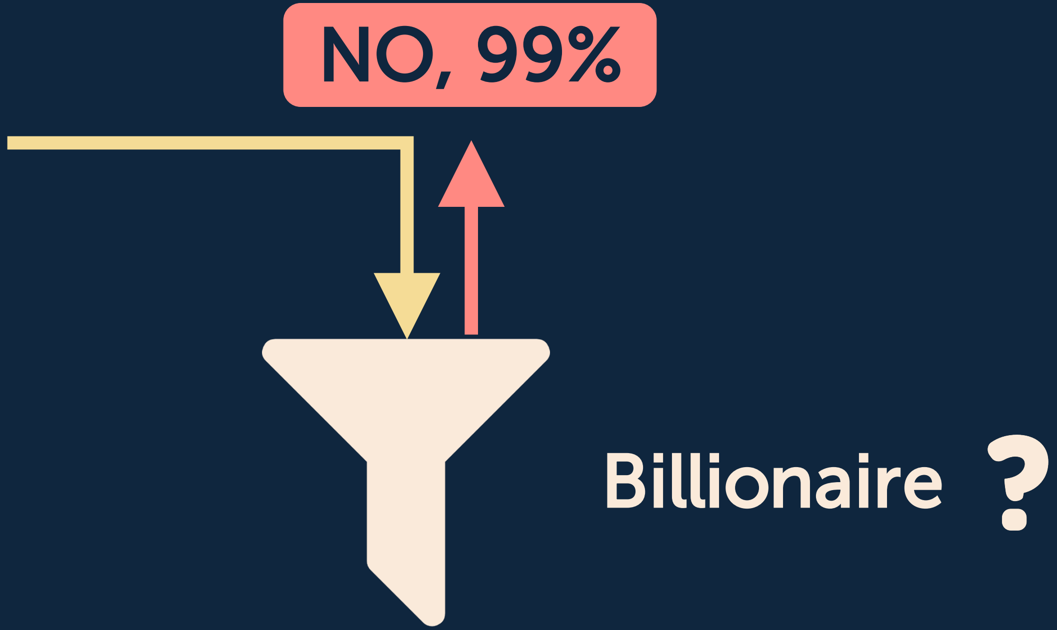


Filters answer approximate membership queries

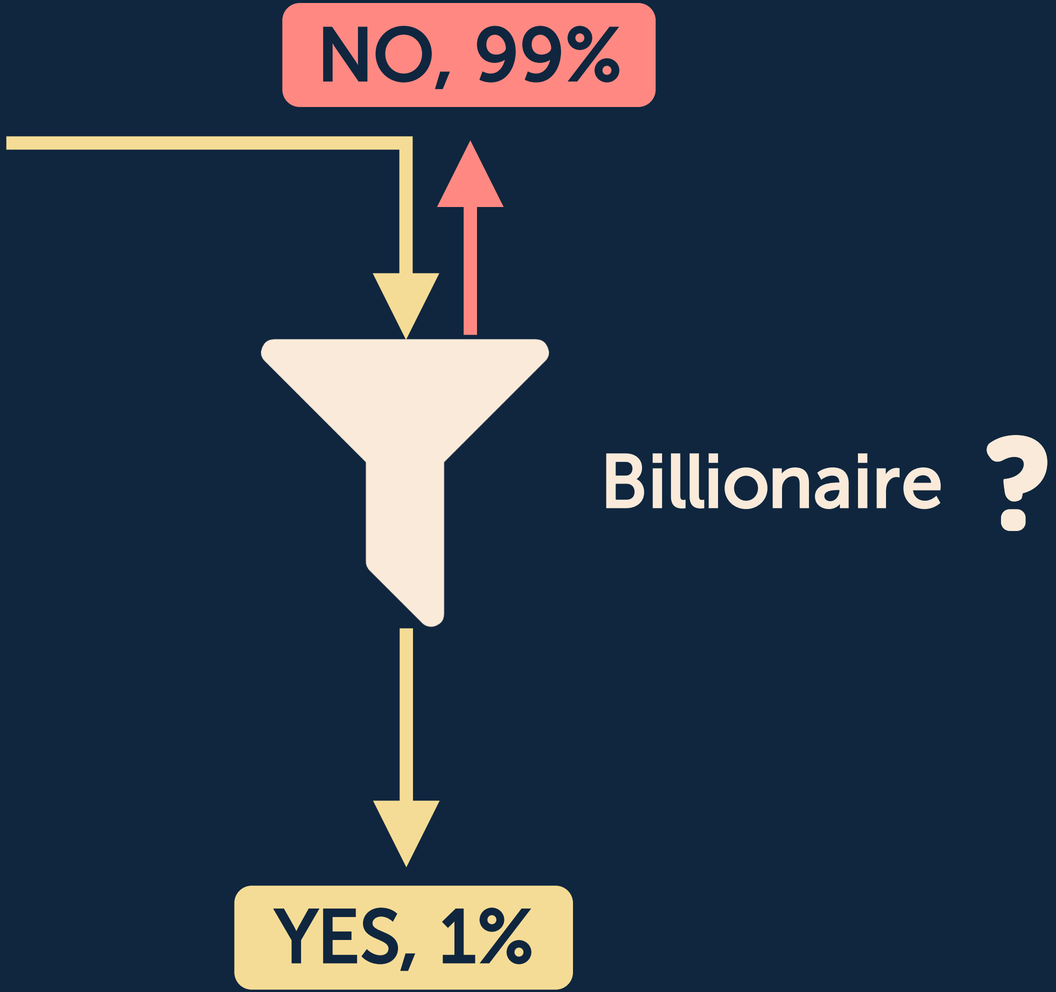


Billionaire ?

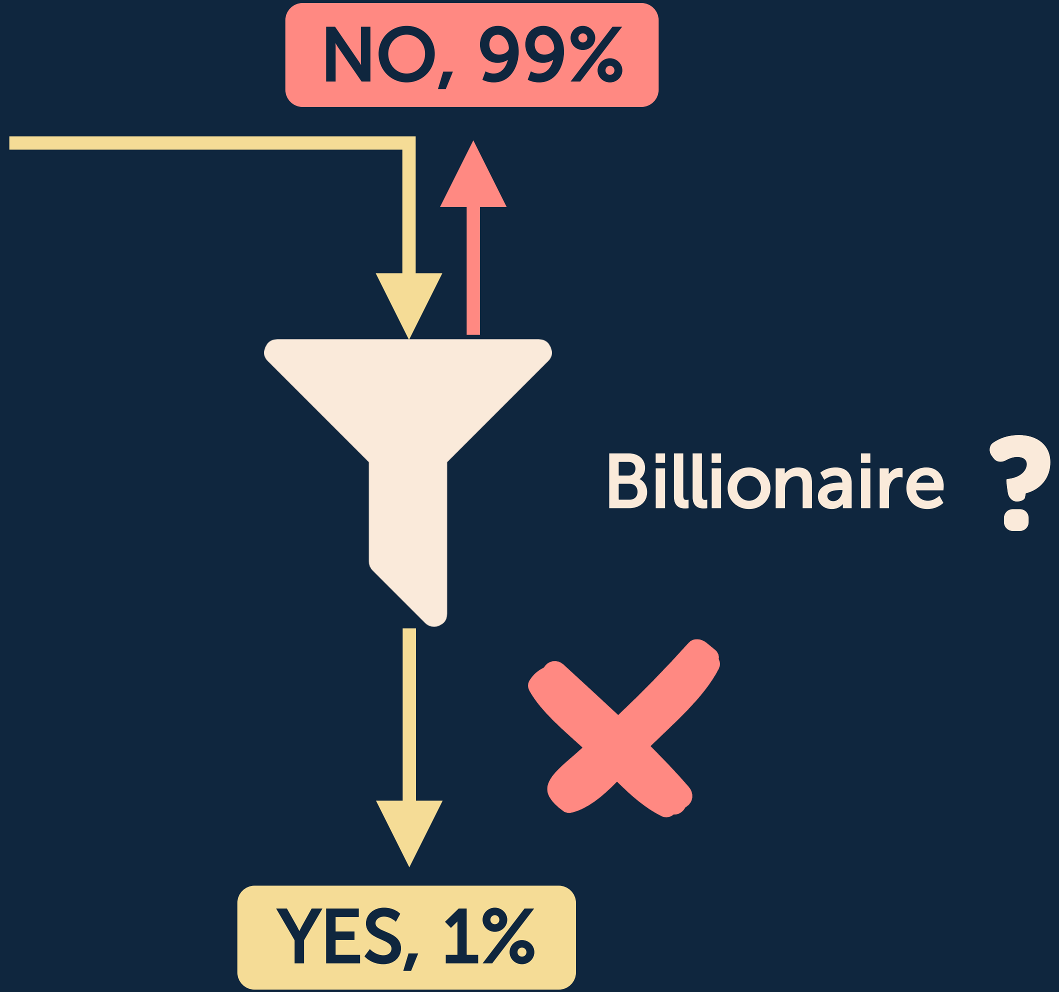
Filters answer approximate membership queries



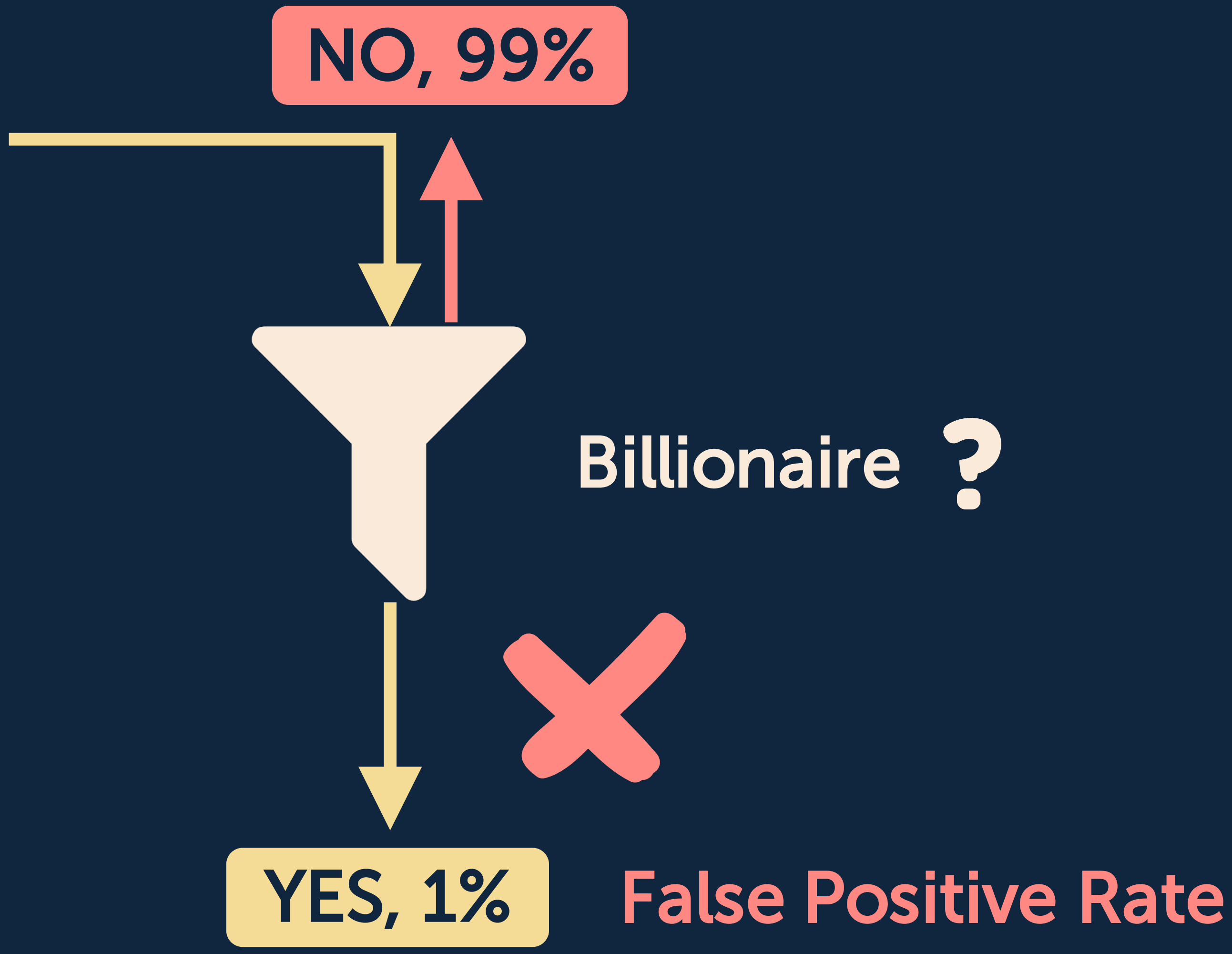
Filters answer approximate membership queries



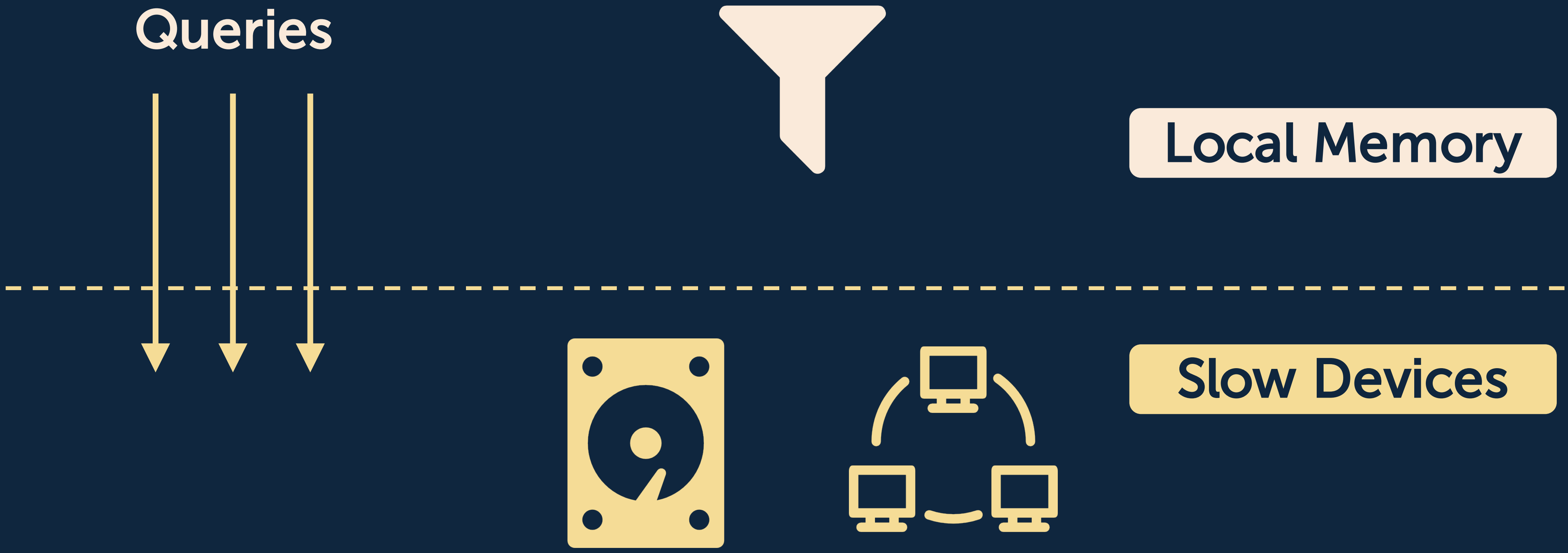
Filters answer approximate membership queries



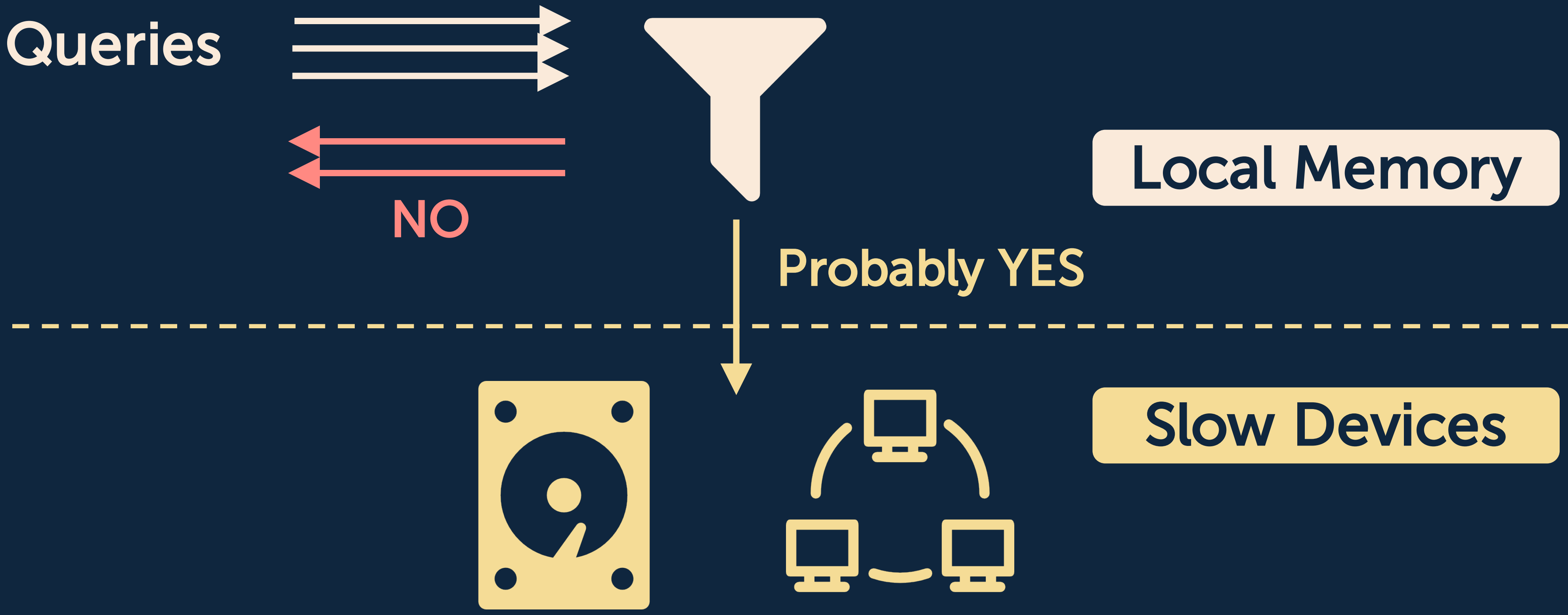
Filters answer approximate membership queries



Filters pre-reject most negative queries



Filters pre-reject most negative queries



Existing filters only support point filtering

Point Filtering

SELECT * FROM Billionaires
WHERE LastName = 'Pavlo'

Bloom Filter (1970)
Quotient Filter (2012)
Cuckoo Filter (2014)

Existing filters only support point filtering

Point Filtering	Range Filtering
SELECT * FROM Billionaires WHERE LastName = ‘Pavlo’	SELECT * FROM Billionaires WHERE LastName LIKE ‘Pav%’

Bloom Filter (1970)
Quotient Filter (2012)
Cuckoo Filter (2014)

Existing filters only support point filtering

Point Filtering	Range Filtering
SELECT * FROM Billionaires WHERE LastName = 'Pavlo'	SELECT * FROM Billionaires WHERE LastName LIKE 'Pav%'
Bloom Filter (1970) Quotient Filter (2012) Cuckoo Filter (2014)	?

Our solution: Succinct Range Filters (SuRF)

First practical, general-purpose range filter

SMALL: close to theoretic minimum

64-bit integer keys, 1% false positive rate: ≈ 12 bits per key

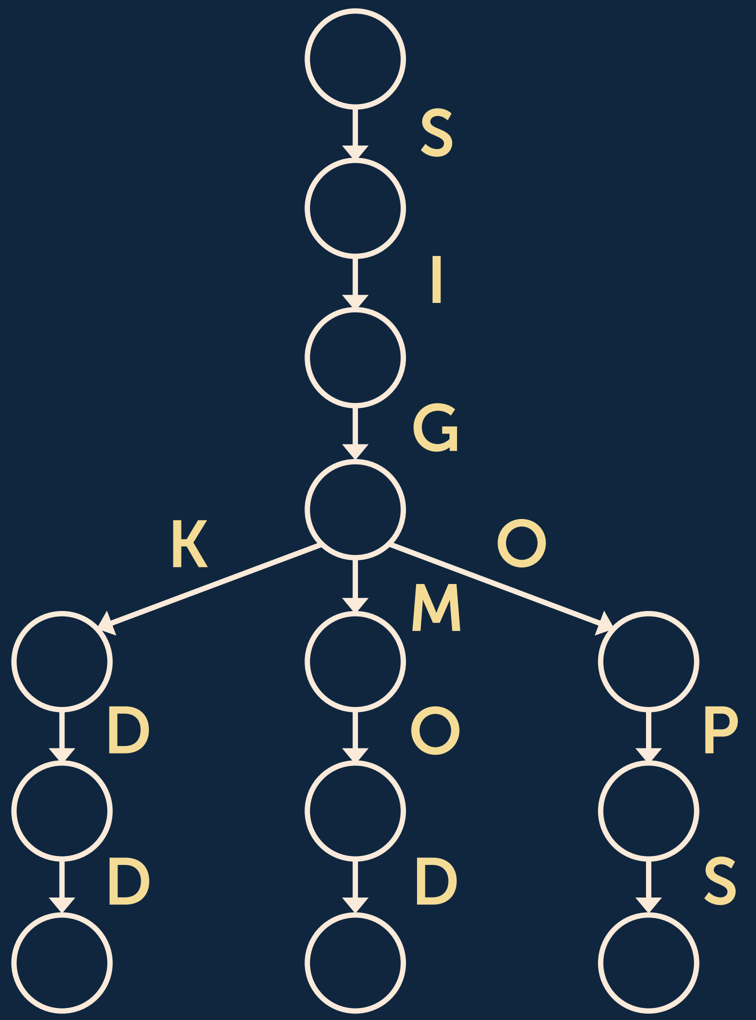
FAST: comparable to fastest trees

10 million 64-bit integer keys: ≈ 200 ns per query

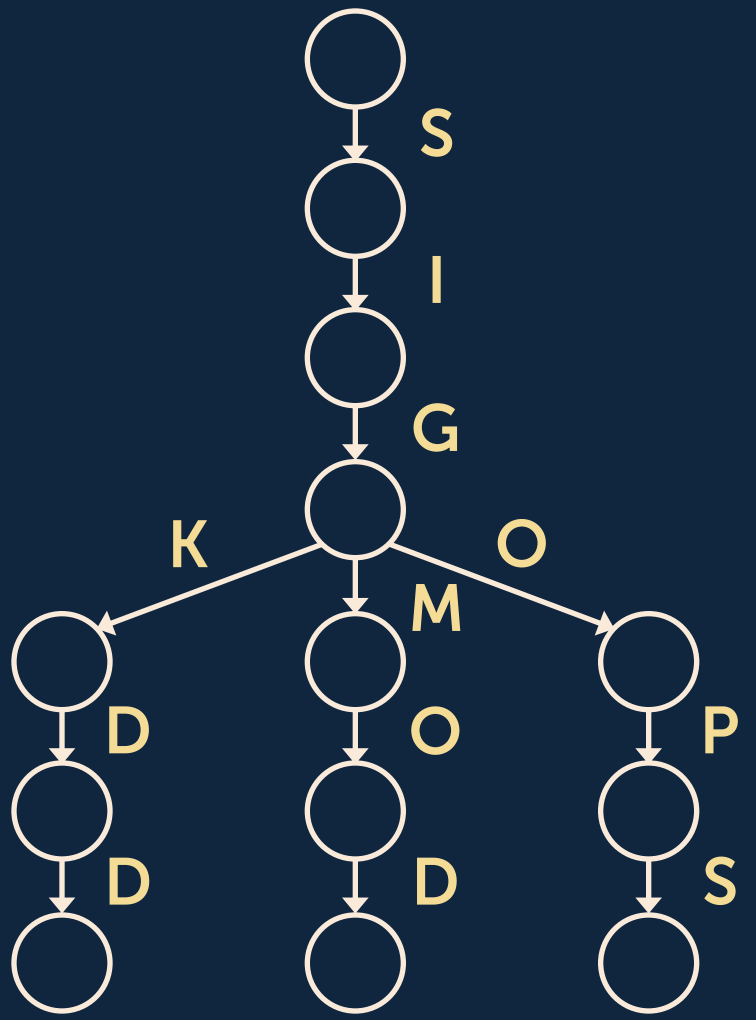
USEFUL: evaluated in RocksDB

speed up range queries by up to 5x

Starting point: a complete trie

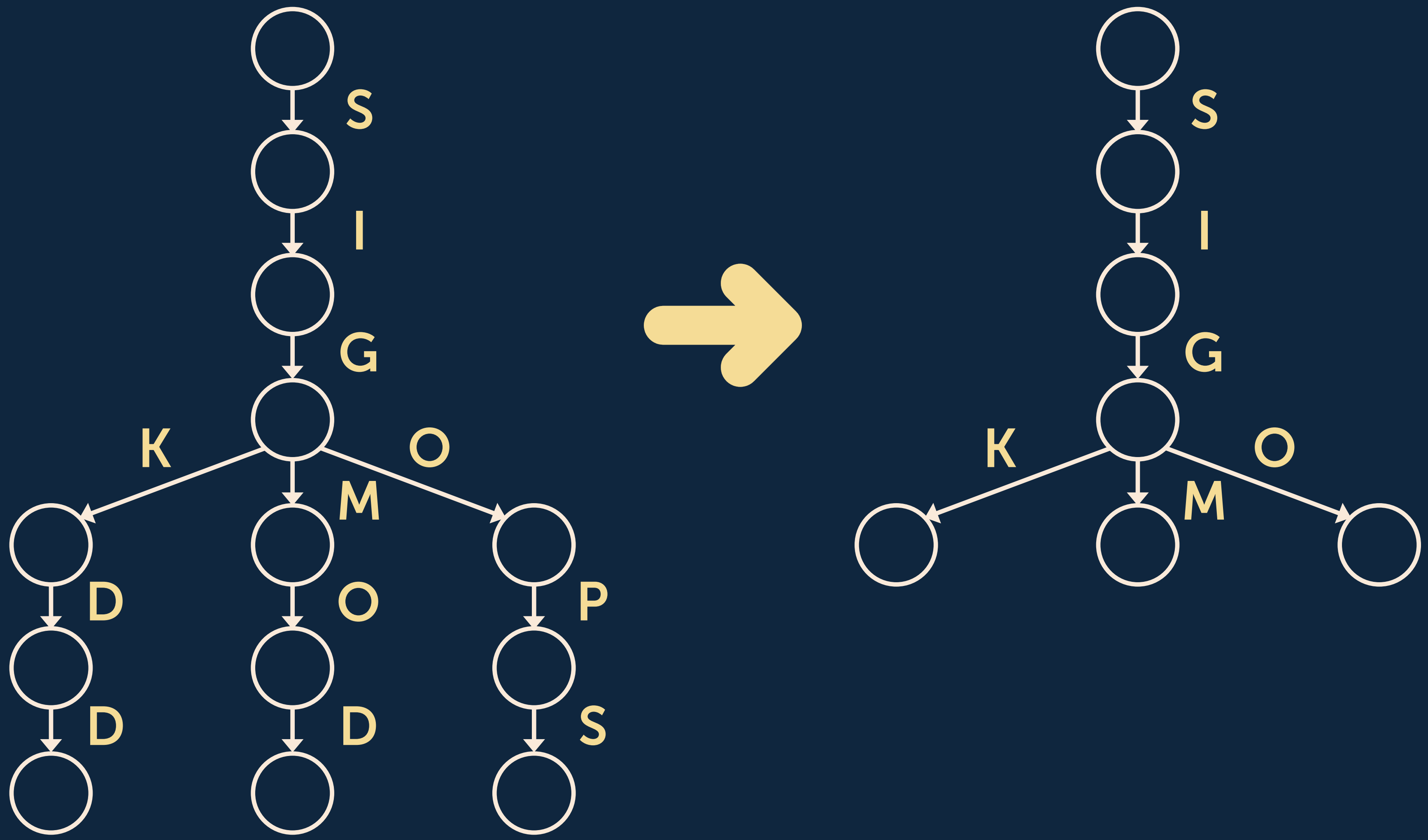


Starting point: a complete trie

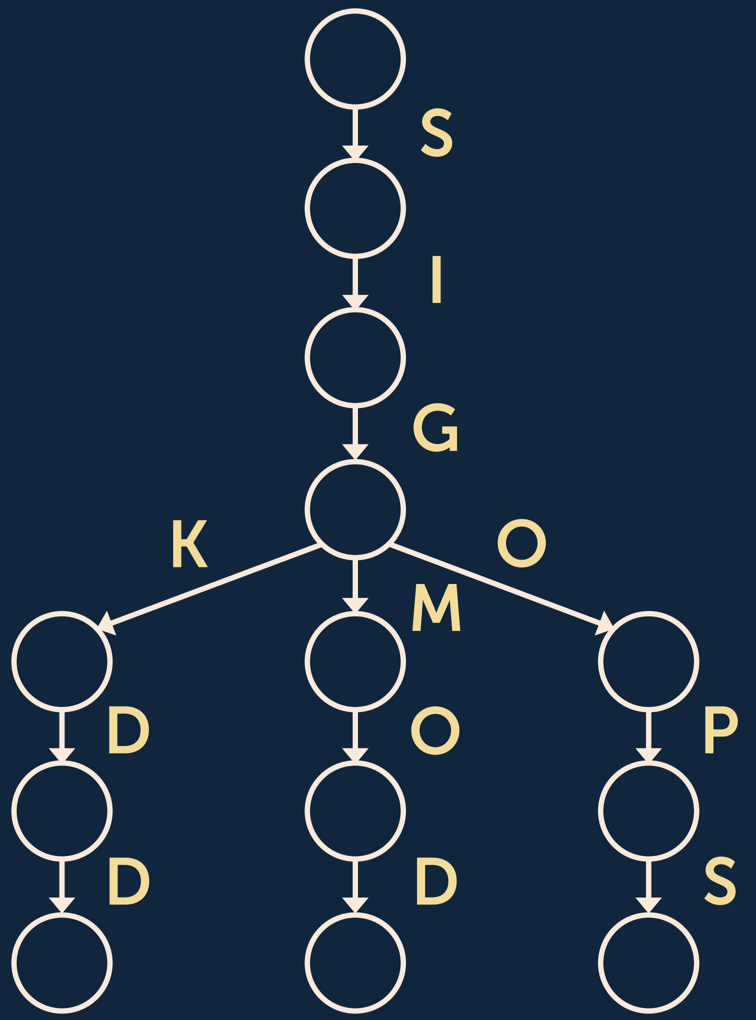


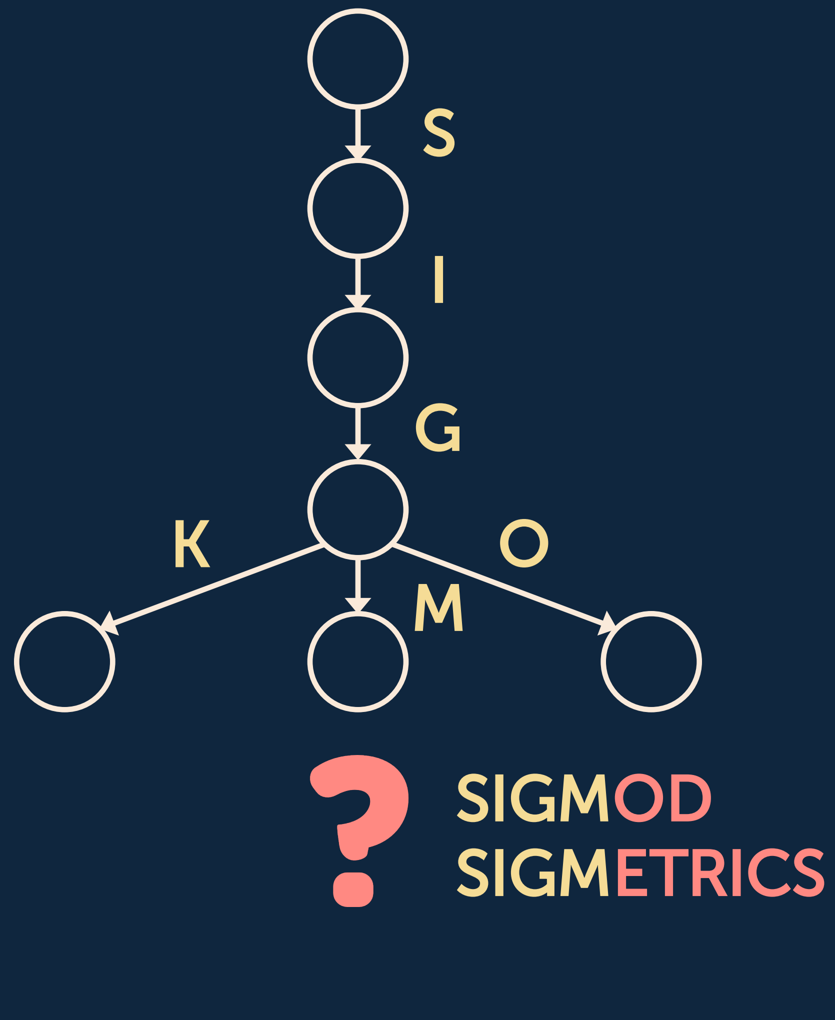
TOO BIG ✗

Make it smaller: a truncated trie



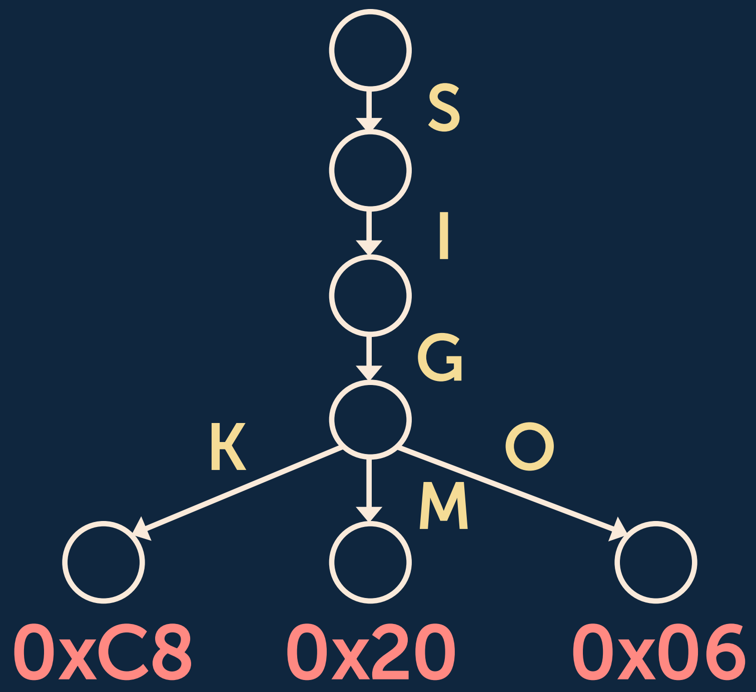
Make it smaller: a truncated trie



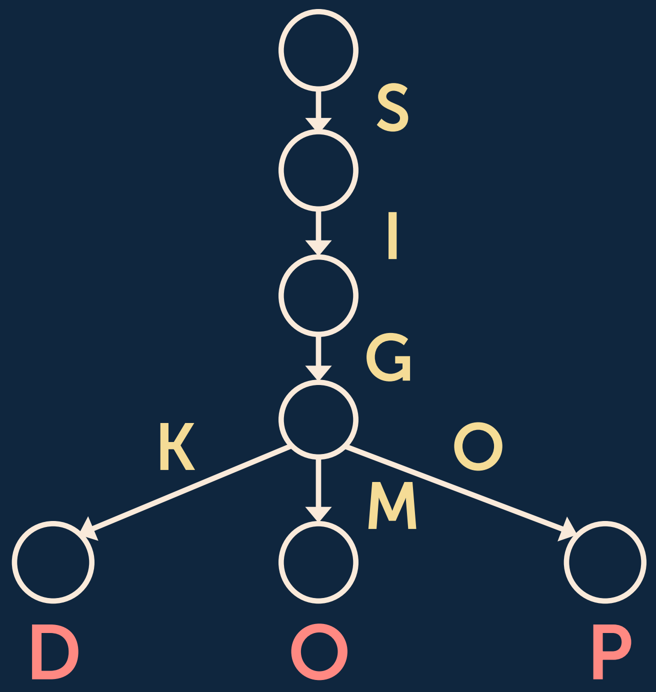


Use suffix bits to reduce false positive rate

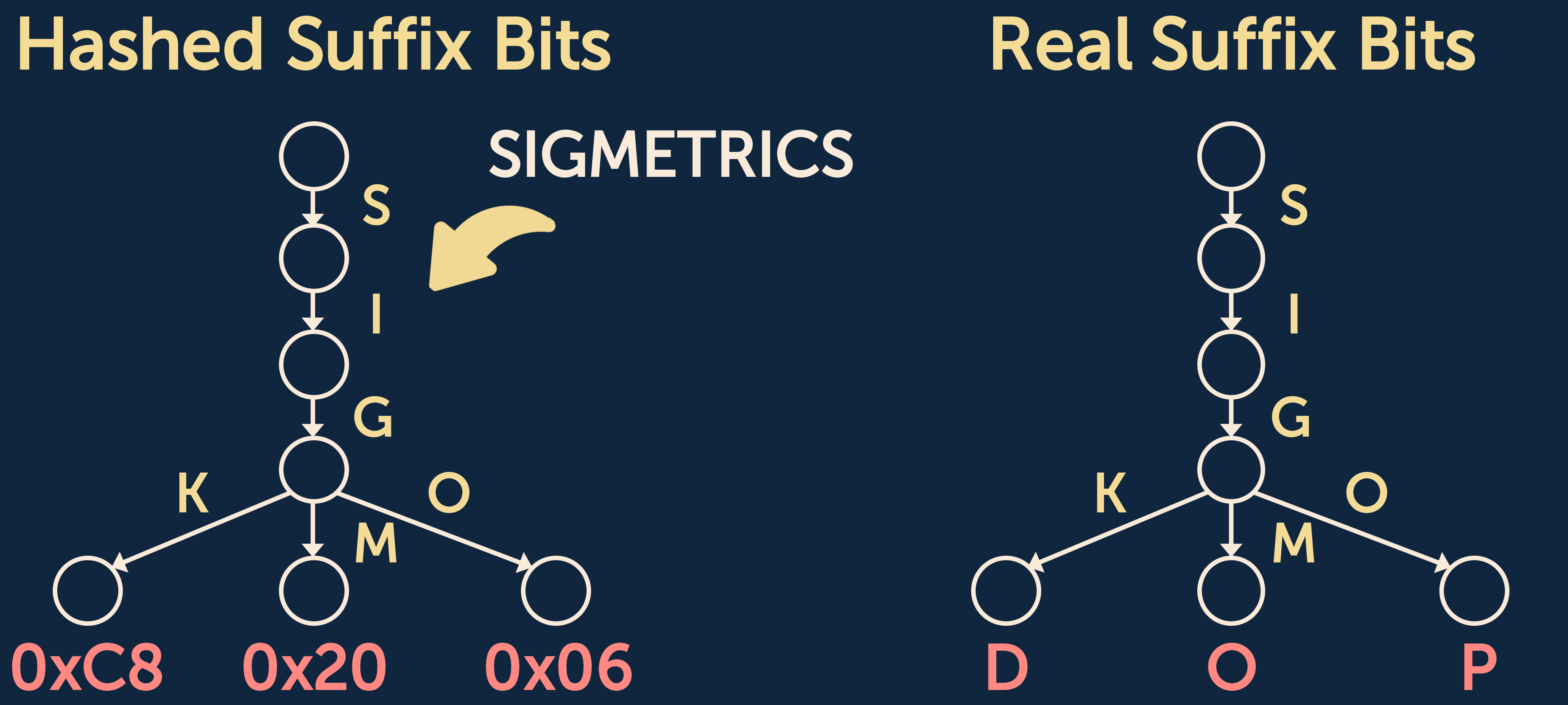
Hashed Suffix Bits



Real Suffix Bits

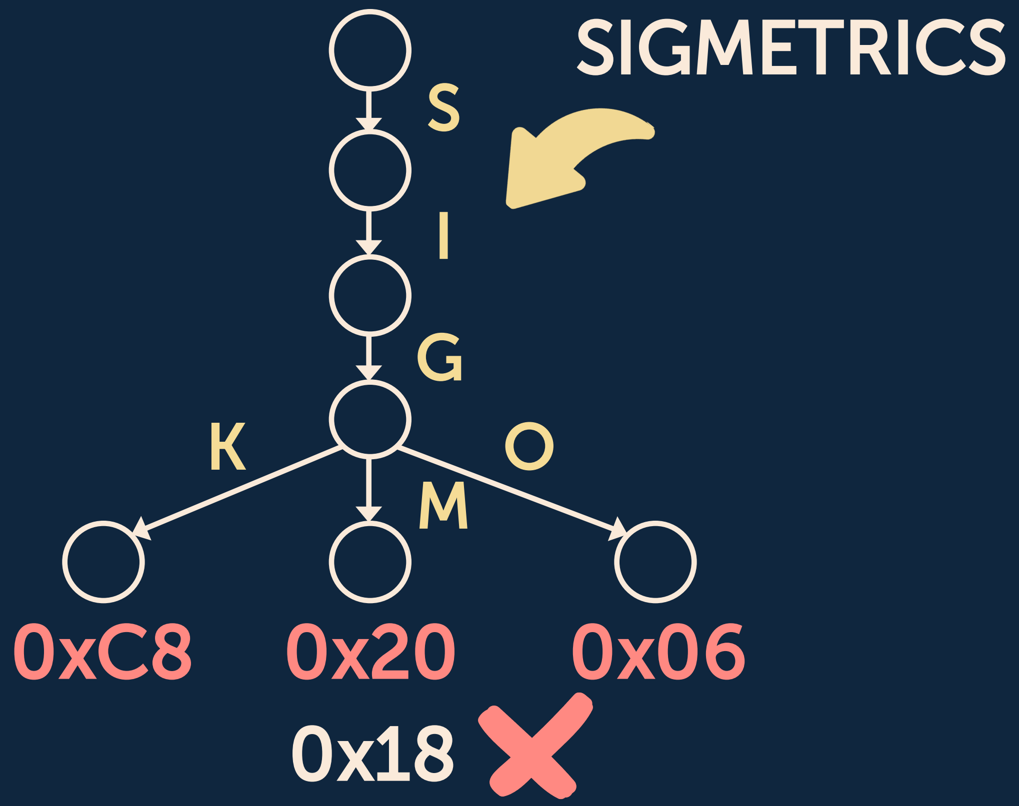


Use suffix bits to reduce false positive rate

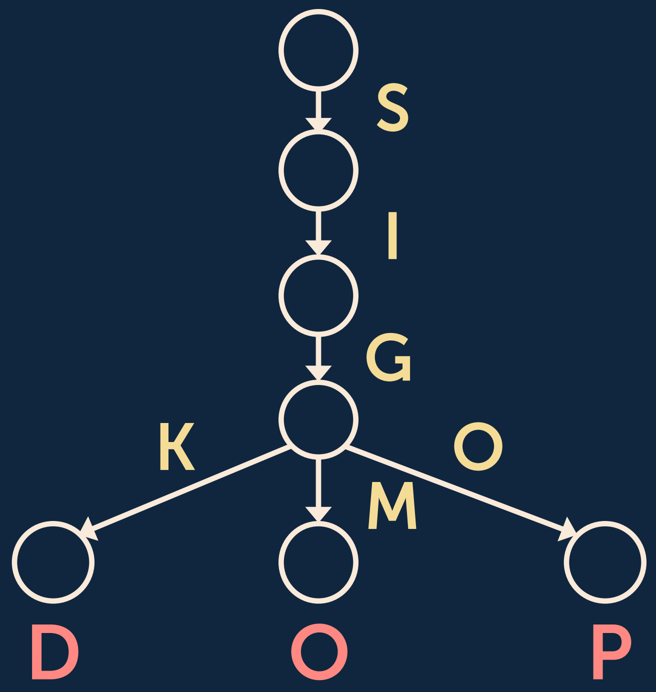


Use suffix bits to reduce false positive rate

Hashed Suffix Bits

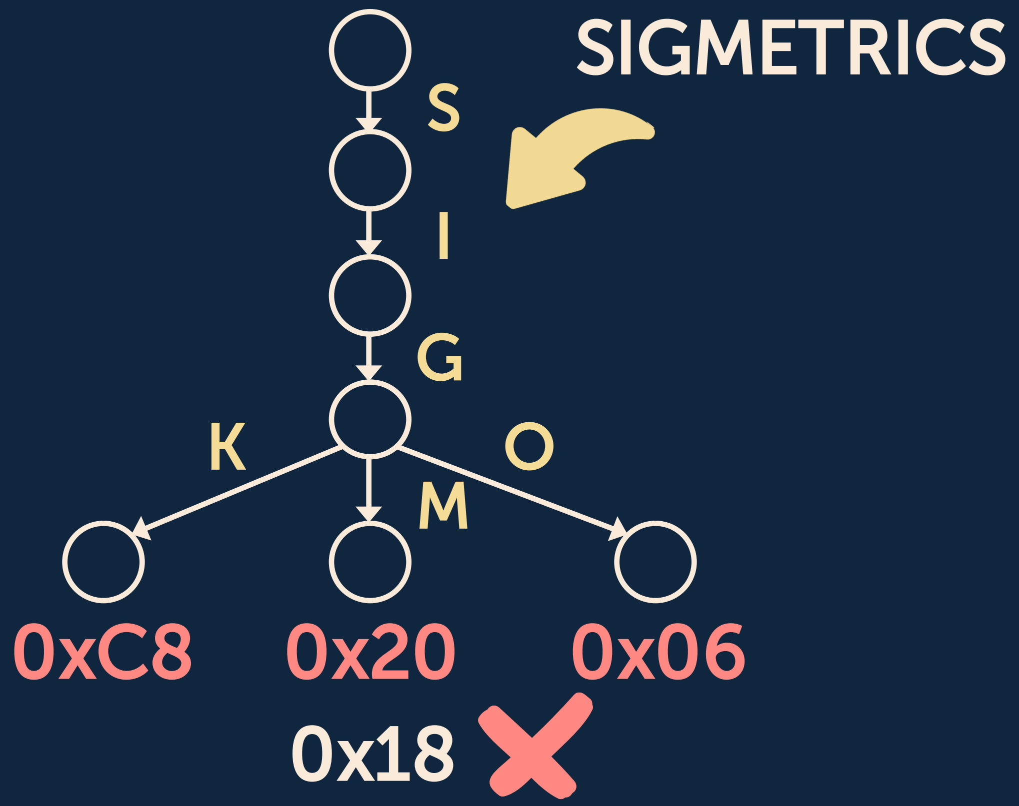


Real Suffix Bits

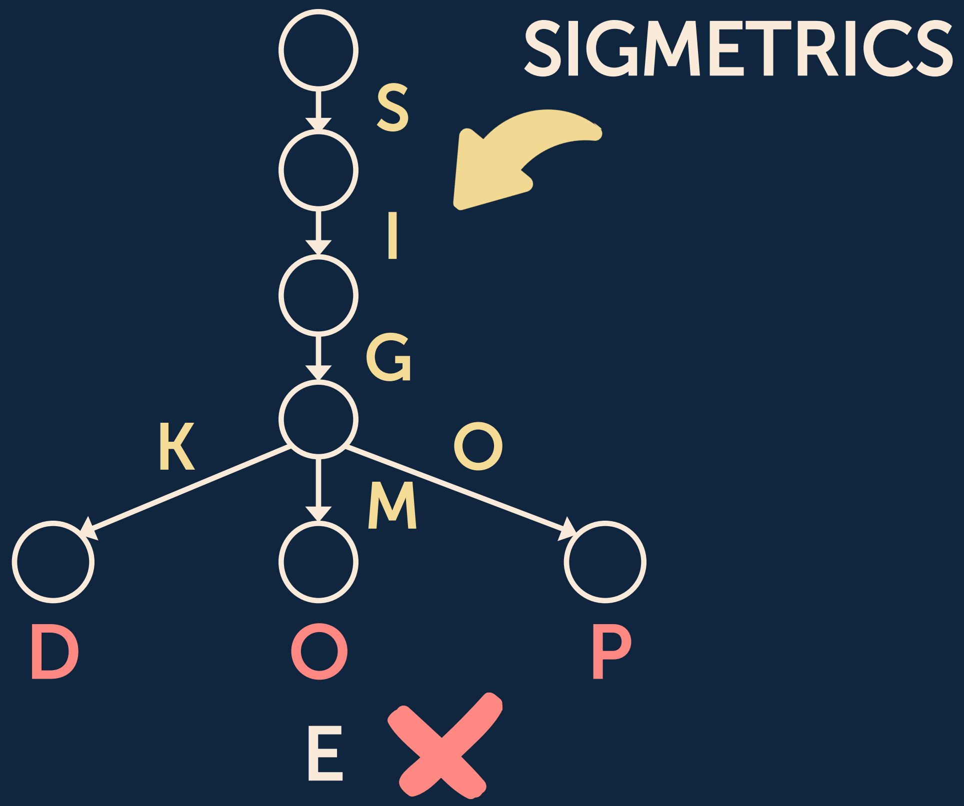


Use suffix bits to reduce false positive rate

Hashed Suffix Bits

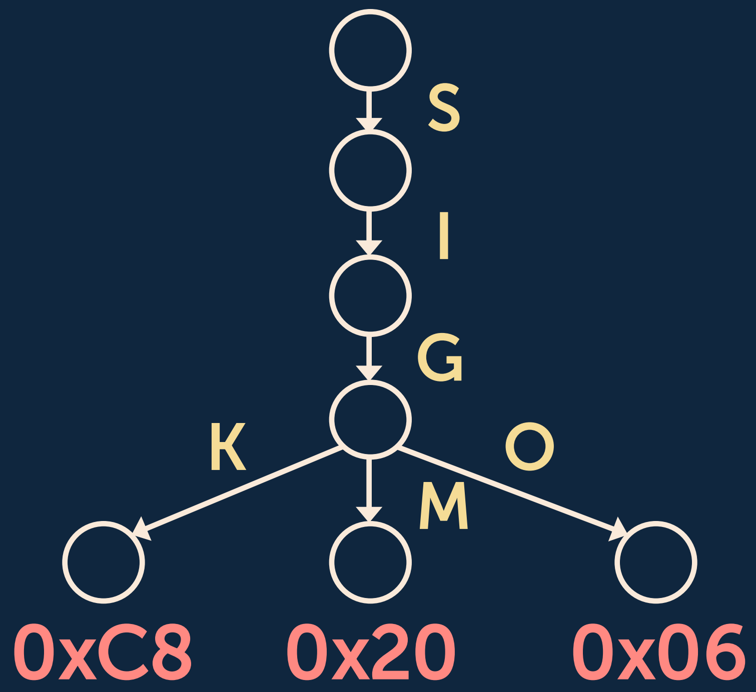


Real Suffix Bits

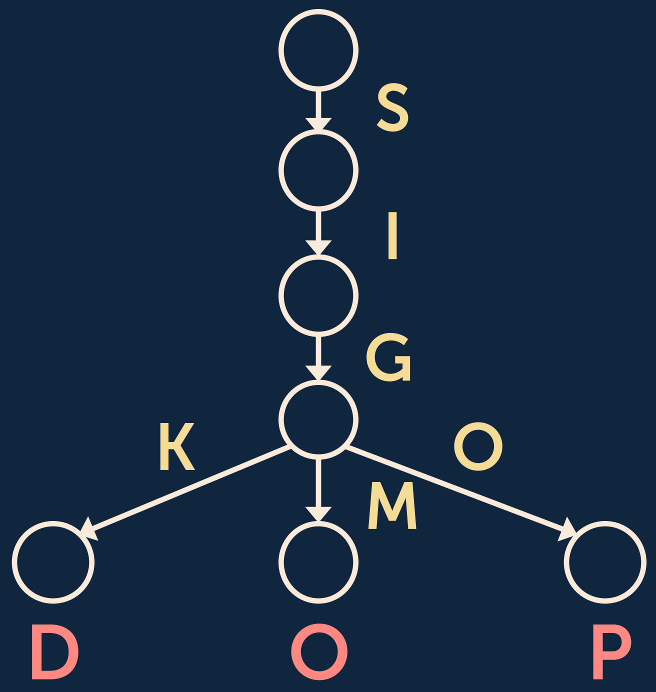


Use suffix bits to reduce false positive rate

Hashed Suffix Bits

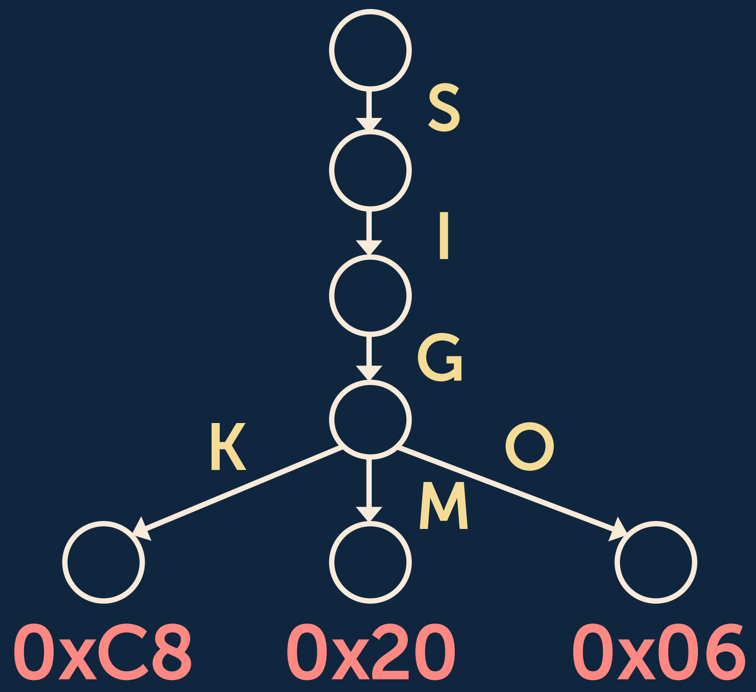


Real Suffix Bits

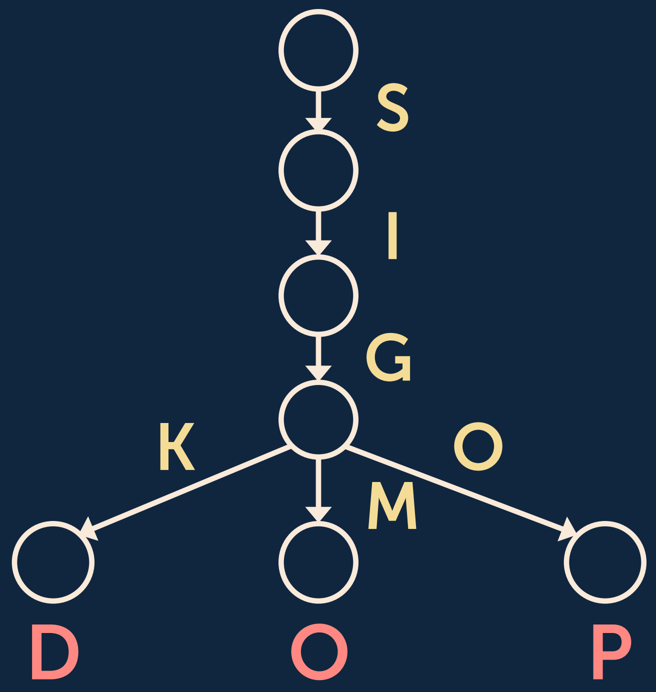


Use suffix bits to reduce false positive rate

Hashed Suffix Bits



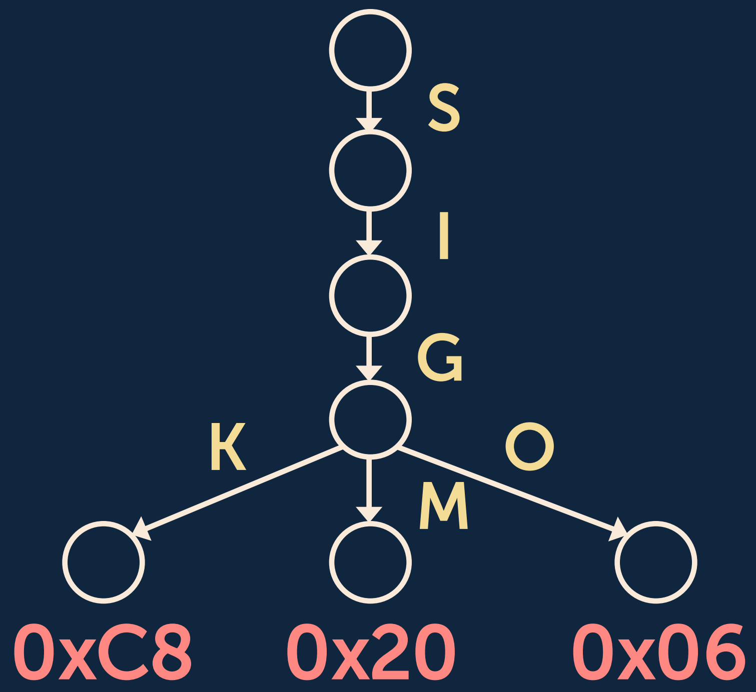
Real Suffix Bits



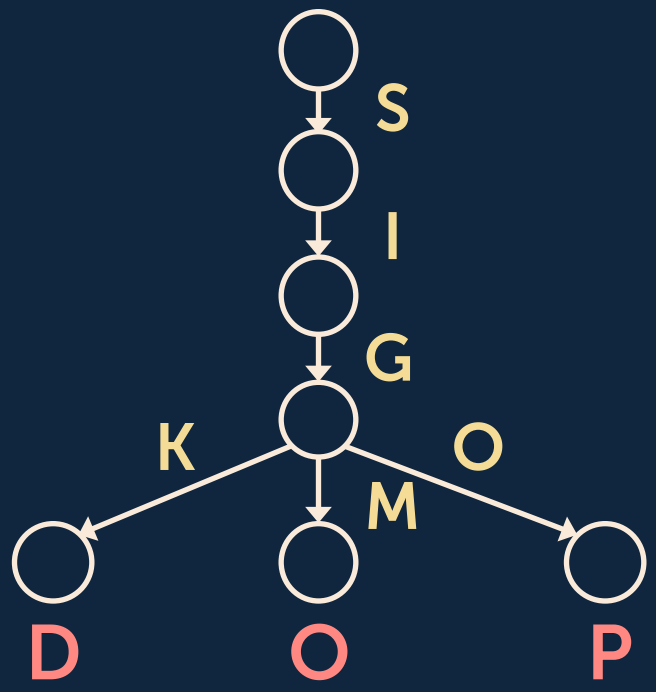
+ Each bit reduces FPR by half

Use suffix bits to reduce false positive rate

Hashed Suffix Bits



Real Suffix Bits

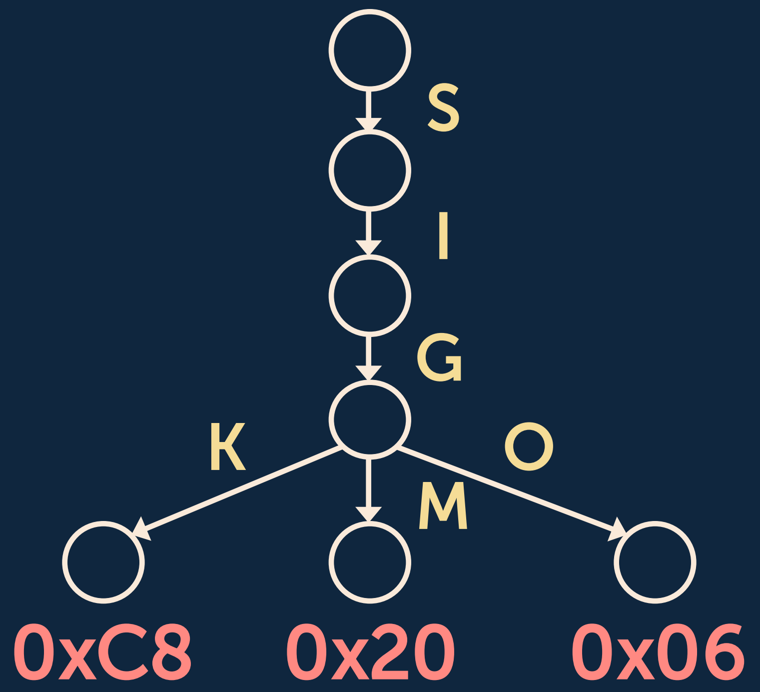


+ Each bit reduces FPR by half

− Cannot help range queries

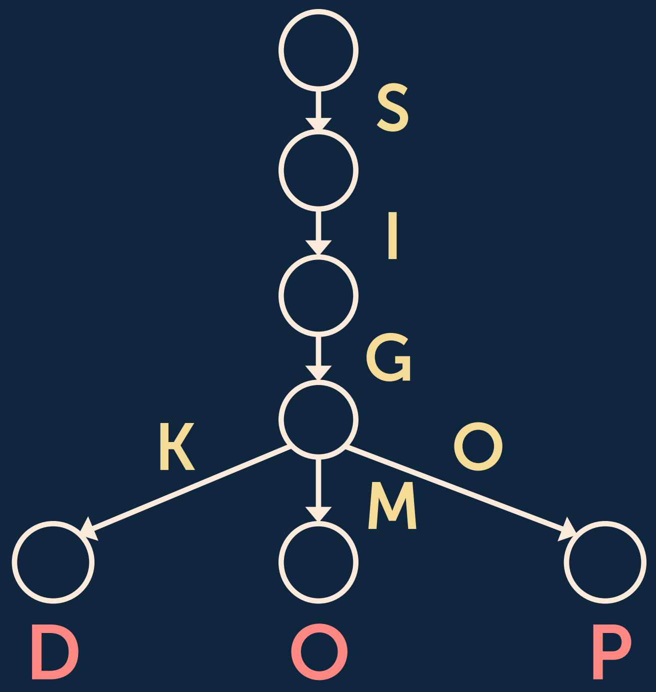
Use suffix bits to reduce false positive rate

Hashed Suffix Bits



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

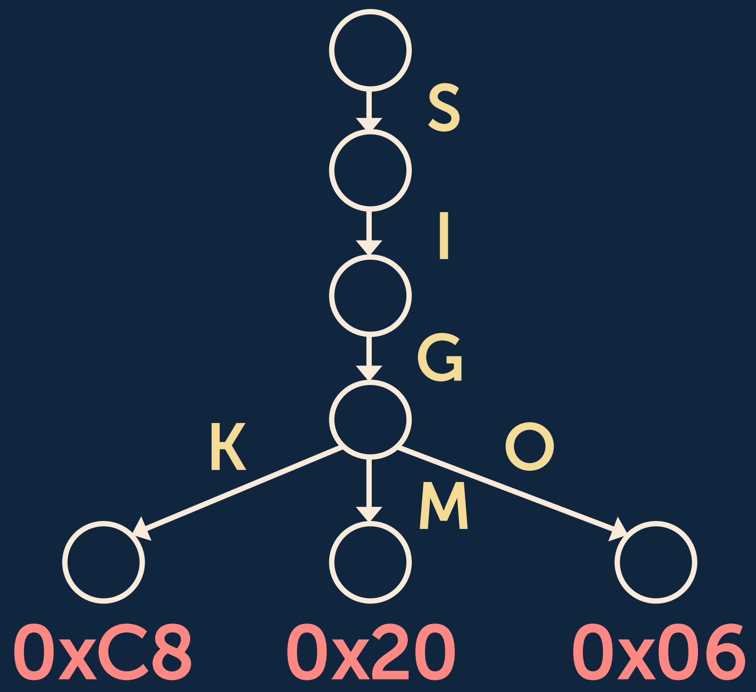
Real Suffix Bits



- Benefit point & range queries

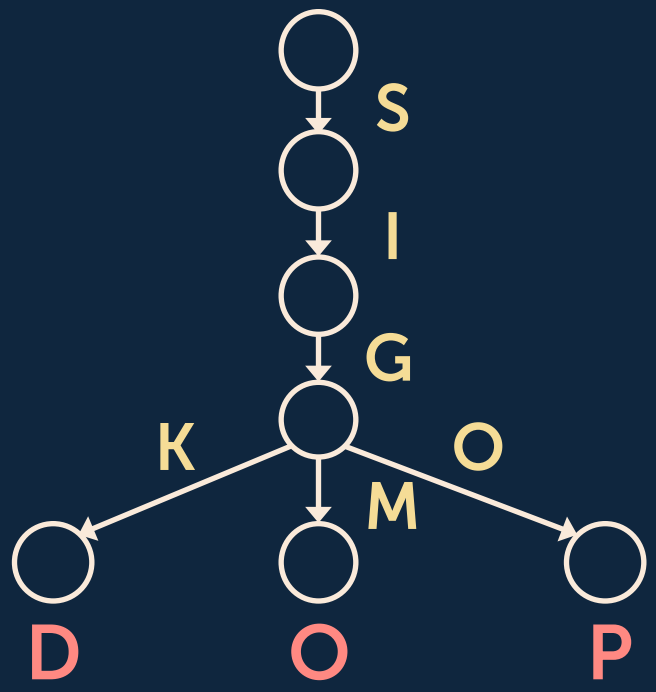
Use suffix bits to reduce false positive rate

Hashed Suffix Bits



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

Real Suffix Bits



- ➕ Benefit point & range queries
- ➖ Weaker distinguishability

Succinct Data Structure

… uses an amount of space that is “close” to the information-theoretic lower bound, but still allows efficient query operations. [wikipedia]

SuRF's encoding is small and fast

① Small

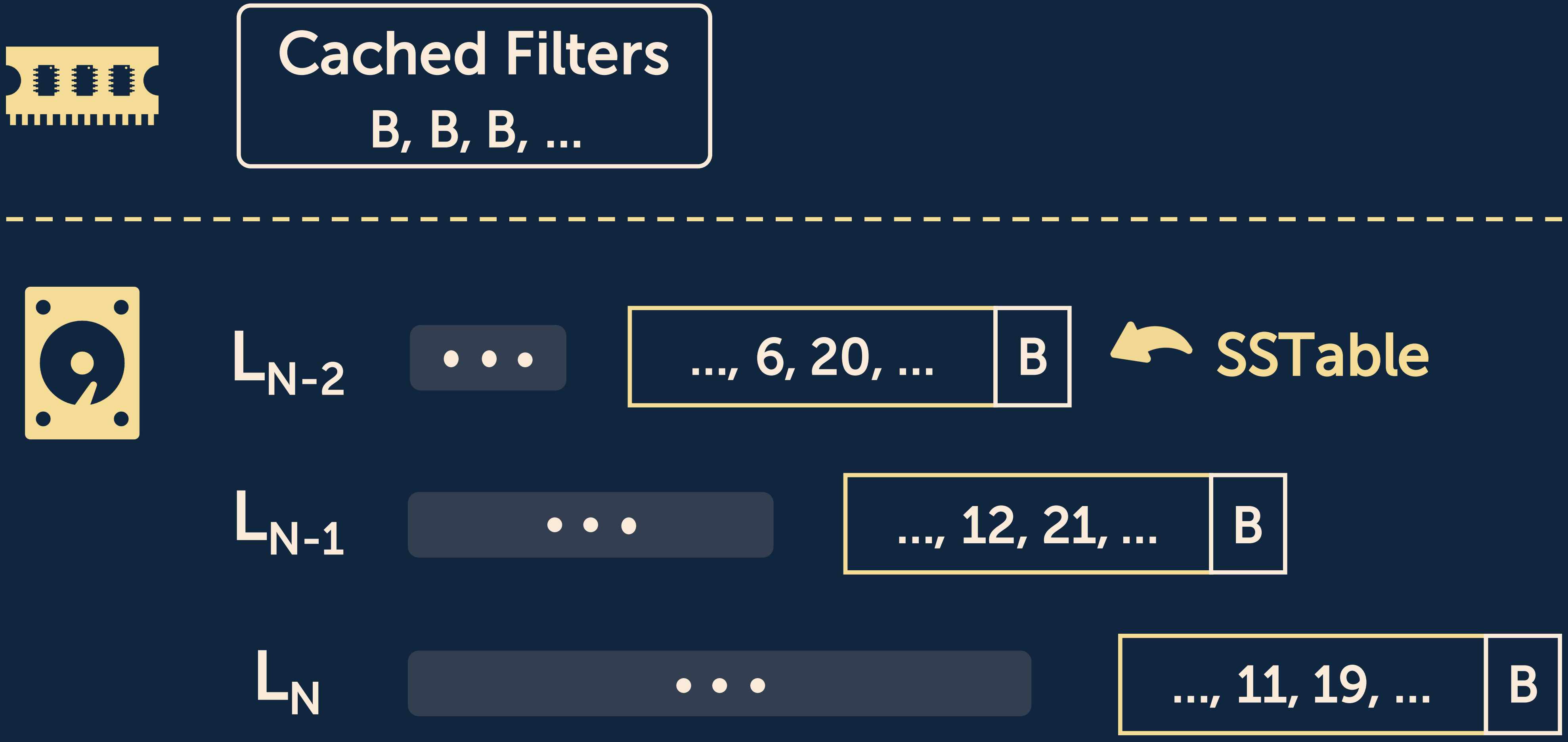
≈10 + suffix bits per key for 64-bit integers

≈14 + suffix bits per key for emails

② Fast

Matches state-of-the-art pointer-based trees

Bloom filters speed up point queries in RocksDB



Bloom filters speed up point queries in RocksDB

Cached Filters

B, B, B, …

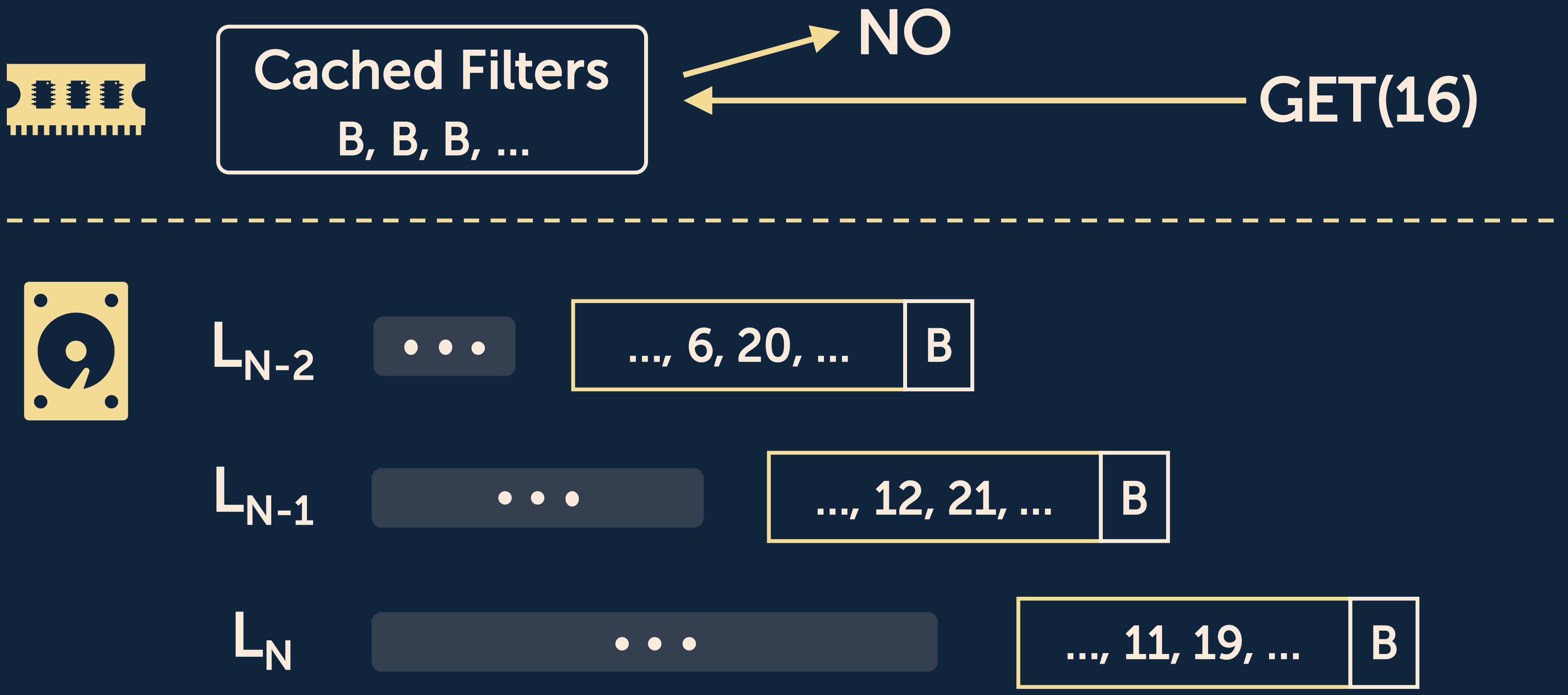
GET(16)

L_{N-2} • • • | …, 6, 20, … | B

L_{N-1} • • • | …, 12, 21, … | B

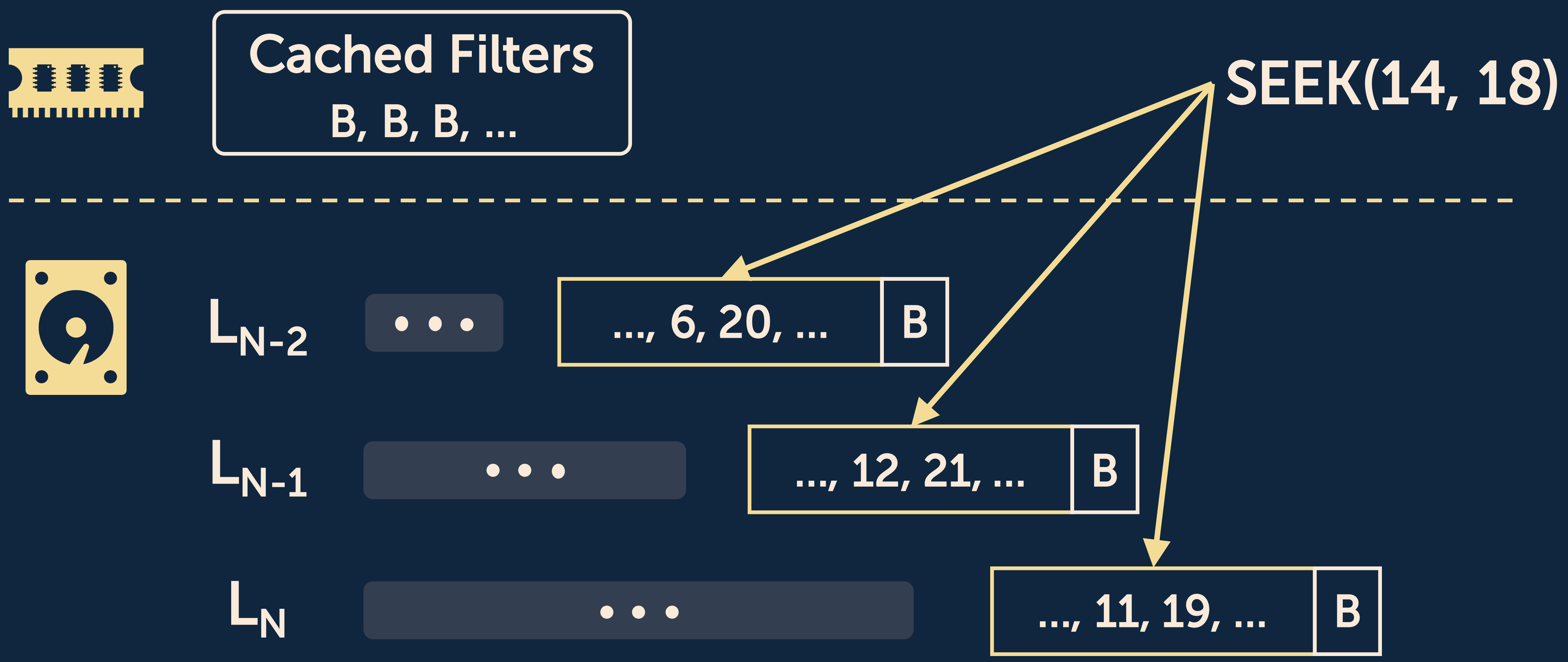
L_N • • • | …, 11, 19, … | B

Bloom filters speed up point queries in RocksDB

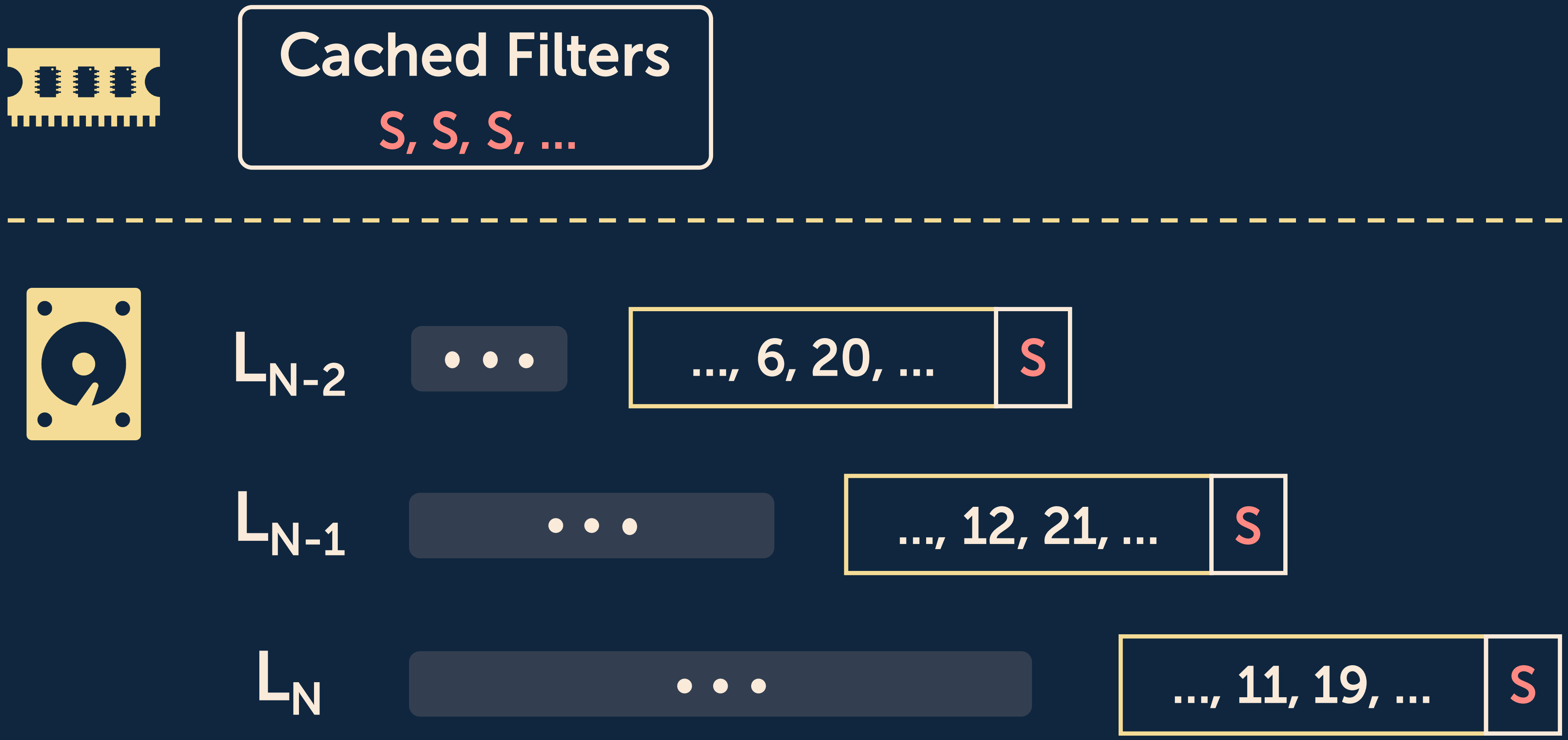


Bloom filters can't help range queries in RocksDB

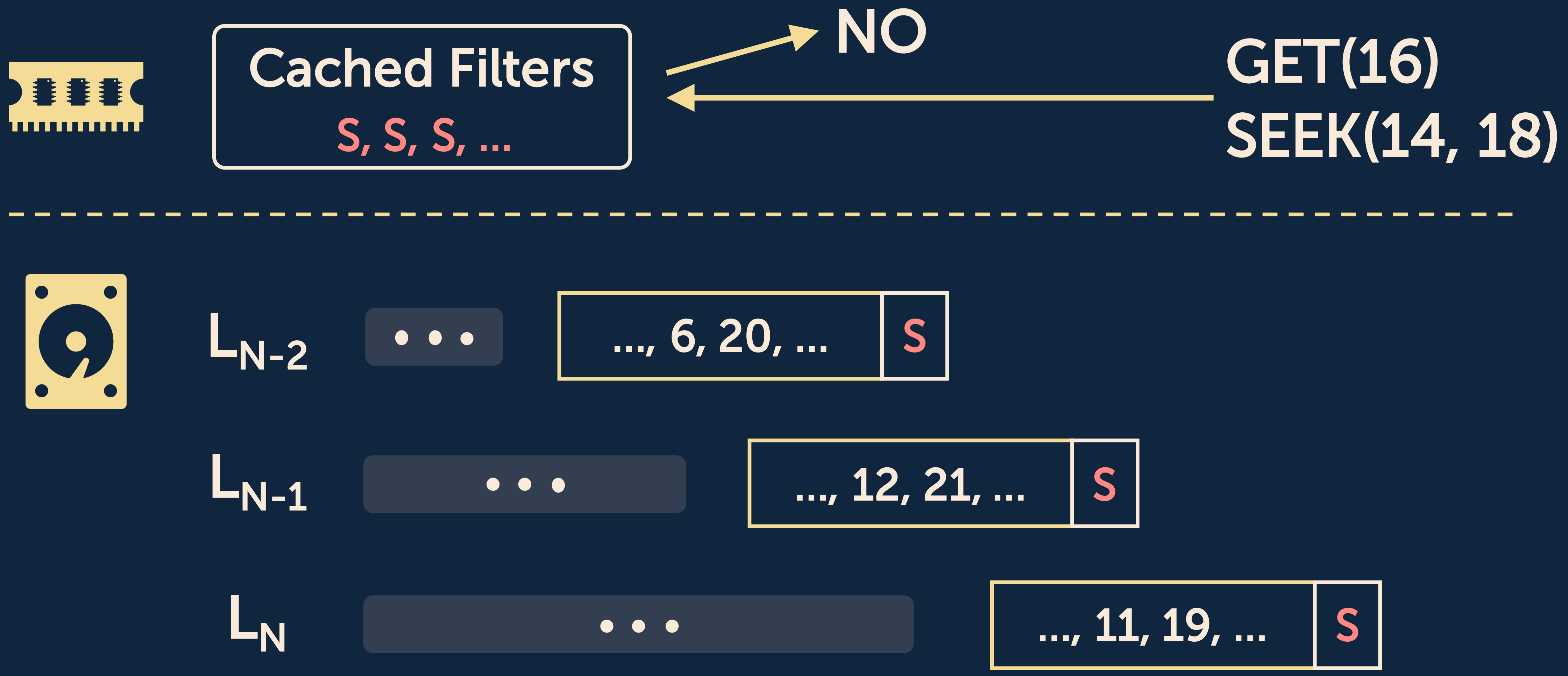
Bloom filters can't help range queries in RocksDB



SuRFs can benefit both point and range queries



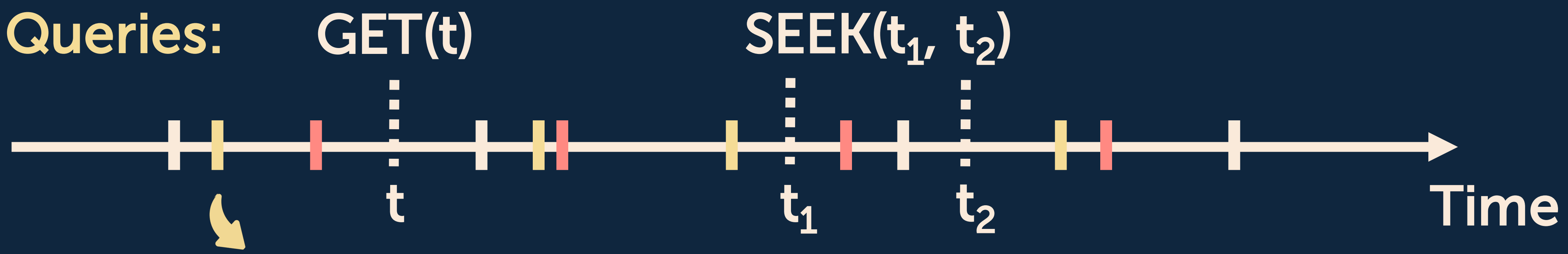
SuRFs can benefit both point and range queries



Evaluation setup: a time-series benchmark

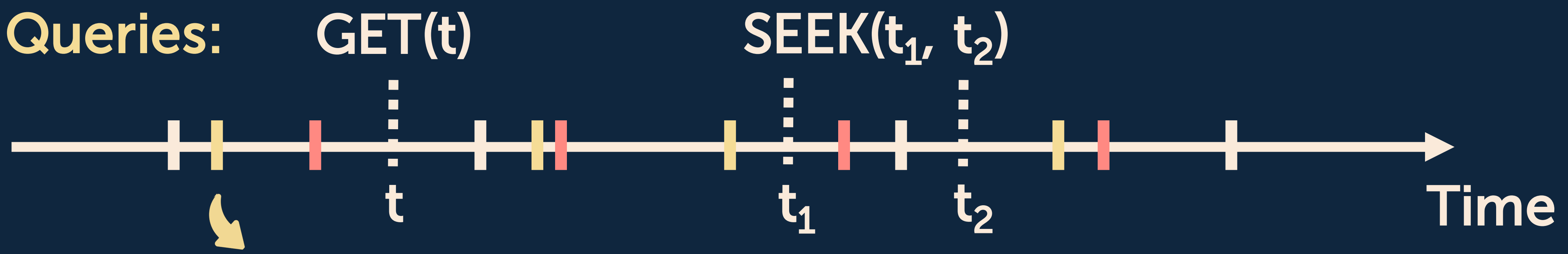
Key: 64-bit timestamp + 64-bit sensor ID
Value: 1KB payload

Evaluation setup: a time-series benchmark



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

Evaluation setup: a time-series benchmark



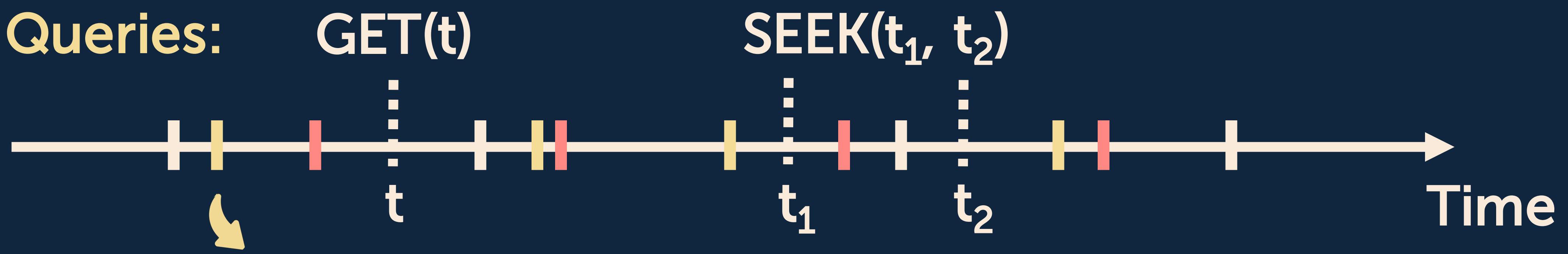
Key: 64-bit timestamp + 64-bit sensor ID
Value: 1KB payload

System Config

Dataset: ≈100 GB on SSD

DRAM: 32 GB

Evaluation setup: a time-series benchmark



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

System Config

Dataset: ≈100 GB on SSD

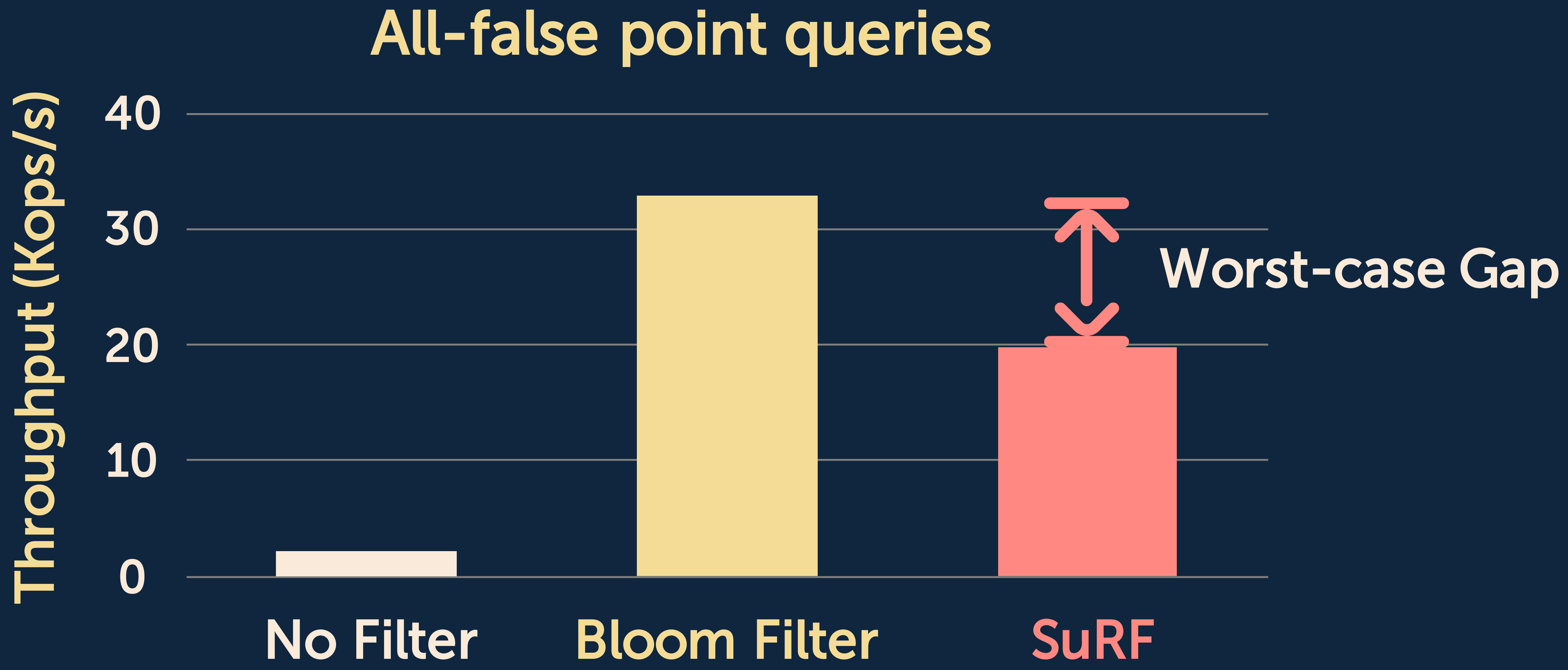
DRAM: 32 GB

Filter Config

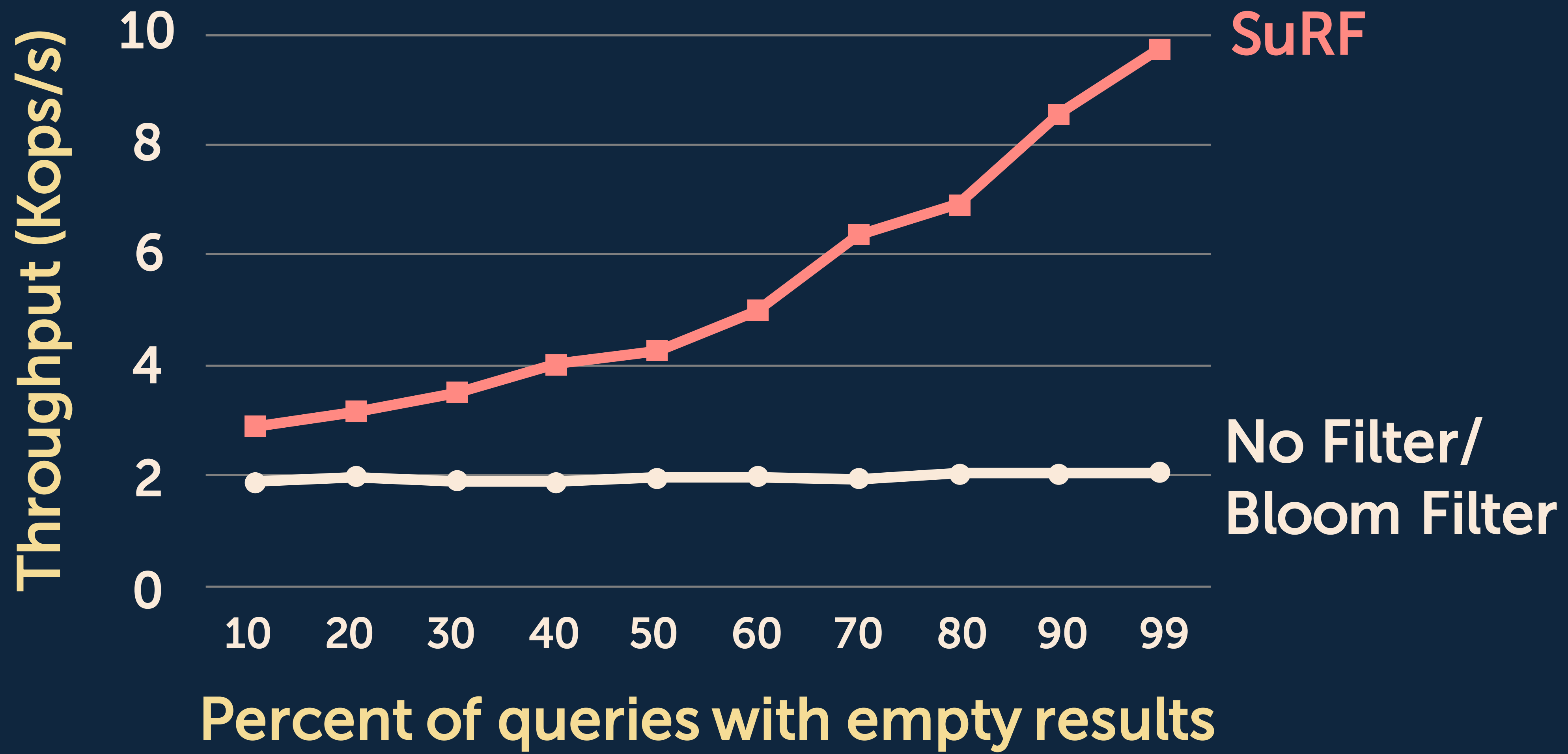
Bloom filter: 14 bits per key

SuRF: 4-bit real suffix

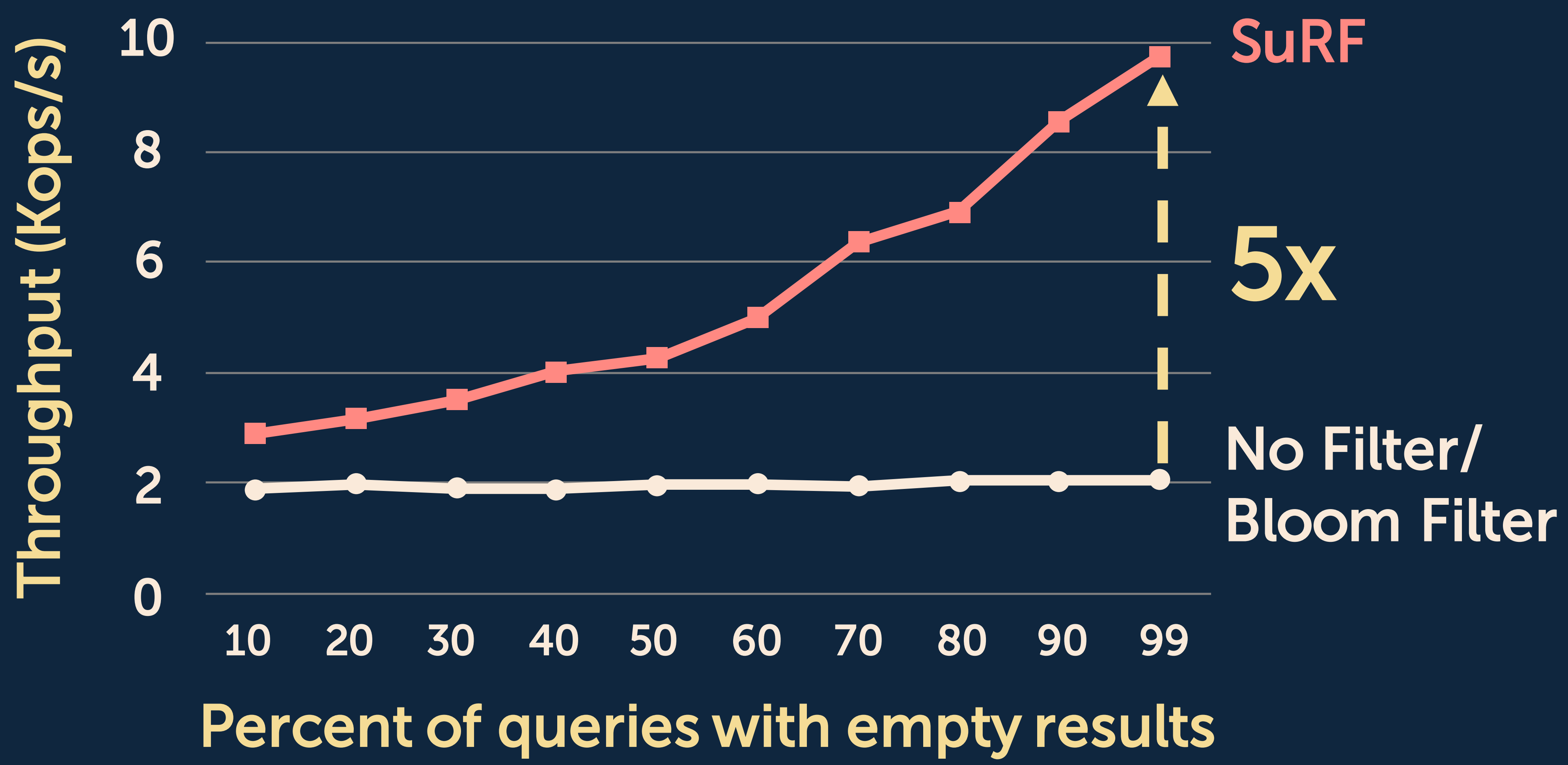
SuRFs still benefit point queries in RocksDB



SuRFs speed up range queries in RocksDB



SuRFs speed up range queries in RocksDB



Conclusion

SuRF is a fast and compact data structure optimized for range filtering

github.com/efficient/SuRF

[Demo] rangefilter.io

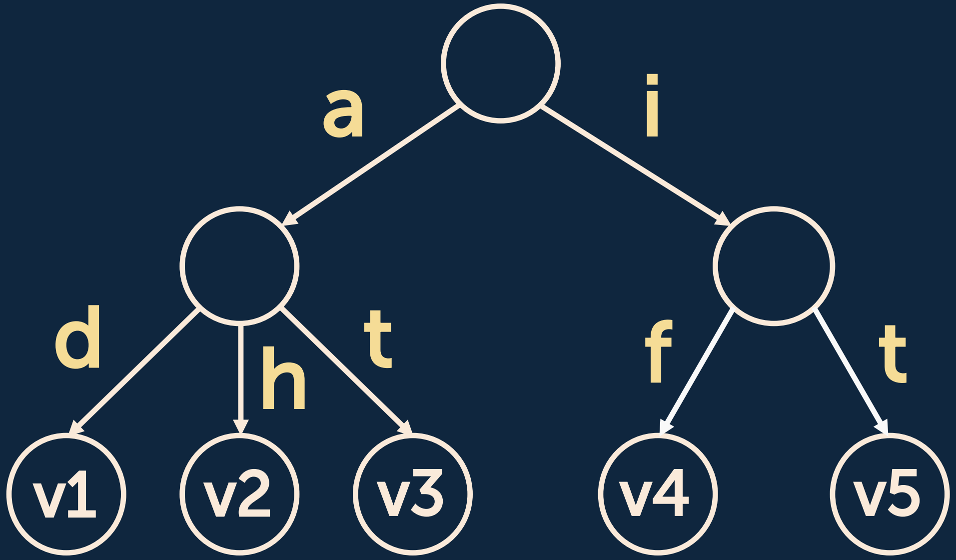
Backup Slides

Comparing ARF to SuRF

Experiment: insert 5M 64-bit integers, 10M Zipf-distributed range queries (ARF uses 2M queries for training)

	ARF	SuRF	Improvement
Bits per Key (held constant)	14	14	-
Range Query Throughput (Mops/s)	0.16	3.3	20x
False Positive Rate	25.7	2.2	12x
Build Time (s)	118	1.2	98x
Build Memory (GB)	26	0.02	1300x
Training Time (s)	117	N/A	N/A
Training Throughput (Mops/s)	0.02	N/A	N/A

LOUDS-Sparse encoding example



LOUDS-Sparse

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

Theoretic Limit ≈ 9.4N bits

10N bits

moveToChild (p) = select(S, rank(HC, p) + 1)

LOUDS-DS trades small space for performance

< 1% space overhead

3x speed-up

B3