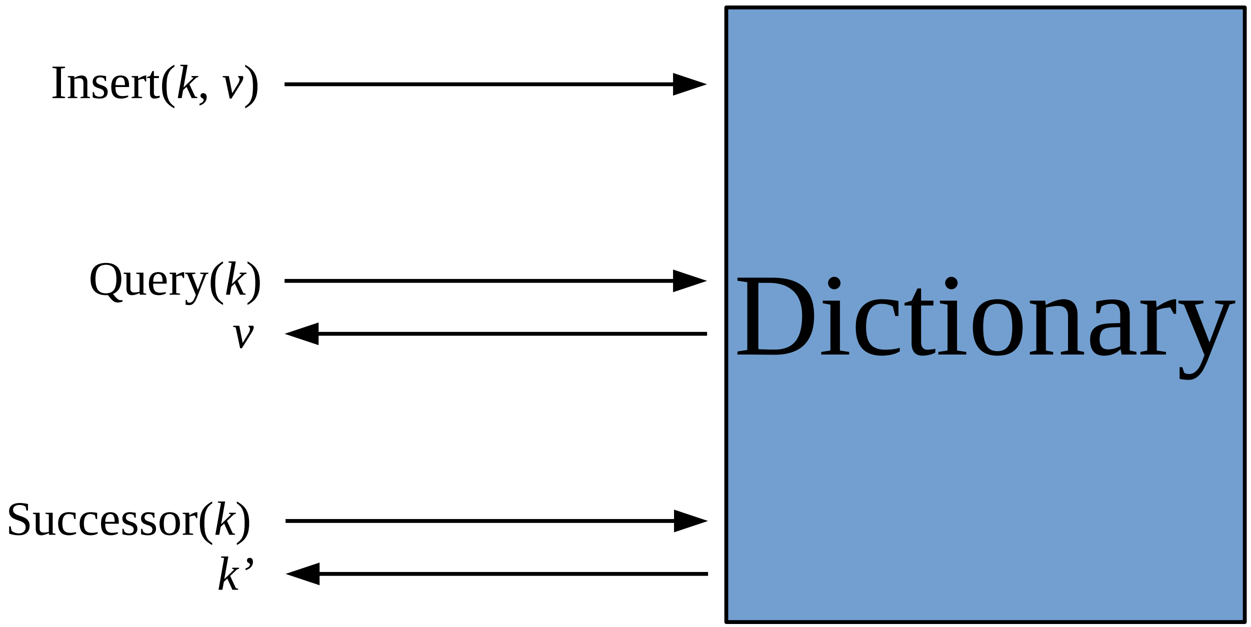
Successor Queries in Optimal External-Memory Dictionaries

Rob Johnson
VMware Research Group

What is a dictionary?



Insert(k, v)

Query(k)

v

Successor(k)

k'

Dictionary

Scan(k, L) = return the L successors of k

Dictionary Performance Trade-Offs

Inserts

Point Queries
&
Scans

The research program

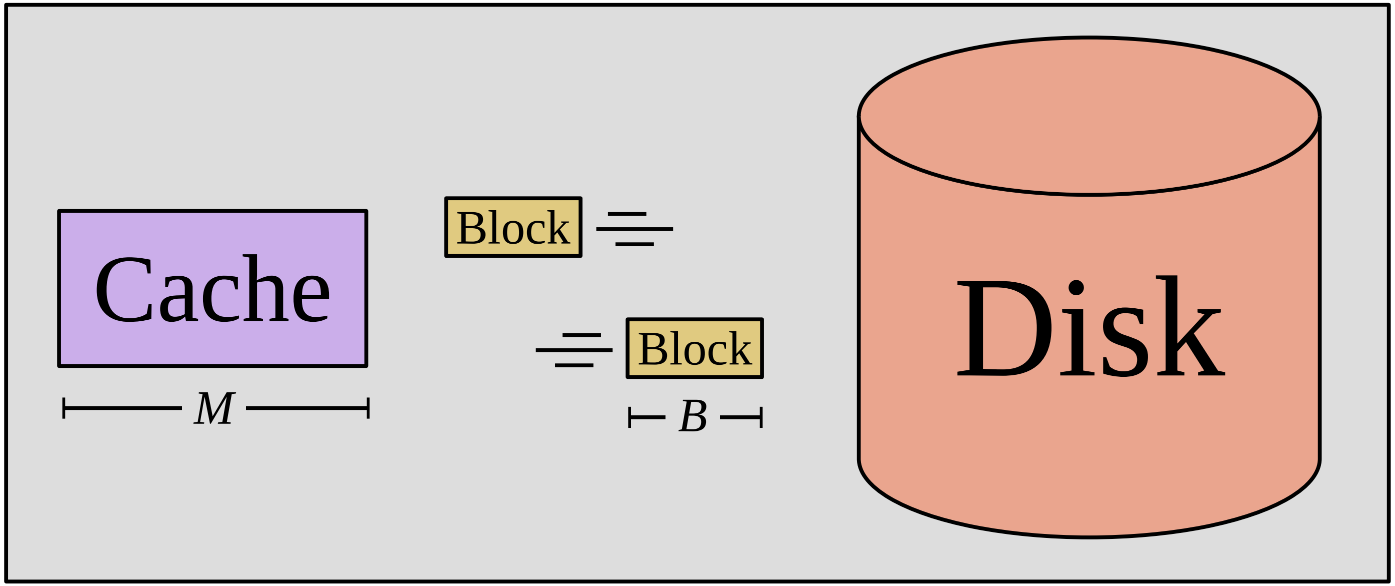
- Make insertions as fast as possible

- While preserving fast point queries

- And “reasonable” successor queries.

The Disk Access Machine (DAM) Model

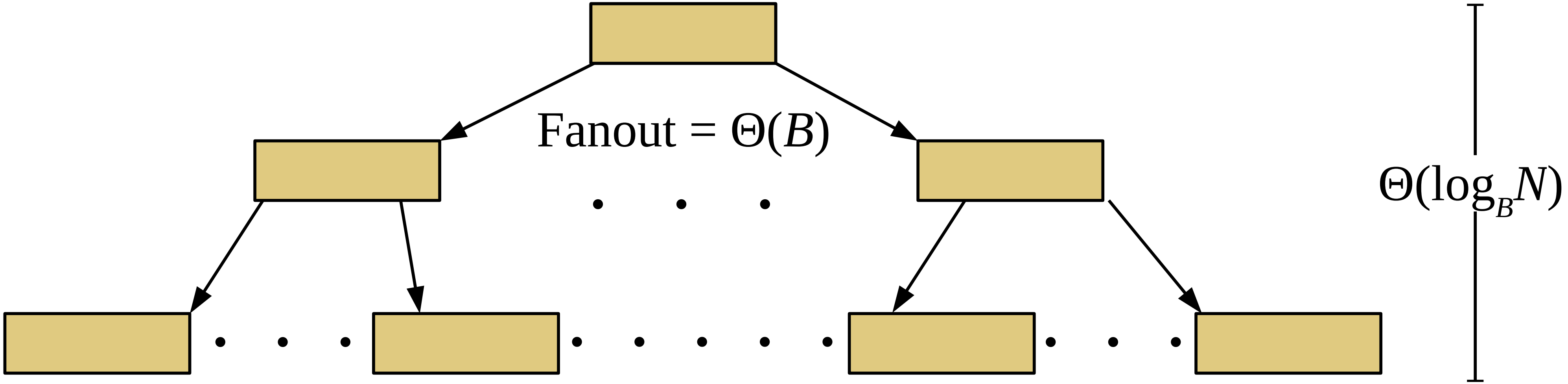
[Aggarwal & Vitter ‘88]



Algorithm design goal: minimize number of block transfers

B-trees were long thought to be optimal

[Bayer & McCreight ‘70]



Insertions
Queries
Successors

$\Big\}$ $\Theta(\log_B N)$ I/Os

Scans: $\Theta(L/B + \log_B N)$ I/Os

The Brodal-Fagerberg bounds

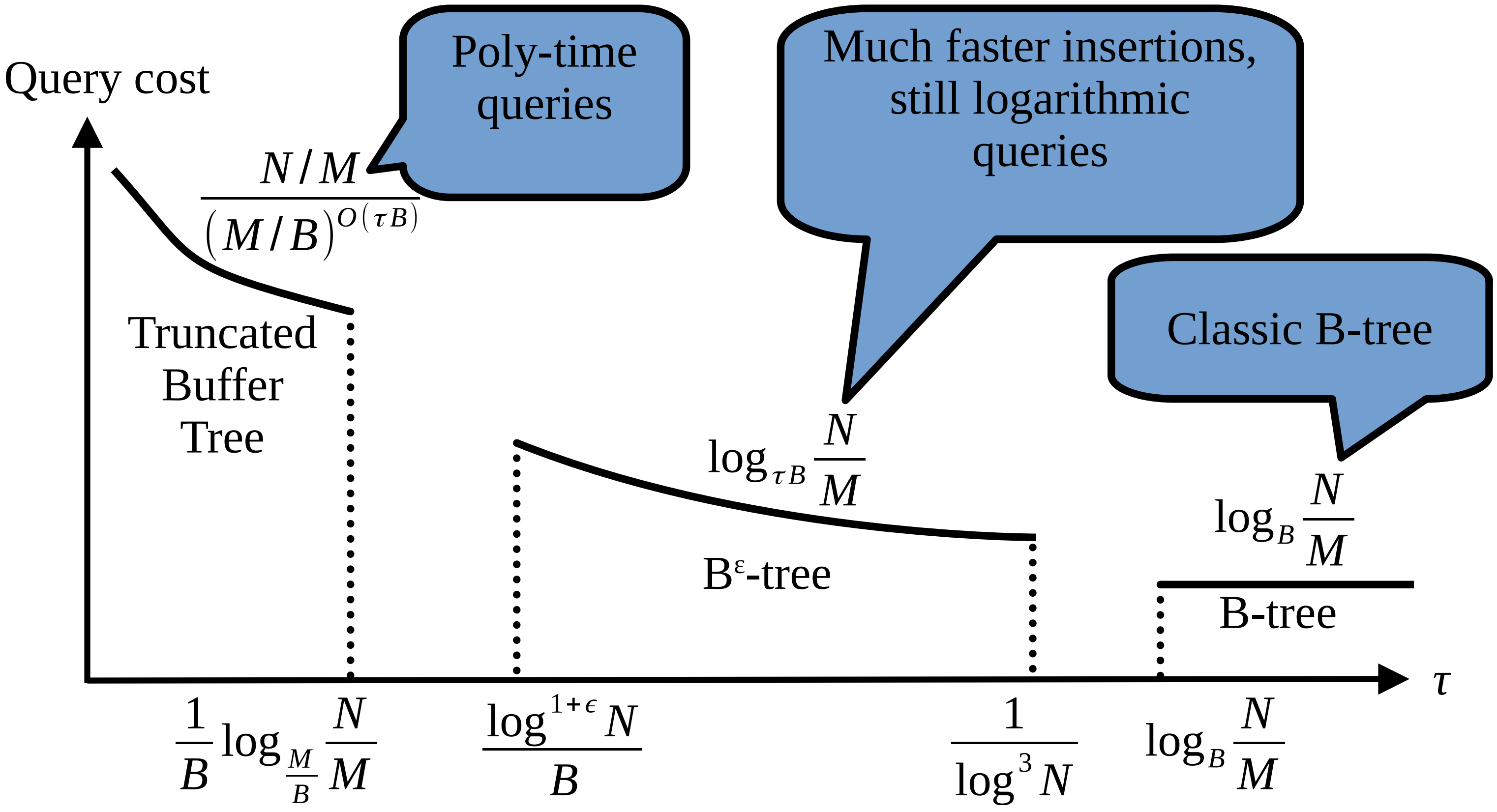
[Brodal & Fagerberg '03]

Atomic key comparison-based bounds

No hashing

Inserts: τ

Queries: $f(\tau)$



*Some conditions apply

The Brodal-Fagerberg bounds

[Brodal & Fagerberg ‘03]

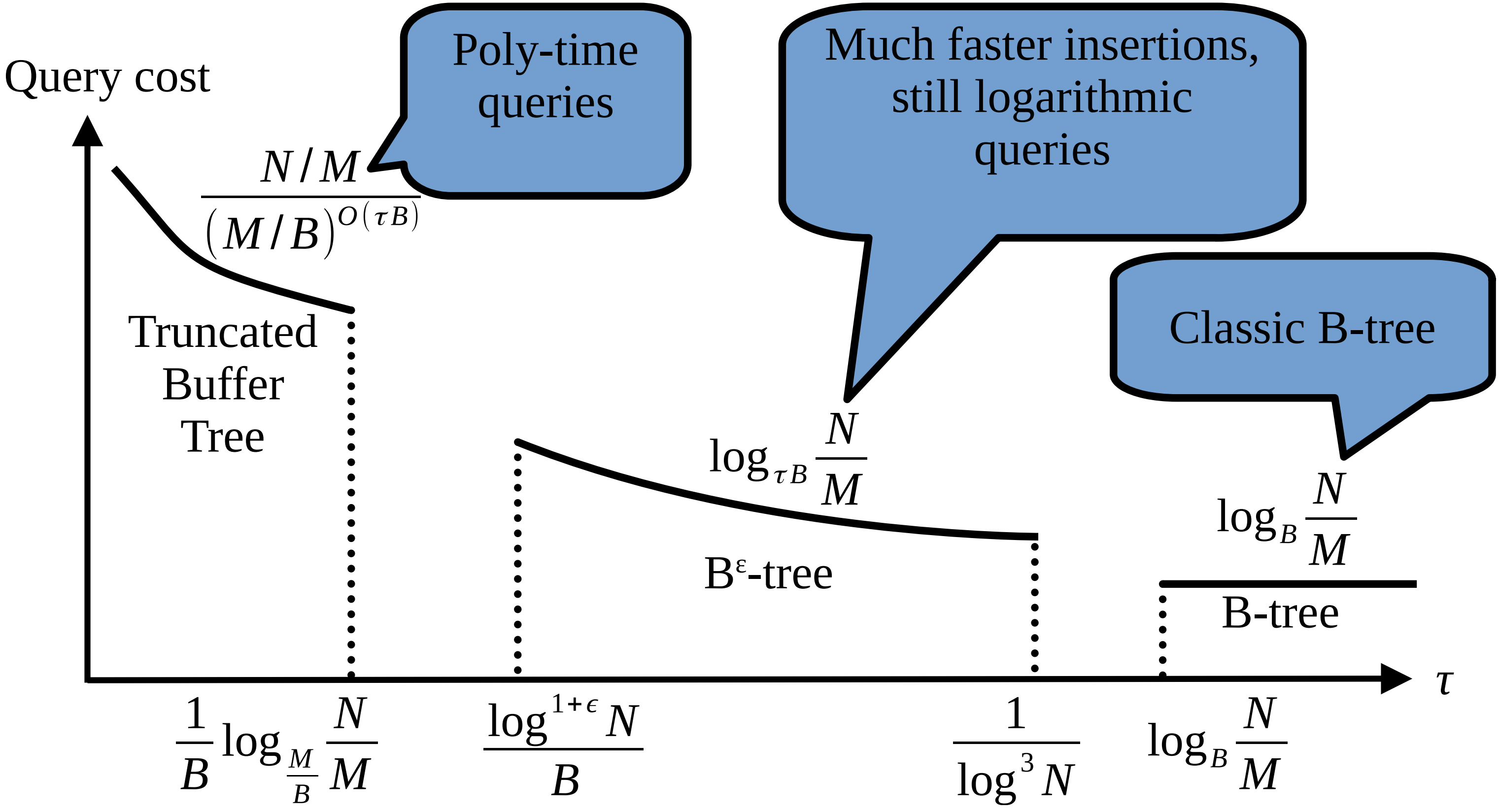
Atomic key comparison-based bounds

No hashing

Inserts: τ

↕

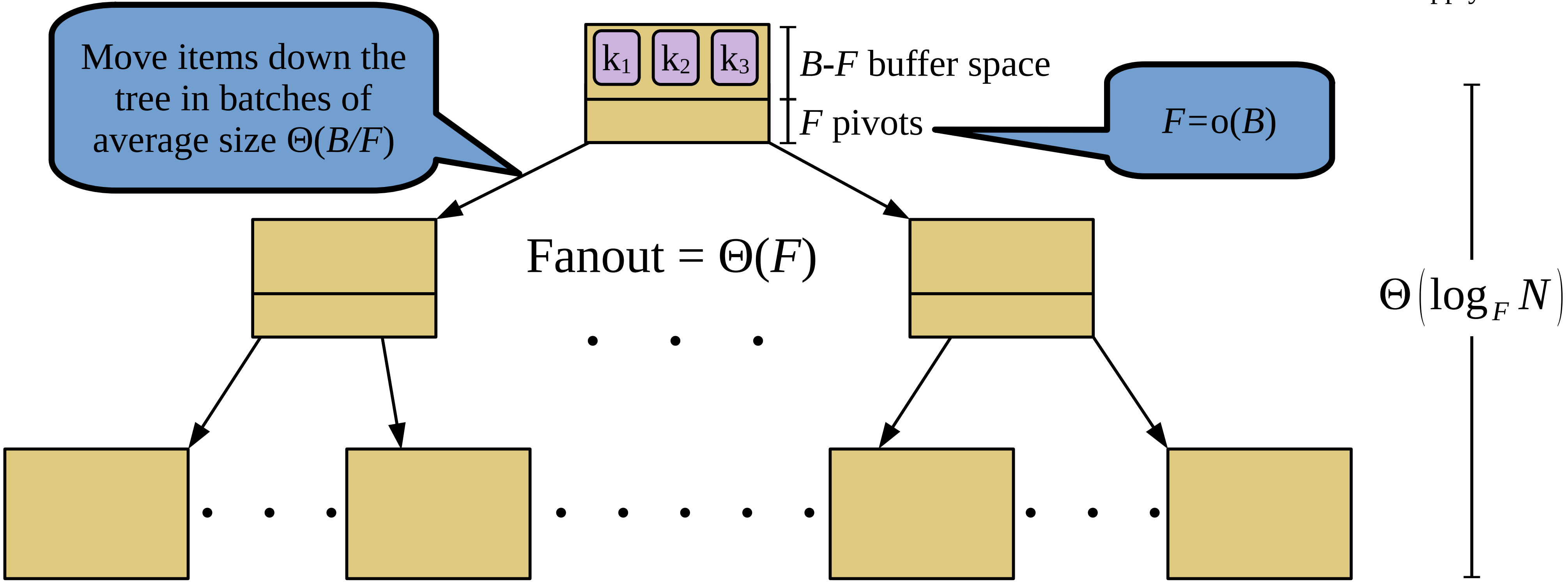
Queries: $f(\tau)$



*Some conditions apply

B^ε-trees meet the Brodal-Fagerberg bound

*Some conditions apply



Move items down the tree in batches of average size $\Theta(B/F)$

$F = o(B)$

Insertions: $\Theta\left(\frac{F \log_F N}{B}\right)$

$\left.\begin{array}{l}\text{Queries}\\ \text{Successors}\end{array}\right\} \Theta\left(\log_F N\right)$

Scans: $\Theta\left(L/B + \log_F N\right)$

B$^{\varepsilon}$-tree asymptotics

Fanout	Insertions	Point queries	Scans
F	$\frac{F \log_F N}{B}$	$\log_F N$	$L/B+\log_F N$
B^{ε}	$\frac{\log_B N}{\epsilon B^{1-\epsilon}}$	$\frac{\log_B N}{\epsilon}$	$L/B+\frac{\log_B N}{\epsilon}$
$\sqrt{B}$	$\frac{\log_B N}{\sqrt{B}}$	$\log_B N$	$L/B+\log_B N$
$\frac{\tau B}{\log N}$	τ	$\log_{\tau B} N$	$L/B+\log_{\tau B} N$

Assuming $\tau \geq \frac{\log^{1+\epsilon} N}{B}$

The Iacono-Pătrașcu bounds

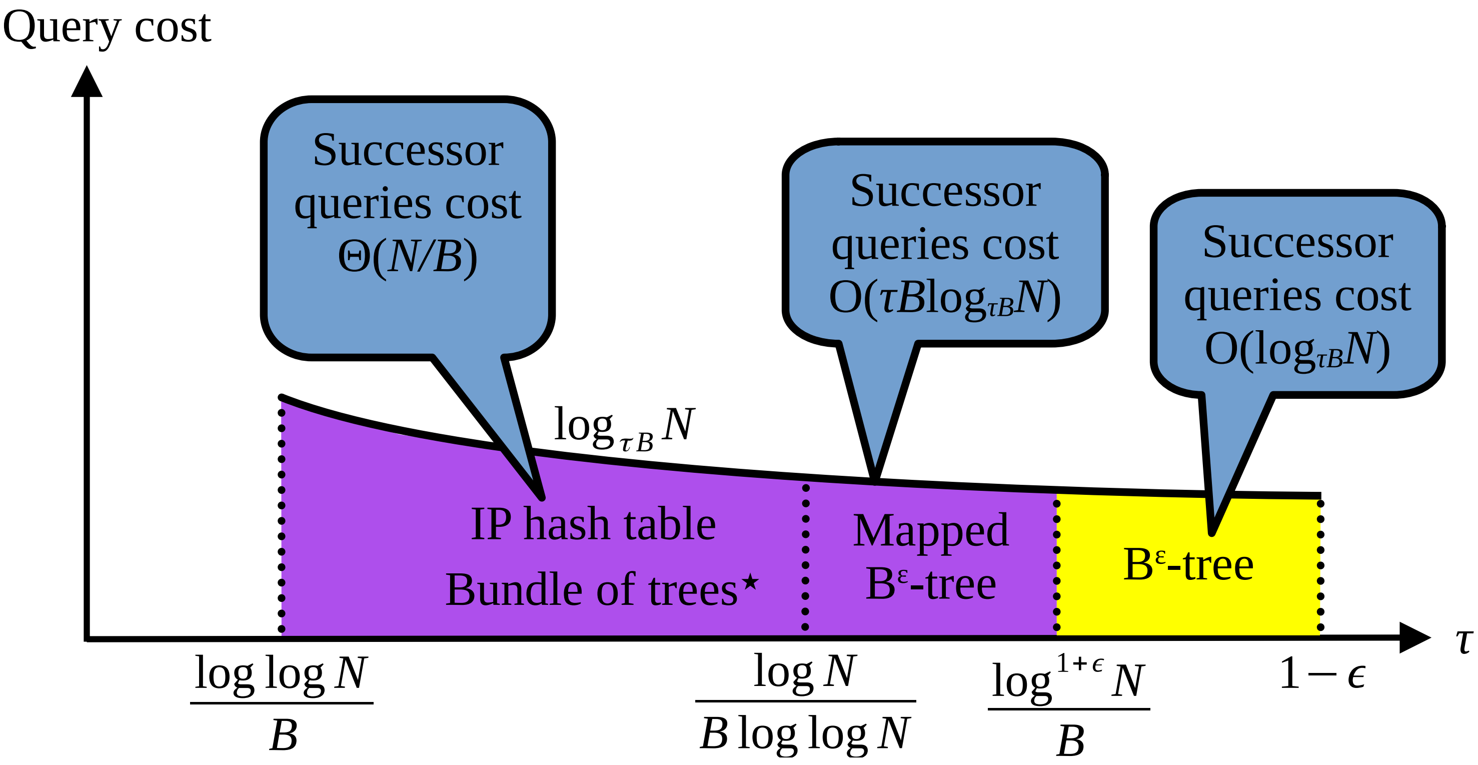
[Iacono & Pătrașcu '11]

Non-atomic key model

Hashing allowed

Inserts: τ

Queries: $\log_{\tau B} N$



⋆[Conway, Farach-Colton, Shilane '18]

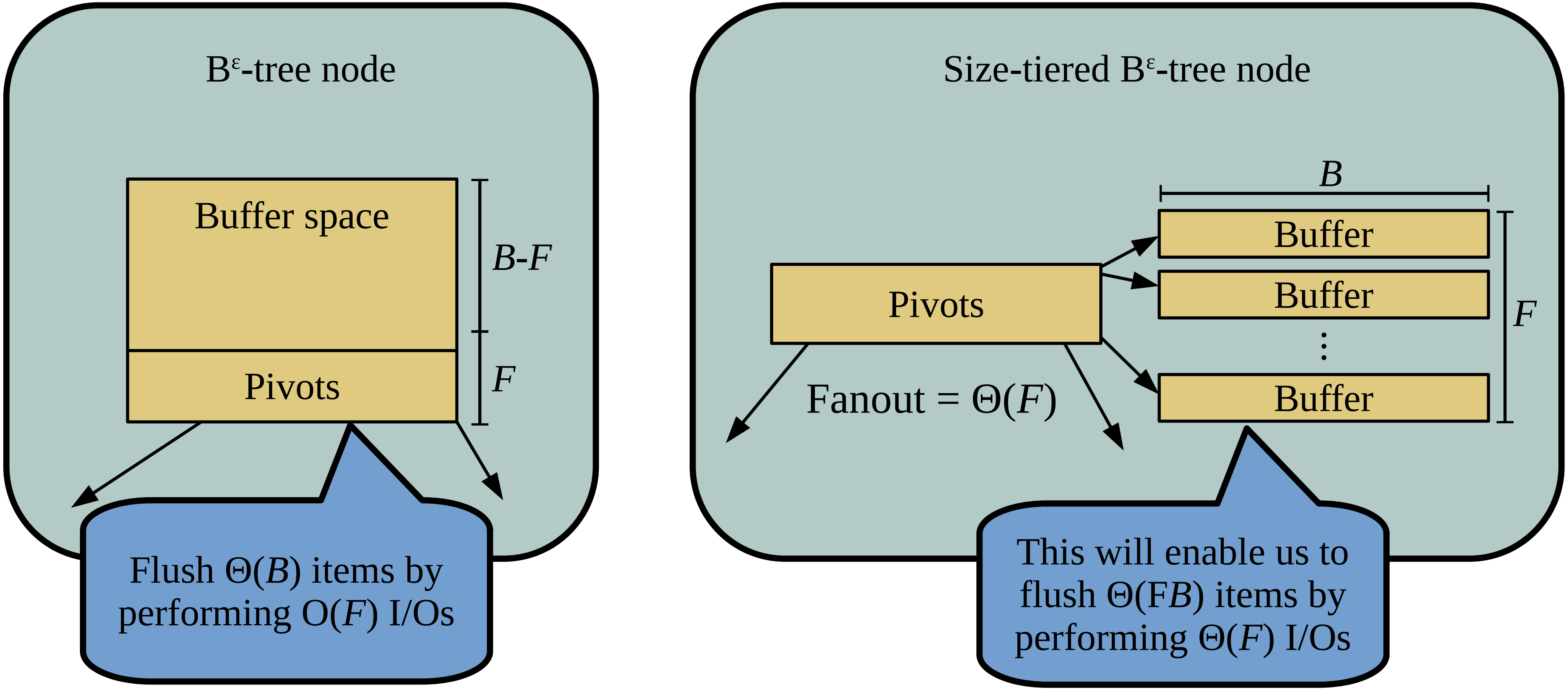
*Some conditions apply

Size-tiered B^{ε}-trees

Faster inserts, slower queries

[Jagadish, Narayan, Seshadri, Sudarshan, Kanneganti ‘97]

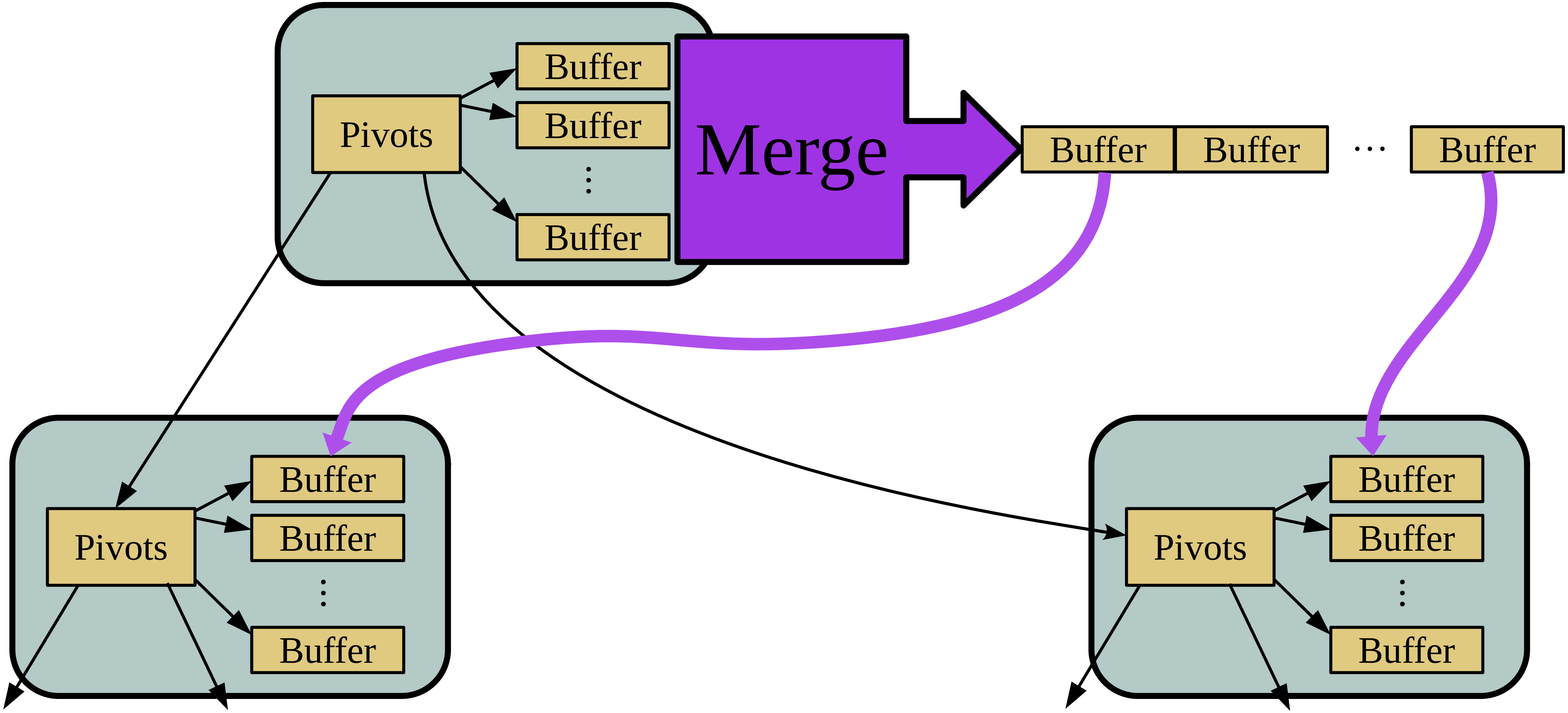
Size-tiered B$^{\varepsilon}$-tree nodes



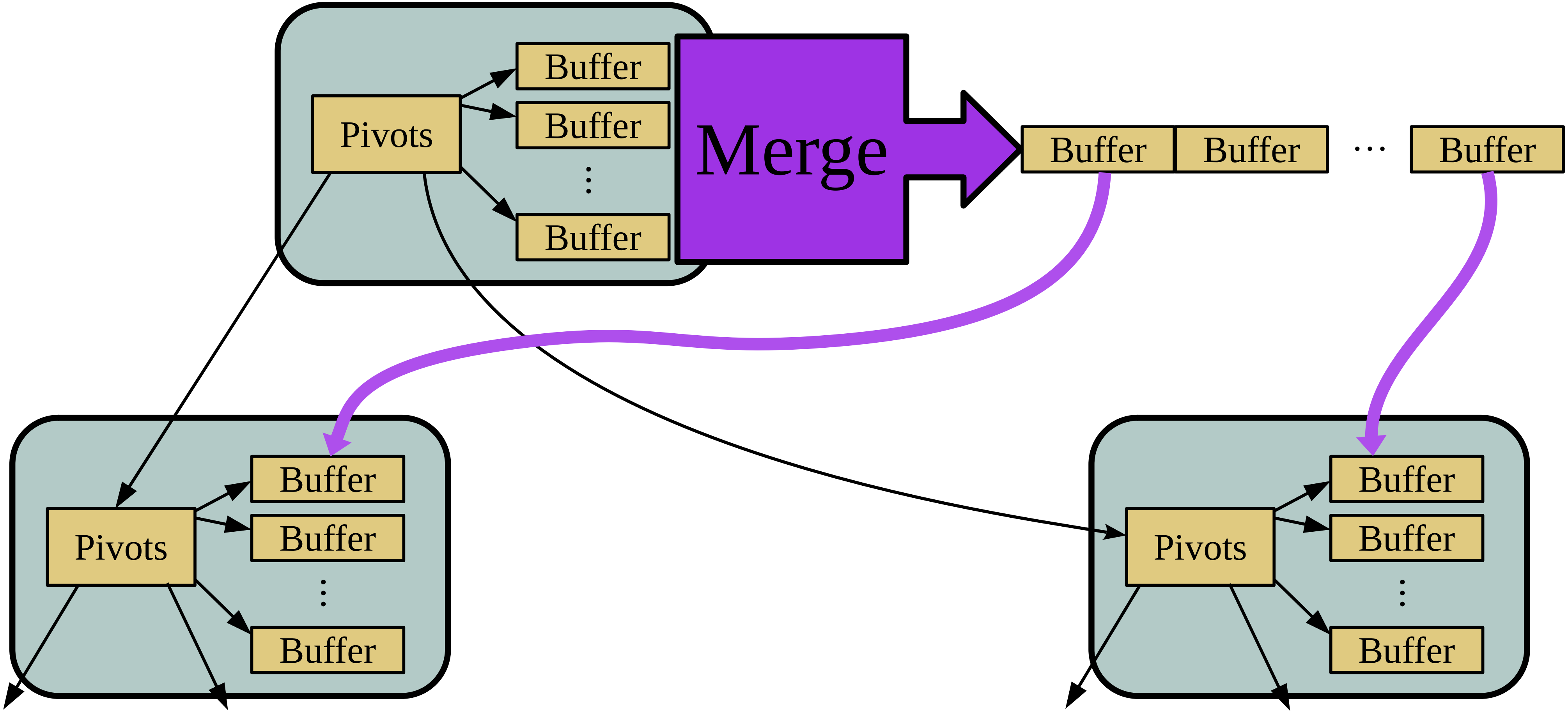
Flush $\Theta(B)$ items by performing O(F) I/Os

This will enable us to flush $\Theta(\text{F}B)$ items by performing $\Theta(F)$ I/Os

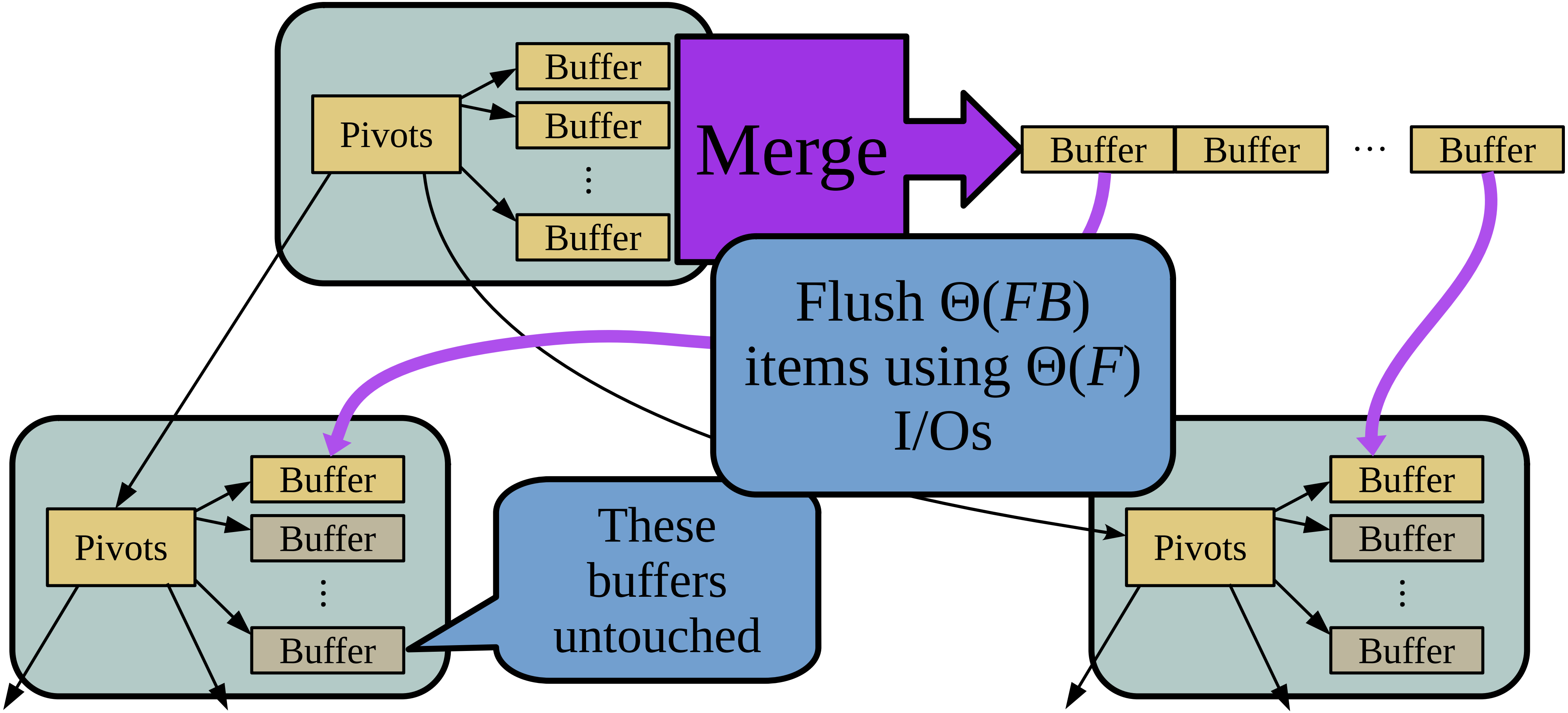
Flushes in size-tiered B$^\varepsilon$-trees



Analysis: Flushes in size-tiered B$^{\varepsilon}$-trees



Analysis: Flushes in size-tiered B$^{\varepsilon}$-trees



Flush $\Theta(FB)$ items using $\Theta(F)$ I/Os

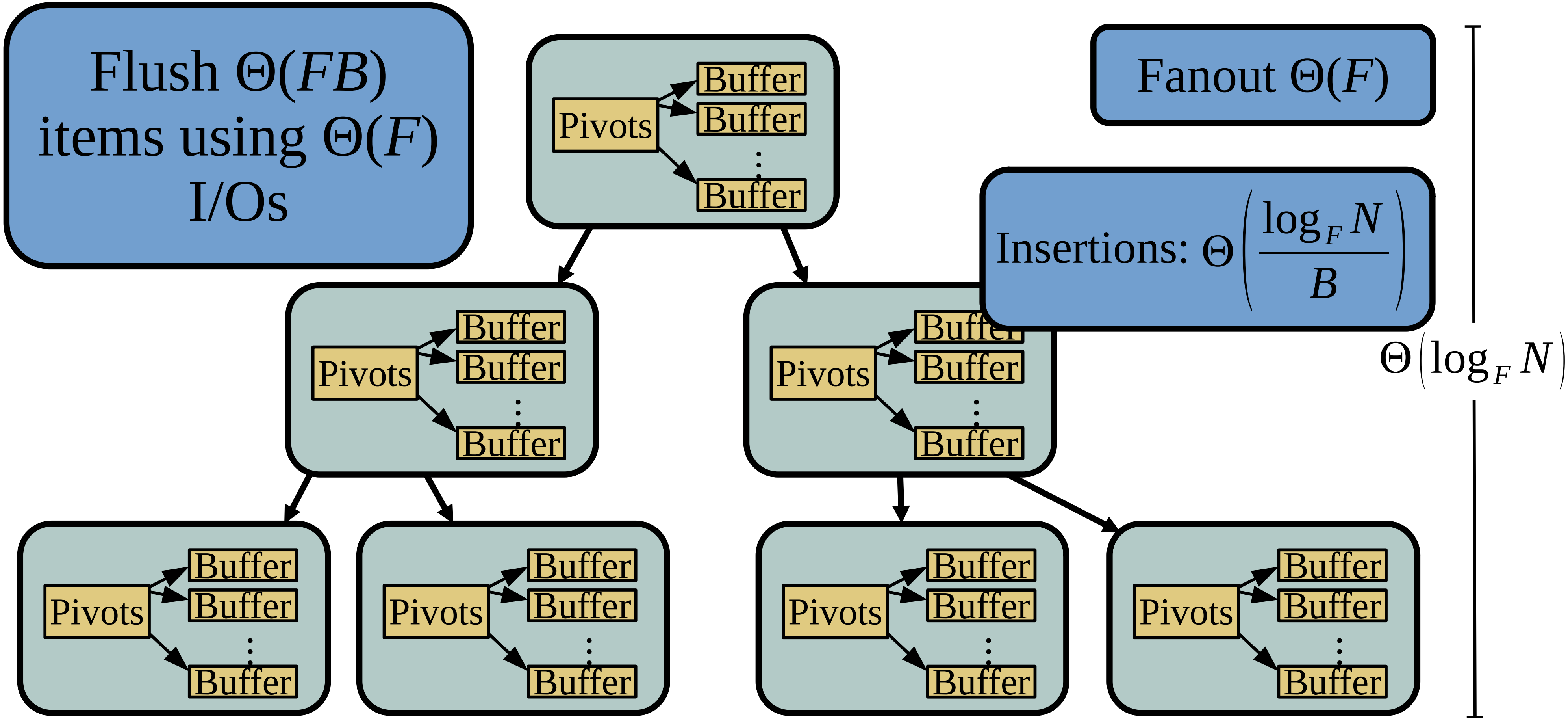
These buffers untouched

Analysis: Insertions in size-tiered B$^{\varepsilon}$-trees

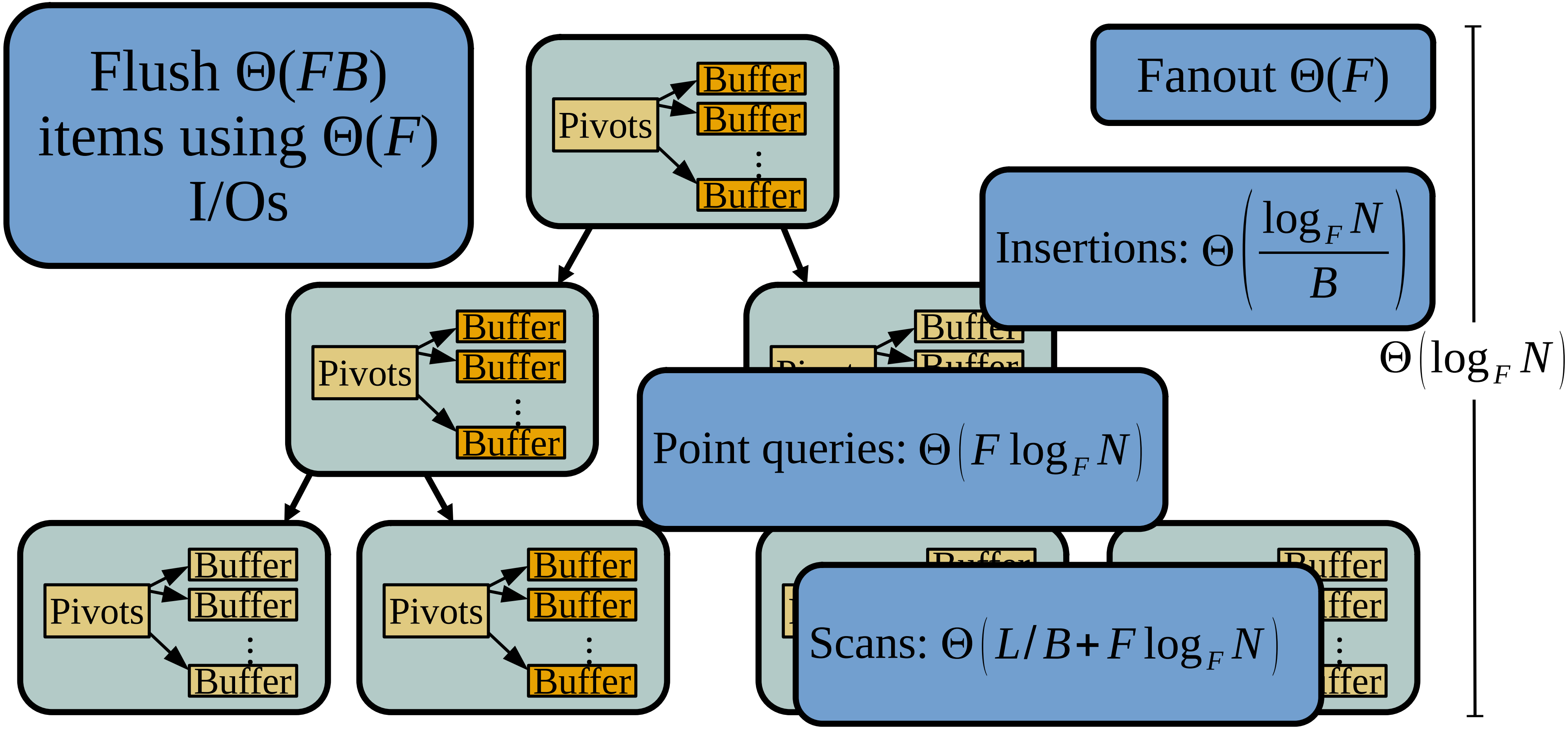
Flush $\Theta(FB)$ items using $\Theta(F)$ I/Os

Fanout $\Theta(F)$

Insertions: $\Theta\left(\frac{\log_F N}{B}\right)$



Analysis: I/O costs in size-tiered B^ε-trees



Flush $\Theta(FB)$ items using $\Theta(F)$ I/Os

Fanout $\Theta(F)$

Insertions: $\Theta\left(\frac{\log_F N}{B}\right)$

Point queries: $\Theta\left(F \log_F N\right)$

Scans: $\Theta\left(L/B + F \log_F N\right)$

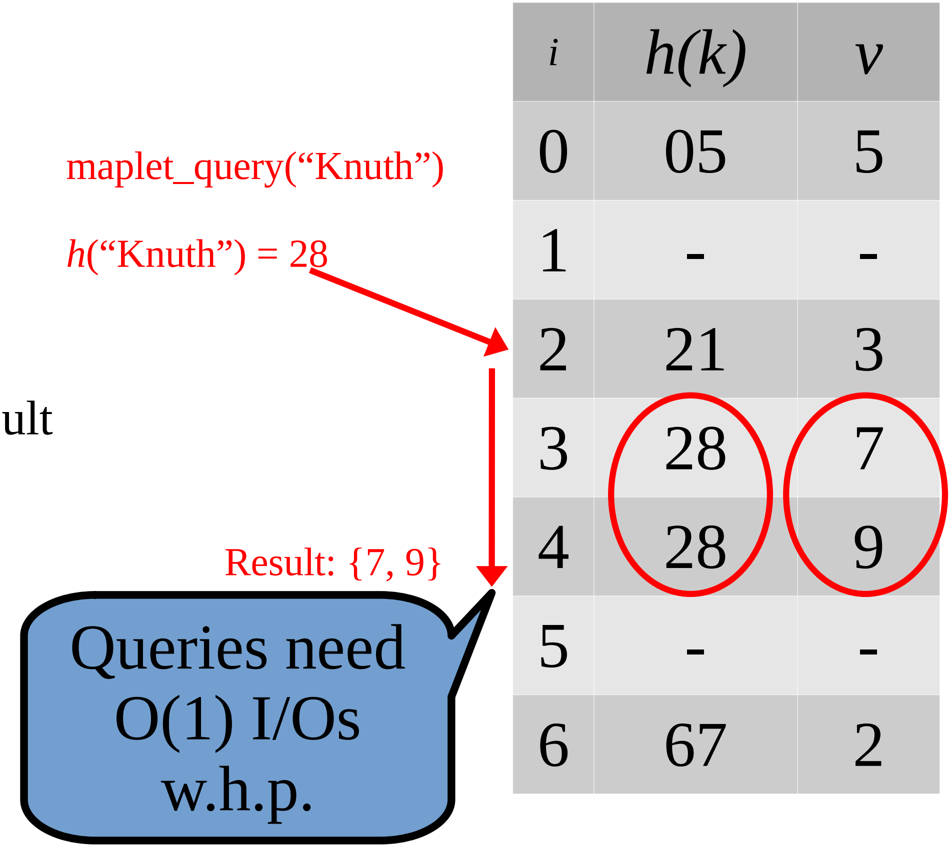
Maplets and mapped B$^\varepsilon$-trees

Fixing queries

[Conway, Farach-Colton, Johnson 2023]

Maplets

- Maplets extend filters from **sets** to **maps**
  - maplet_query(k) → { $v_1, v_2, \ldots, v_\ell$ }
- Maplets save space by allowing false positives
  - False positives are extra values in a query result
  - False-positive rate = E[# of extra values]
- Basic implementation:
  - Store a ordered linear-probing hash table of *(h(k), v)* pairs
  - Compress table using ***quotienting*** [Knuth 1973]



maplet_query("Knuth")

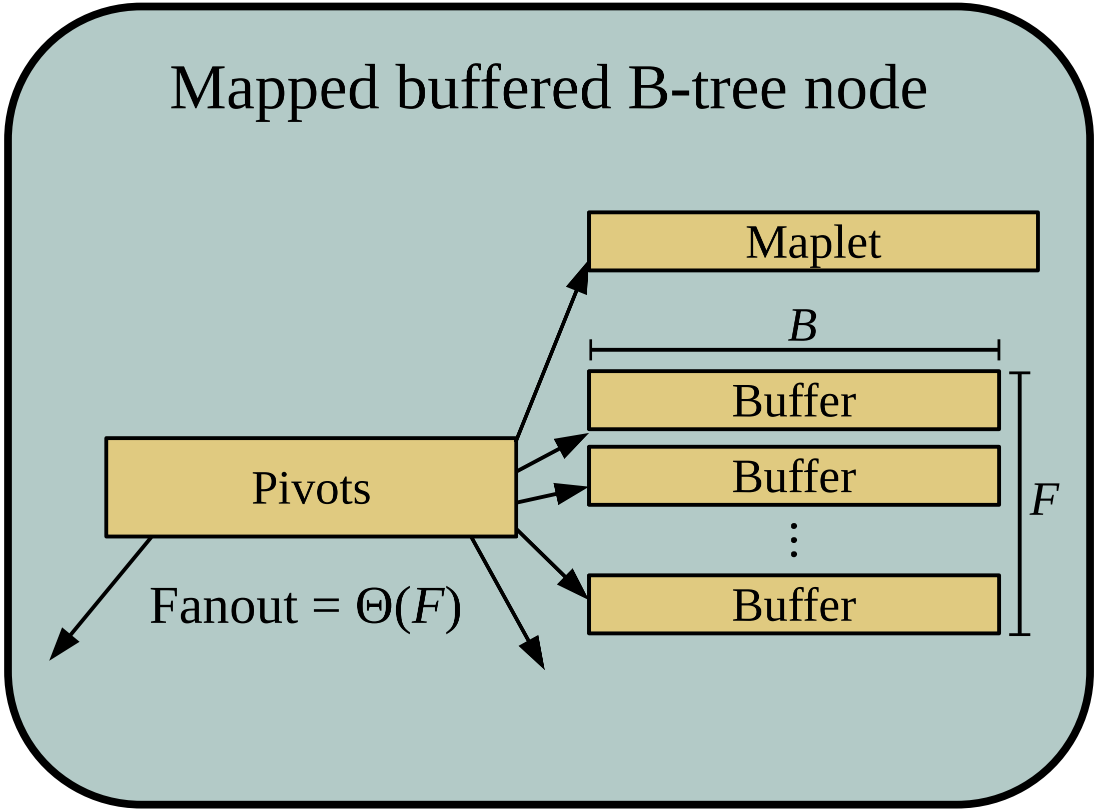
h("Knuth") = 28

Result: {7, 9}

Queries need O(1) I/Os w.h.p.

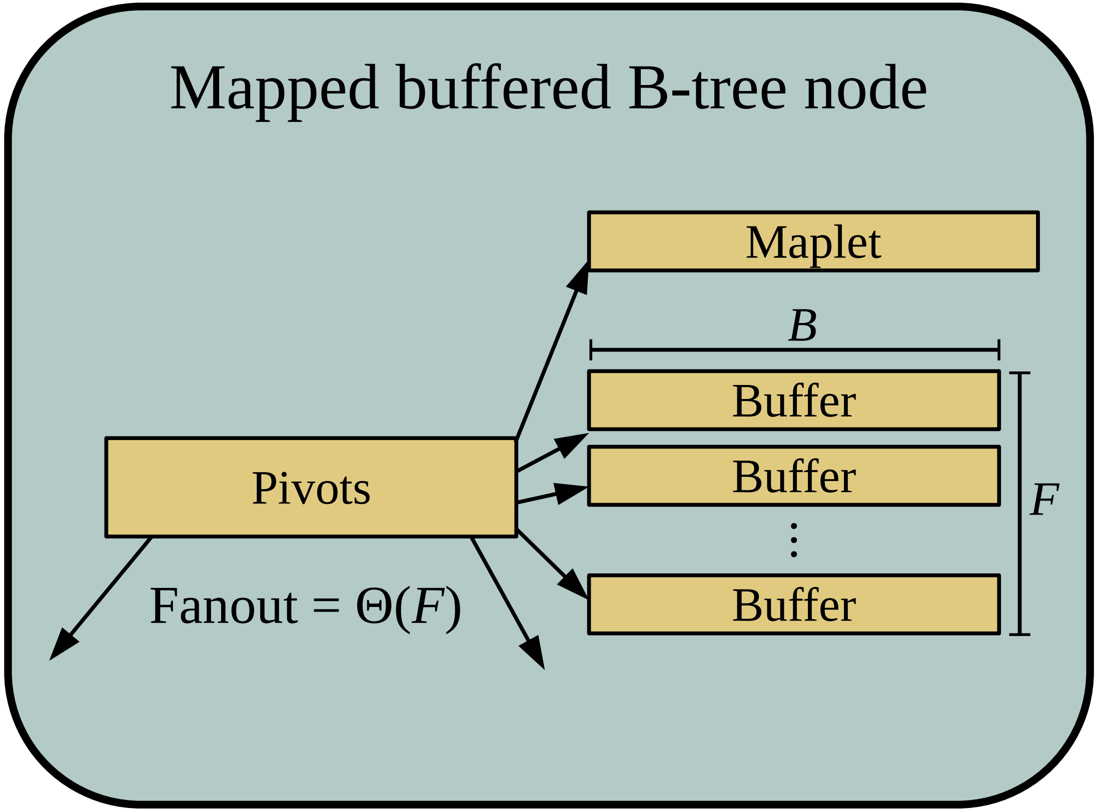
i	*h(k)*	*v*
0	05	5
1	-	-
2	21	3
3	28	7
4	28	9
5	-	-
6	67	2

Mapped B$^{\varepsilon}$-tree nodes



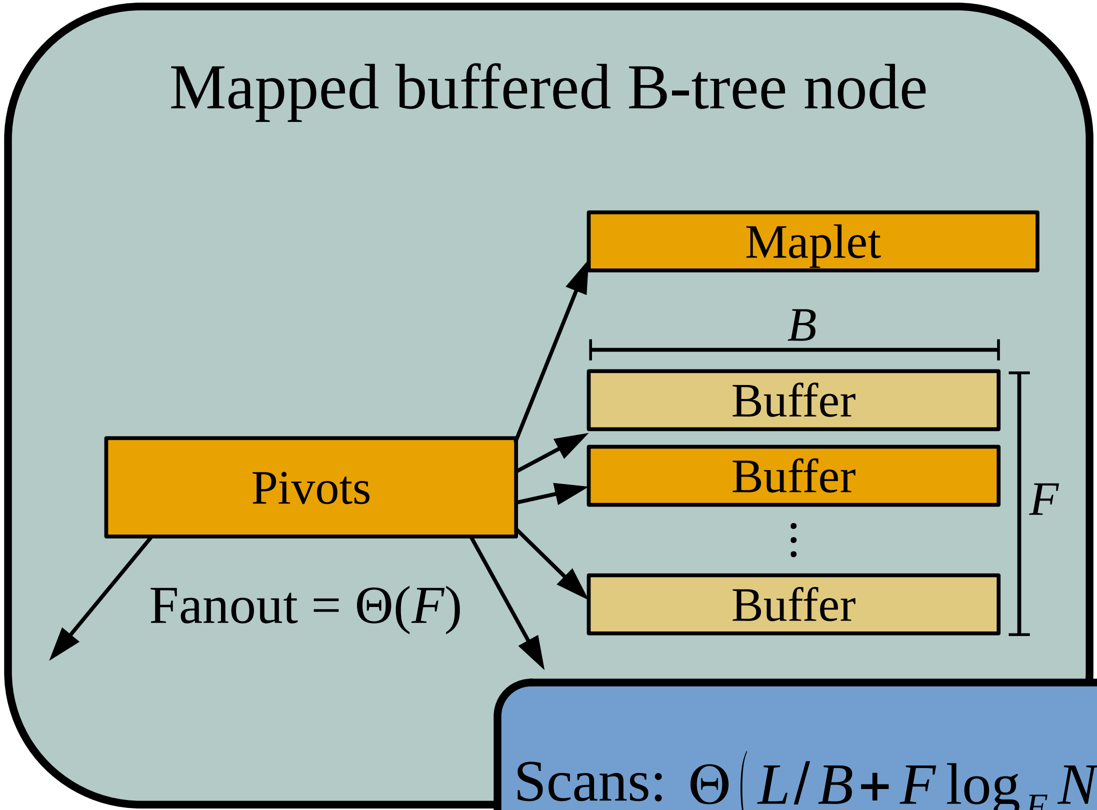
Maplet: $k \rightarrow$ { buffers containing k }

Queries in mapped B$^{\varepsilon}$-trees



Maplet: k → { buffers containing k }

Queries in mapped B$^{\varepsilon}$-trees



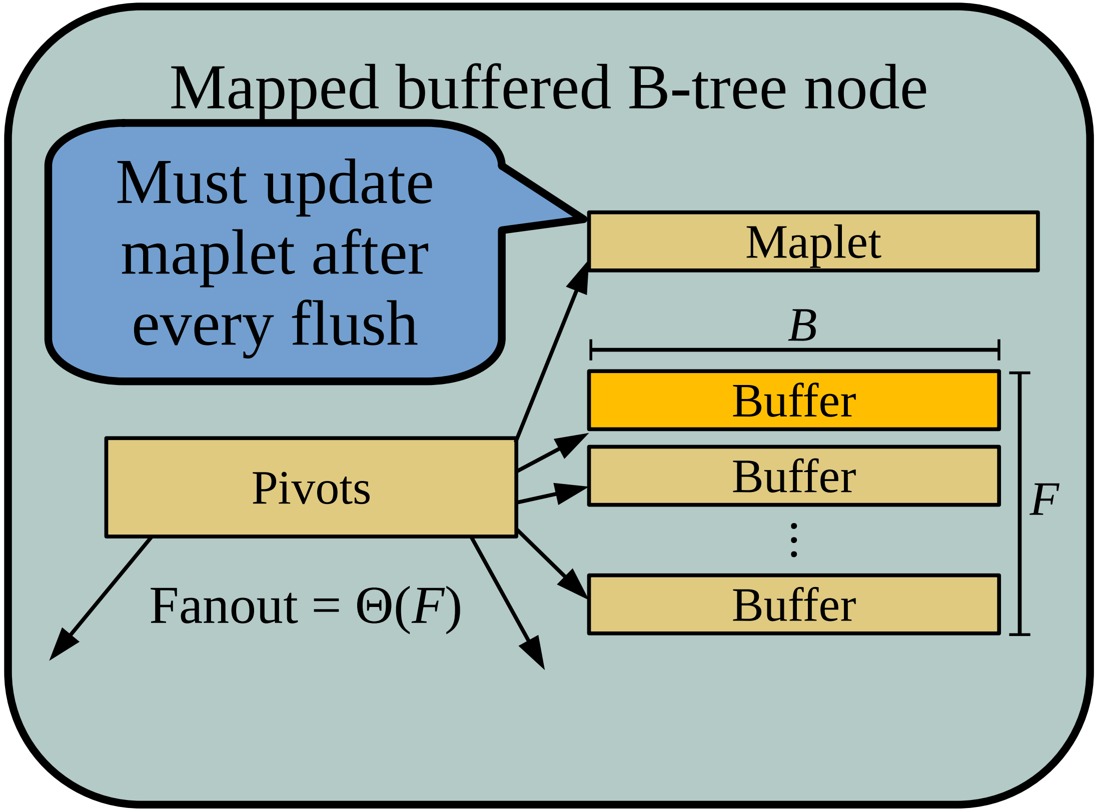
Maplet: $k \rightarrow$ { buffers containing k }

Queries access O(1) blocks in each node

Point queries: $\Theta\left(\log_F N\right)$

Scans: $\Theta\left(L/B+F\log_F N\right)$

Flushes in mapped B$^{\varepsilon}$-trees

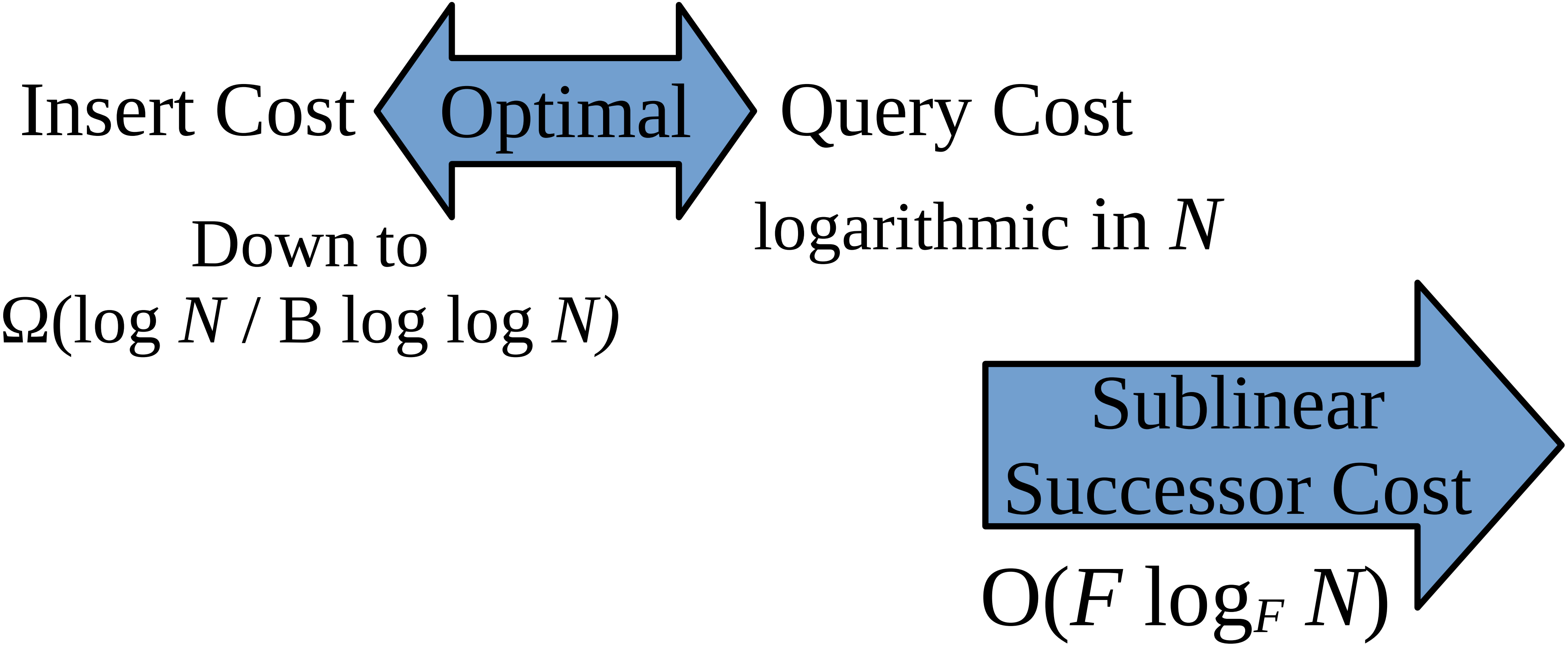

Mapped buffered B-tree node
Must update maplet after every flush
Maplet
B
Buffer
Buffer
⋮
Buffer
F
Pivots
Fanout = $\Theta(F)$

Maplet: $k \rightarrow$ { buffers containing k }

Maplet maintenance not a bottleneck to optimality or in practice

Theorem: Mapped B$^{\varepsilon}$-tree meets IP lower bound when $F = \Omega(\log N / \log \log N)$.

Summary of theoretical results



Insert Cost

Optimal

Query Cost

Down to $\Omega(\log N / \mathrm{B} \log \log N)$

logarithmic in N

Sublinear Successor Cost

$\mathrm{O}(F \log_F N)$

Empirical performance measurements

PostgreSQL

B-tree

SPLINTERDB

Mapped B^{ε}-tree

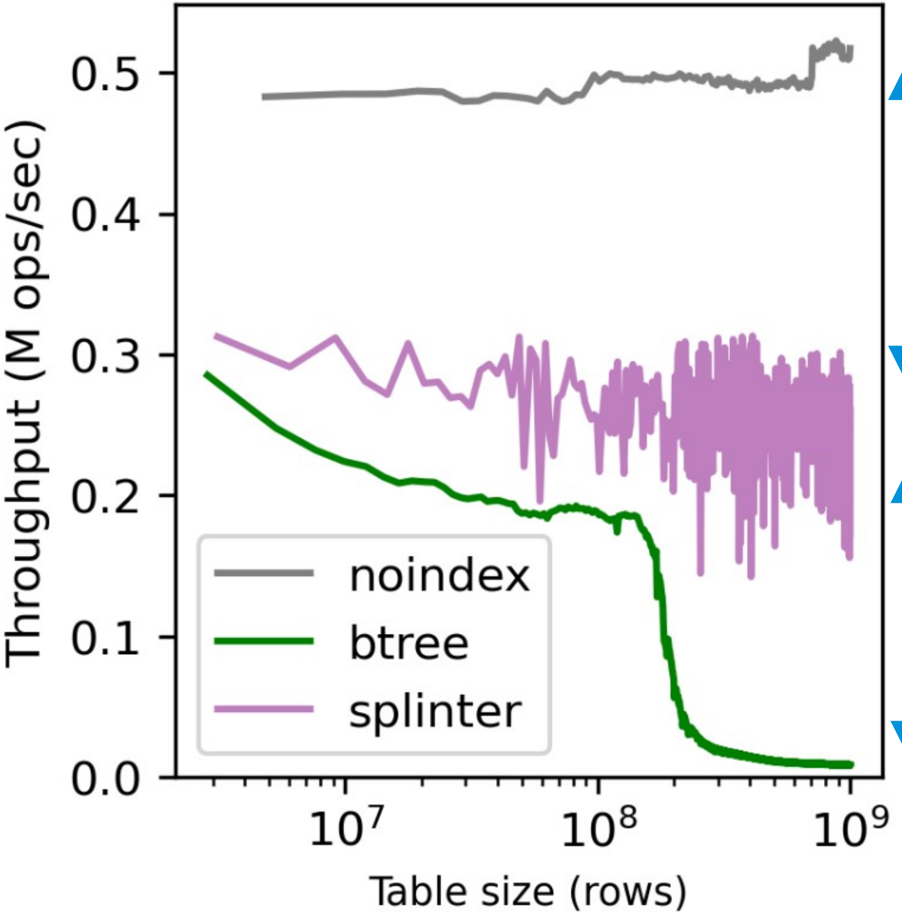
Random inserts

AWS i4i.16xlarge:
64 CPUs, fast local storage

1 client inserting

1 billion random rows

Higher throughput is better.



SplinterDB is ~50% slower than no-index at all

SplinterDB throughput fluctuates due to flushes

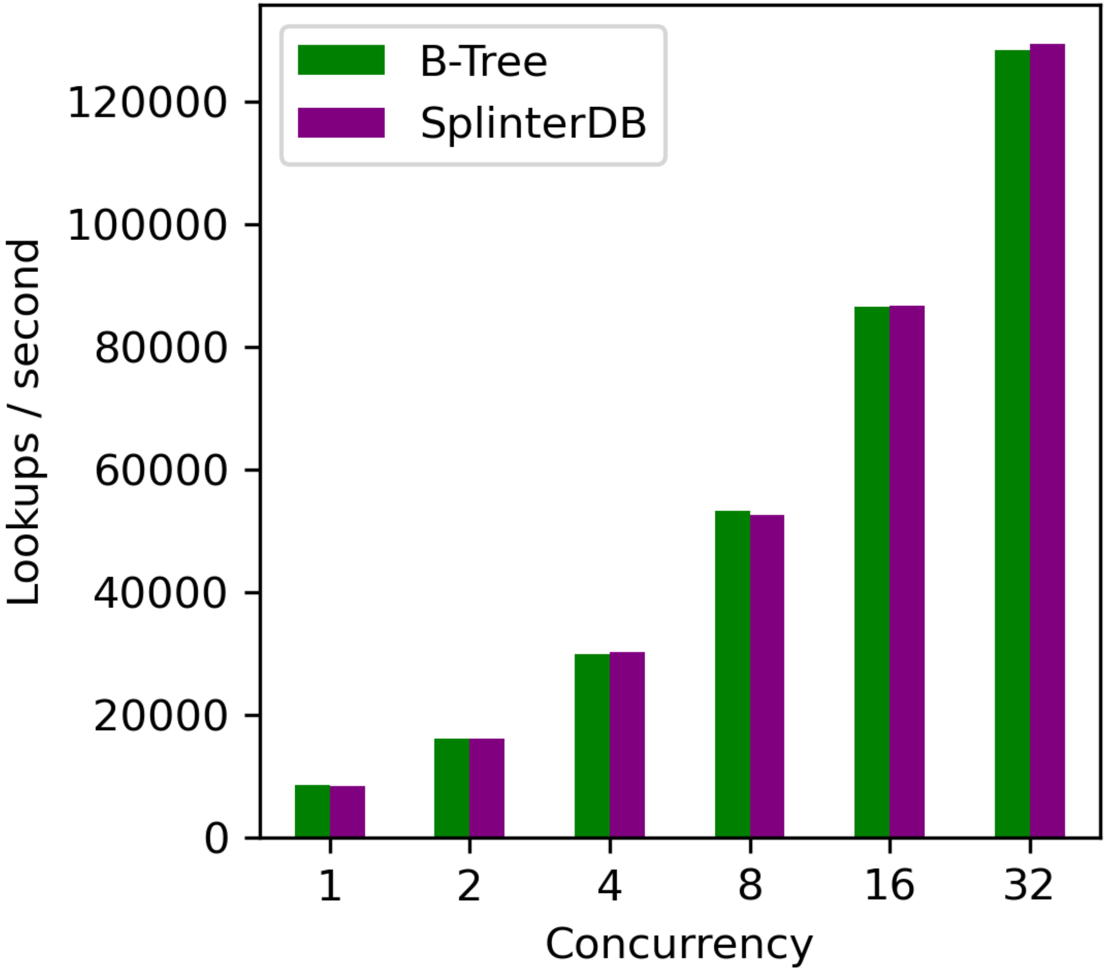
SplinterDB is **18x** faster than B-Tree

Random Point Queries

AWS i4i.16xlarge:
64 CPUs, fast local storage

1 Billion row table

with unique index.



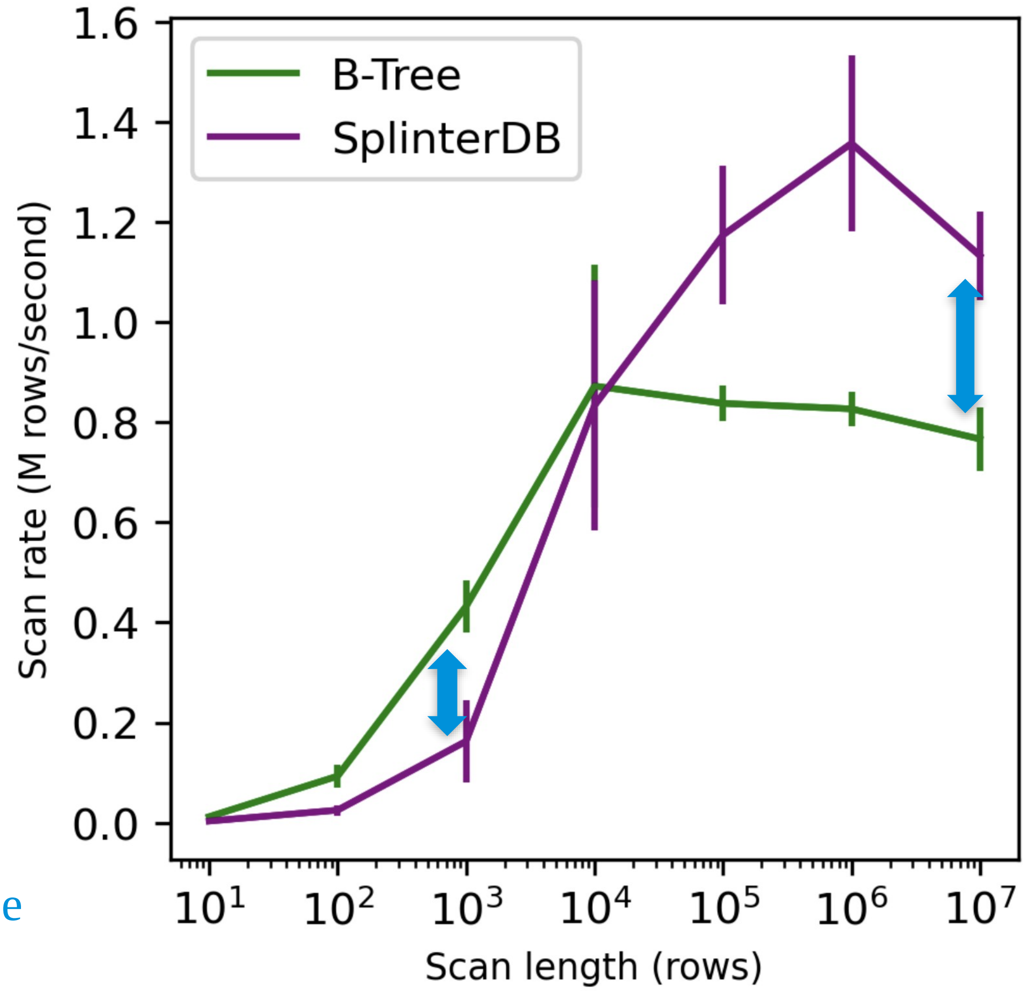
Scans

AWS i4i.16xlarge:
64 CPUs, fast local storage

Scans on a

1 billion row table,

from a cold cache



Scanning <= 1000 rows,

Splinter adds 1-5ms extra

latency compared to B-Tree

Scanning >= 100k

rows, Splinter is ~50%

faster than B-Tree.

Open Questions

Can we get insertion costs below log N / B log log N while keeping sublinear successor queries?

- Maplets are not the bottleneck

Can we improve successor query costs?

- Range maplets?
- Fractional cascading?

Successor lower bounds?