Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

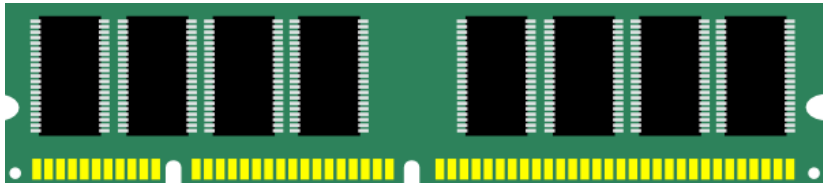
Deukyeon Hwang
UNIST

Wook-Hee Kim
UNIST

Youjip Won
Hanyang Univ.

Beomseok Nam
UNIST/SKKU

Background – Persistent Memory



Fast but Asymmetric Access Latency

Non-Volatility

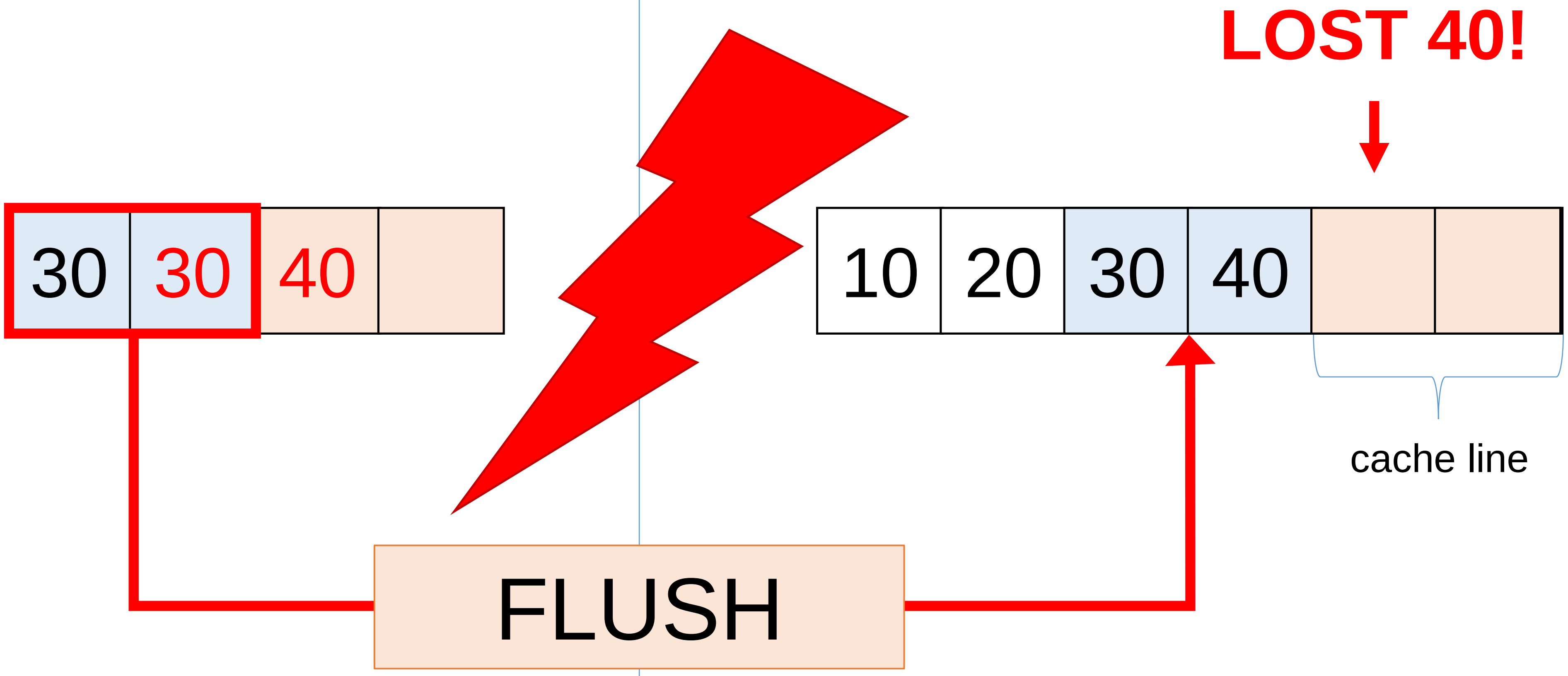
Byte-Addressability

Large Capacity

Background – B+-Tree for Persistent Memory

CPU Caches
(Volatile)

Persistent Memory (Non-Volatile)



LOST 40!

30 30 40

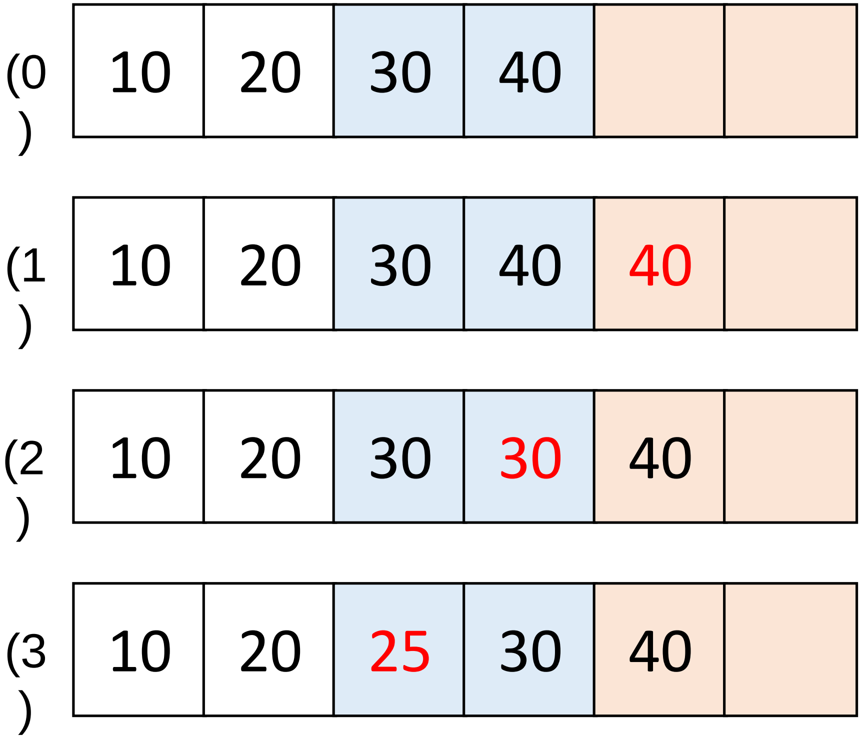
10 20 30 40

cache line

FLUSH

Background – B+-Tree for Persistent Memory

Inserting 25 into a node



(0)	10	20	30	40		
(1)	10	20	30	40	40	
(2)	10	20	30	30	40	
(3)	10	20	25	30	40	

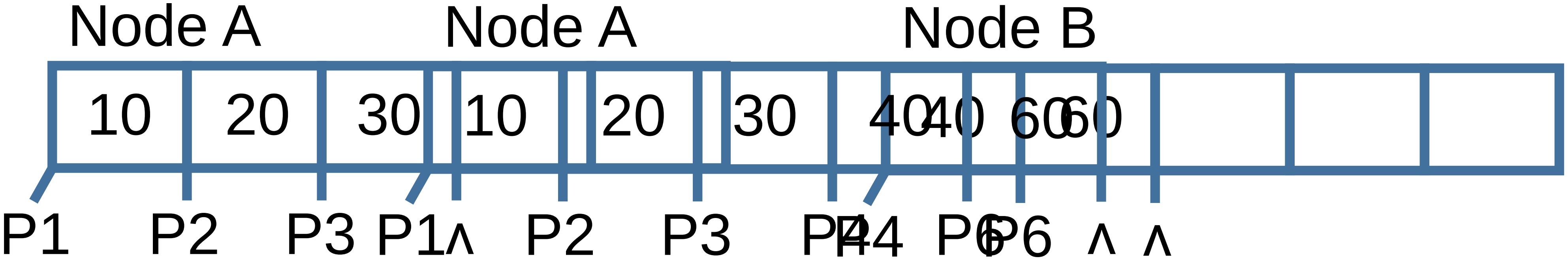
Partially updated tree node is inconsistent

↓

Append-Only Update

Background – B+-Tree for Persistent Memory

Node Split


Node A Node A Node B

10 20 30 10 20 30 40 40 60 60

P1 P2 P3 P1 ∧ P2 P3 P4 P4 P6 P6 ∧ ∧

Logging → Selective Persistence (Internal node in DRAM)

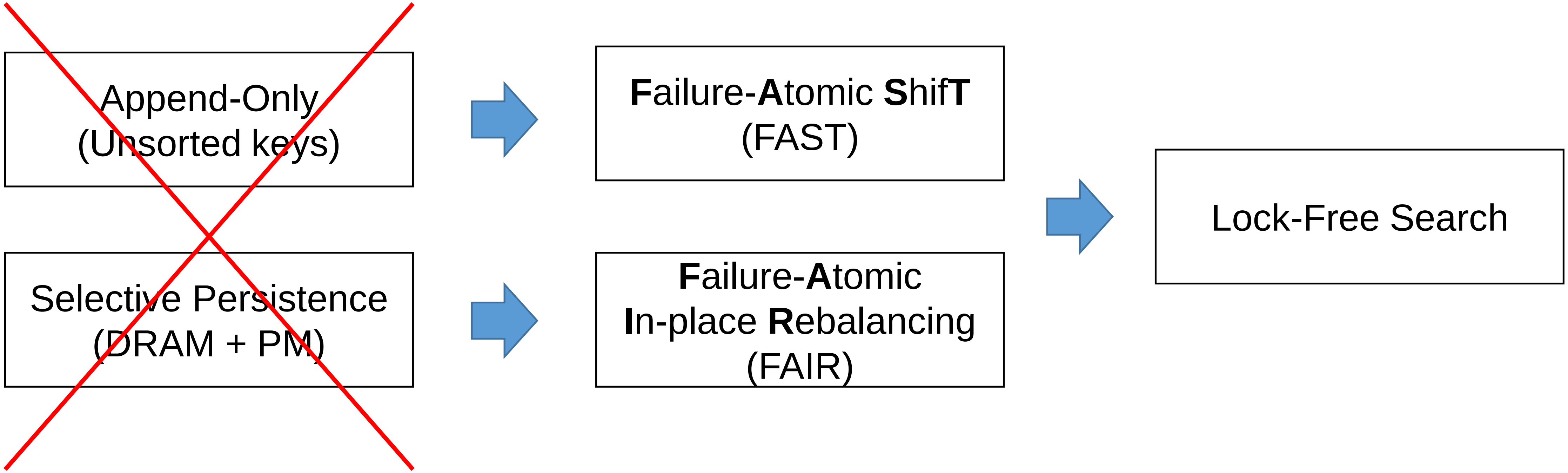
Background – B+-Tree for Persistent Memory

- Append-Only
  - Unsorted keys
- Selective Persistence
  - Internal node → DRAM
  - Internal nodes have to be reconstructed from leaf nodes after failures
  - Logging for leaf nodes

- Previous solutions

NV-Tree [FAST'15]	Append-Only leaf update + Selective Persistence
wB+-Tree [VLDB'15]	Append-Only node update + bitmap/slot array metadata
FP-Tree [SIGMOD'16]	Append-Only leaf update + fingerprints + Selective Persistence

Contributions



~~Append-Only~~
~~(Unsorted keys)~~

~~Selective Persistence~~
~~(DRAM + PM)~~

Failure-**A**tomic **S**hif**T**
(FAST)

Failure-**A**tomic
In-place **R**ebalancing
(FAIR)

Lock-Free Search

Background – Reordering Memory Access

- Modern processors reorder instructions to utilize the memory bandwidth
- Memory ordering in x86 and ARM

	x86	**ARM**
stores-after-stores	Y	N
stores-after-loads	N	N
loads-after-stores	N	N
loads-after-loads	N	N
Inst. w/ dependency	Y	Y

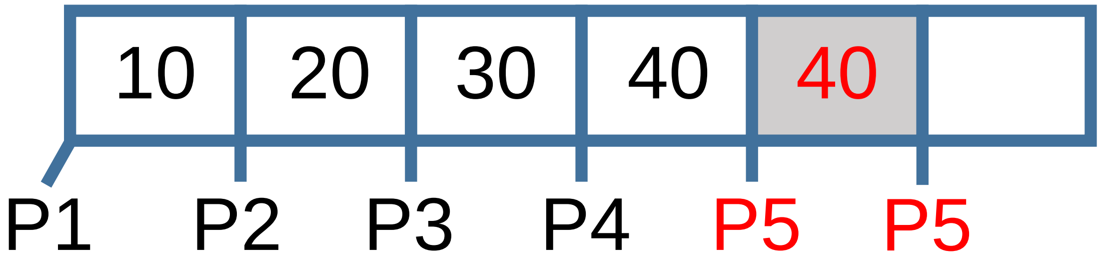
- x86 guarantees **Total Store Ordering (TSO)**
- Dependent instructions are not reordered

Failure-Atomic ShifT (FAST)

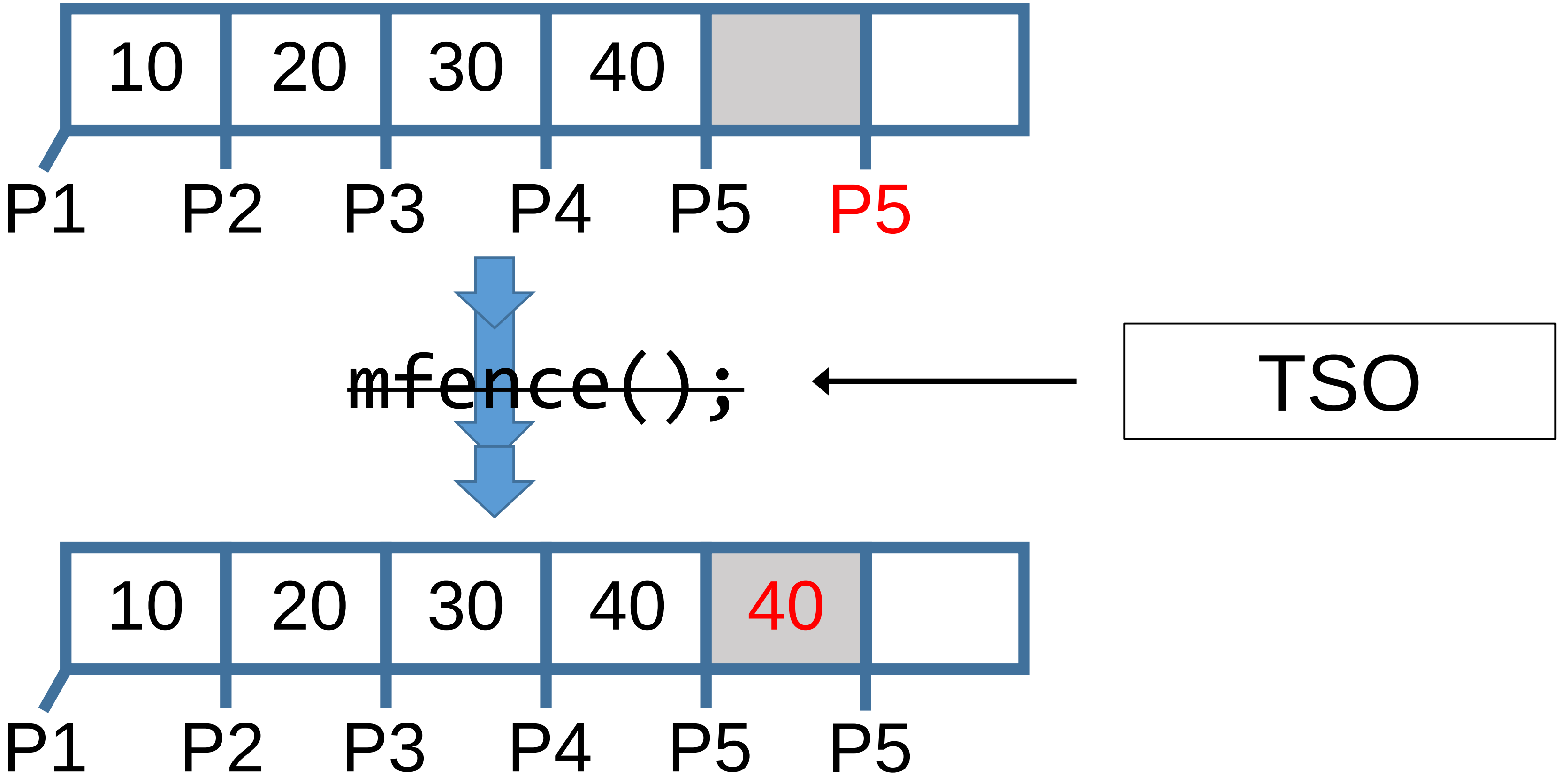
- Pointers in B+-Tree store unique memory addresses
- 8-byte pointer can be atomically updated

Read transactions detect *transient inconsistency* between duplicate pointers

- *transient inconsistency*
    - In-flight state partially updated by a write transaction

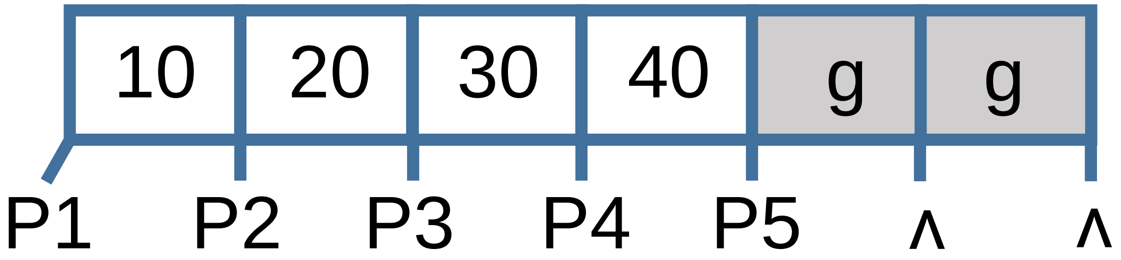


Failure-Atomic ShifT (FAST)



Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST

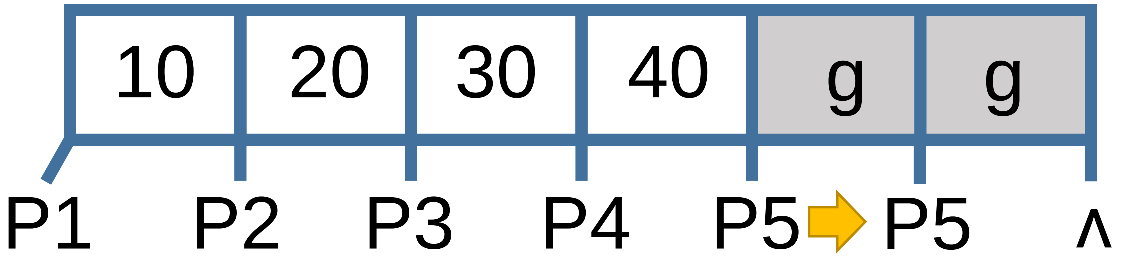


g: Garbage
ʌ: Null

Read transactions can succeed in finding a key even if a system crashes in any step

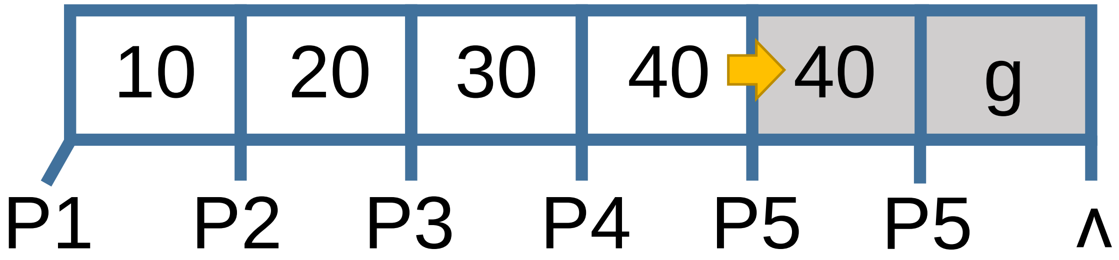
Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST



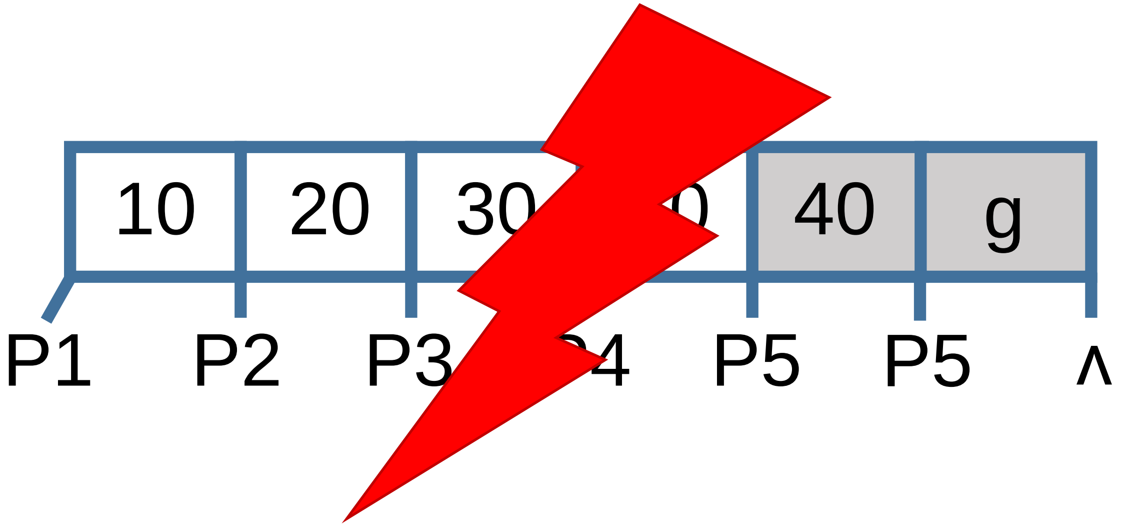
Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST



Failure-Atomic ShifT (FAST)

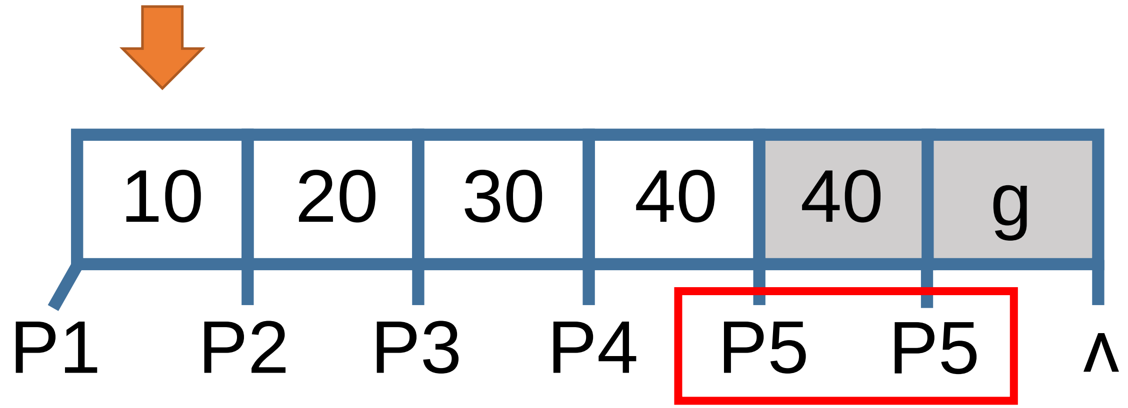
Insert (25, P6) into a node using FAST



Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST

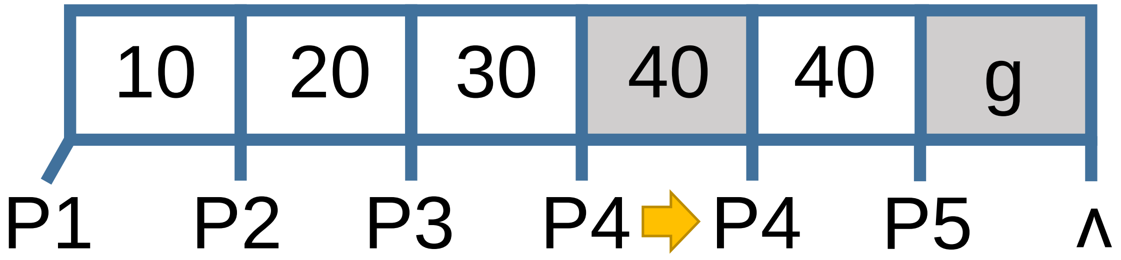
read transaction



Key 40 between duplicate pointers is ignored!

Failure-Atomic ShifT (FAST)

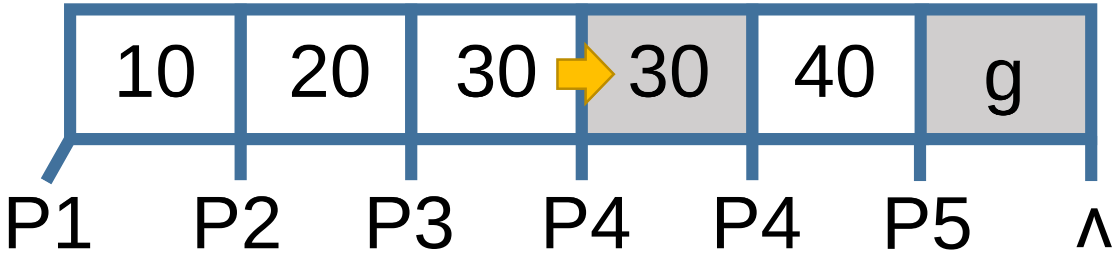
Insert (25, P6) into a node using FAST



Shifting P4 invalidates the left 40

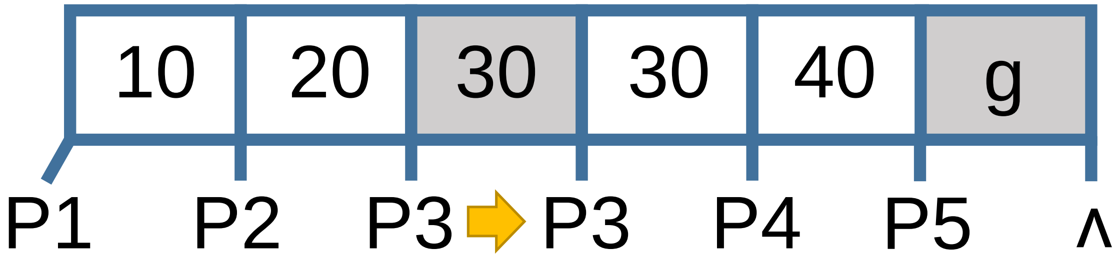
Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST



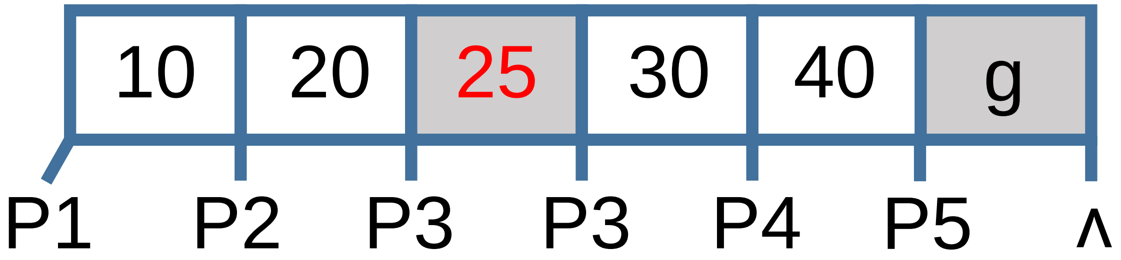
Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST



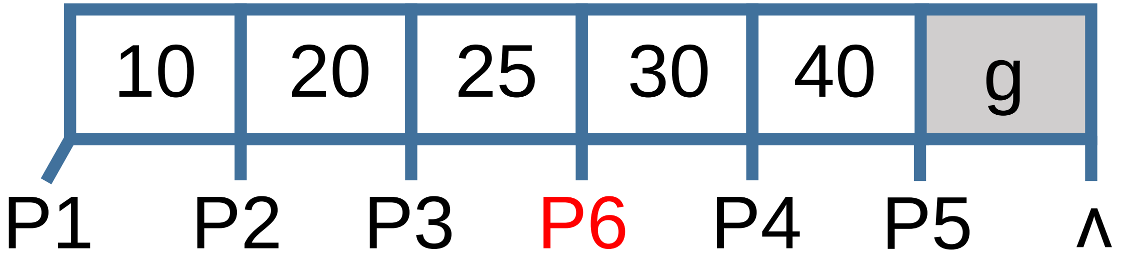
Failure-Atomic ShifT (FAST)

Insert (25, P6) into a node using FAST



Failure-Atomic ShifT (FAST)

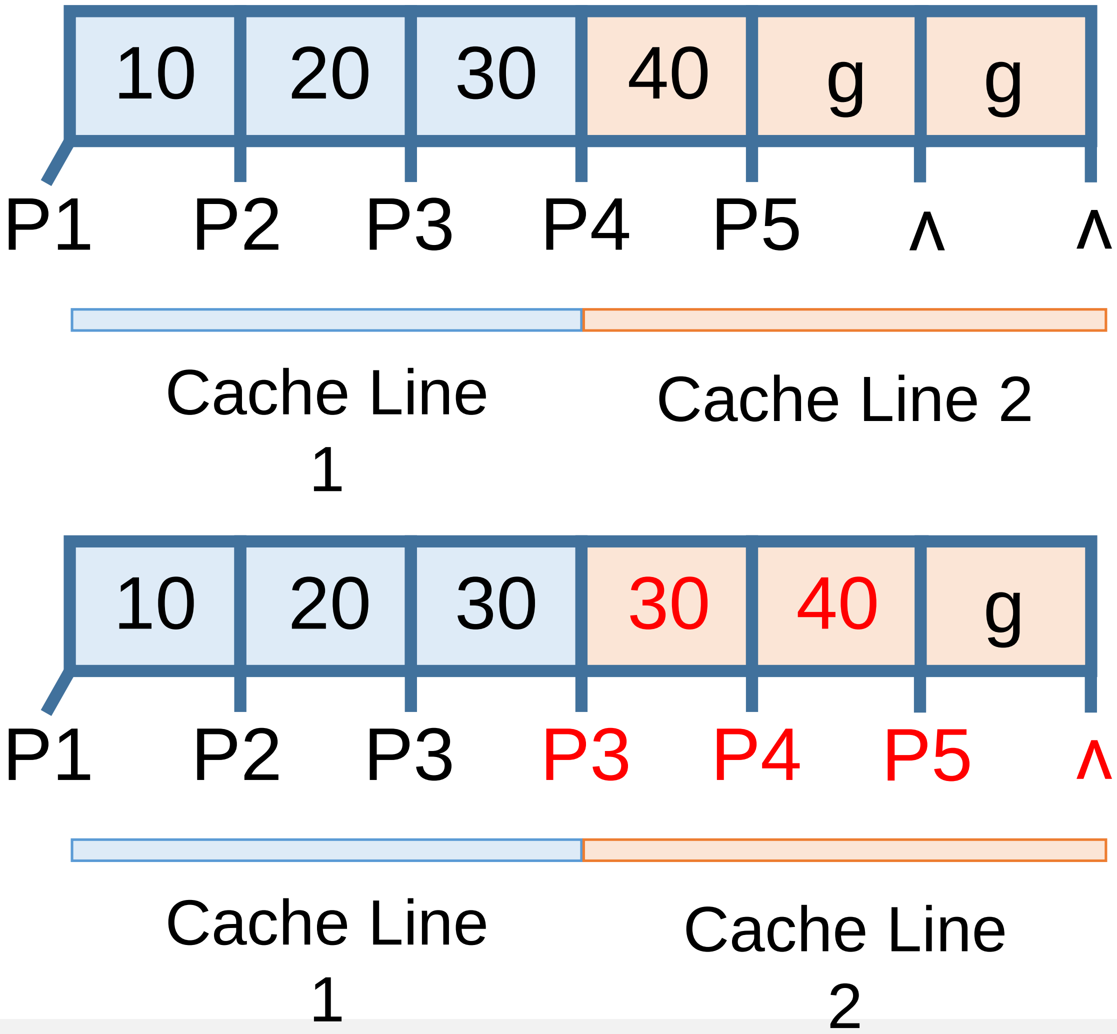
Insert (25, P6) into a node using FAST



Storing P6 validates 25

Failure-Atomic ShifT (FAST)

- It is necessary to call `clflush` at the boundary of cache line



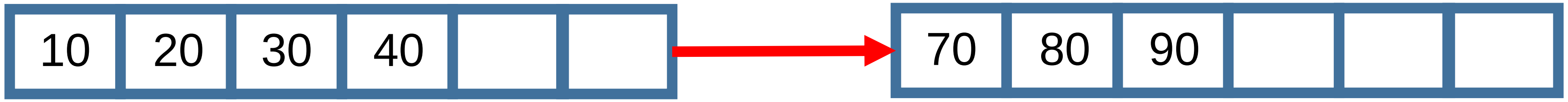
```
mfence()
clflush( Cache Line 2 )
mfence()
```

Failure-Atomic In-place Rebalancing (FAIR)

- Let's avoid expensive logging

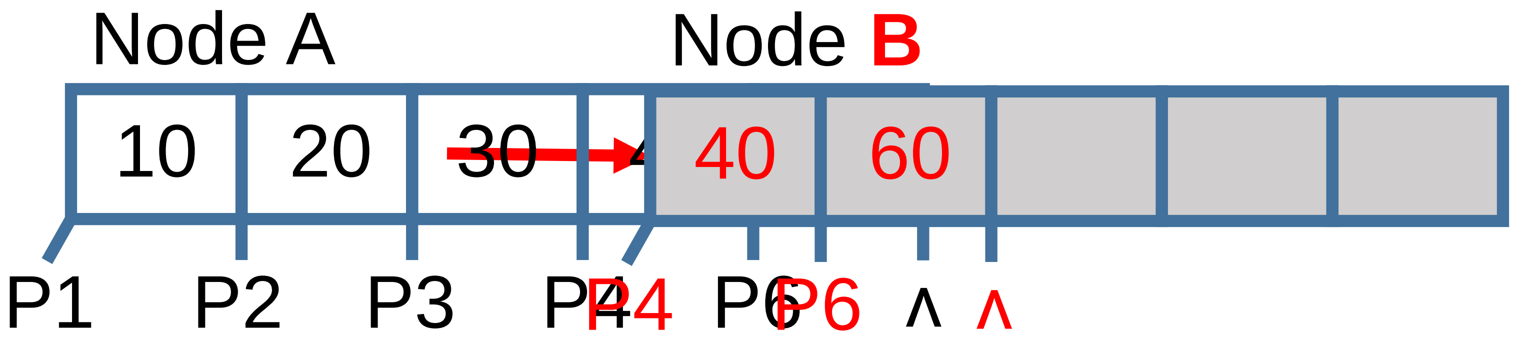
by making read transactions
be aware of rebalancing operations

- B^{link}-Tree



Failure-Atomic In-place Rebalancing (FAIR)

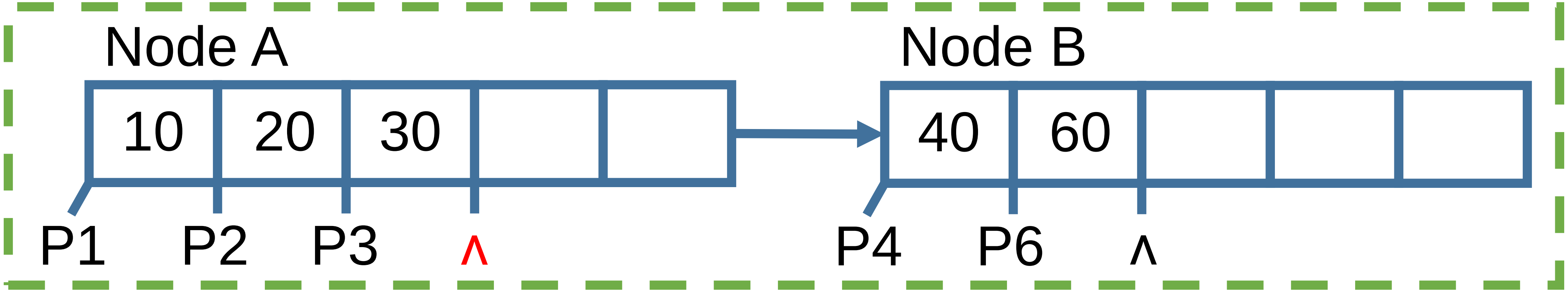
FAIR split a node



A read transaction can detect transient inconsistency if keys are out of order

Failure-Atomic In-place Rebalancing (FAIR)

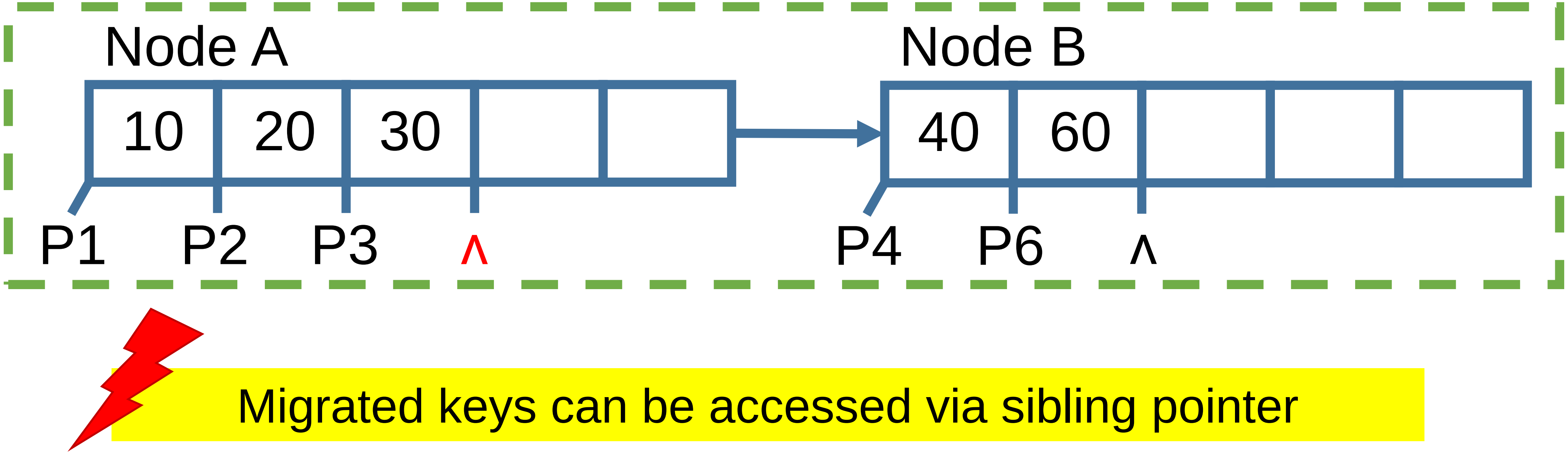
FAIR split a node



Setting NULL pointer validates Node B.
Node A and Node B are virtually a single node

Failure-Atomic In-place Rebalancing (FAIR)

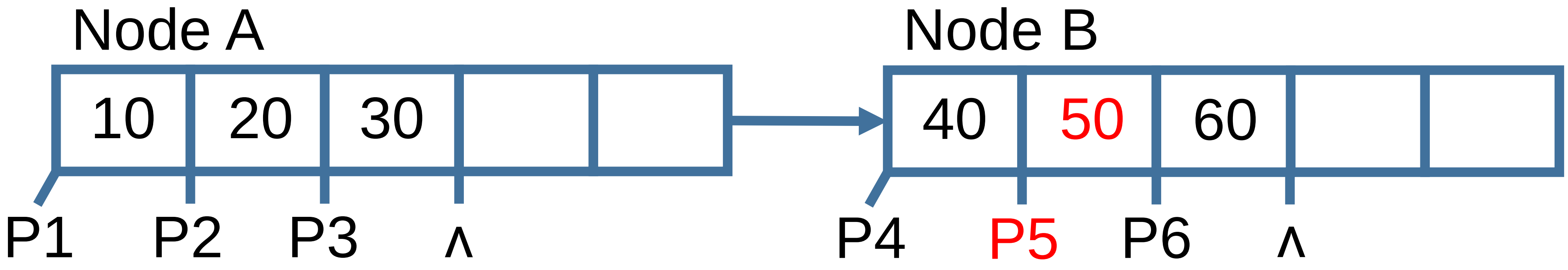
FAIR split a node



Migrated keys can be accessed via sibling pointer

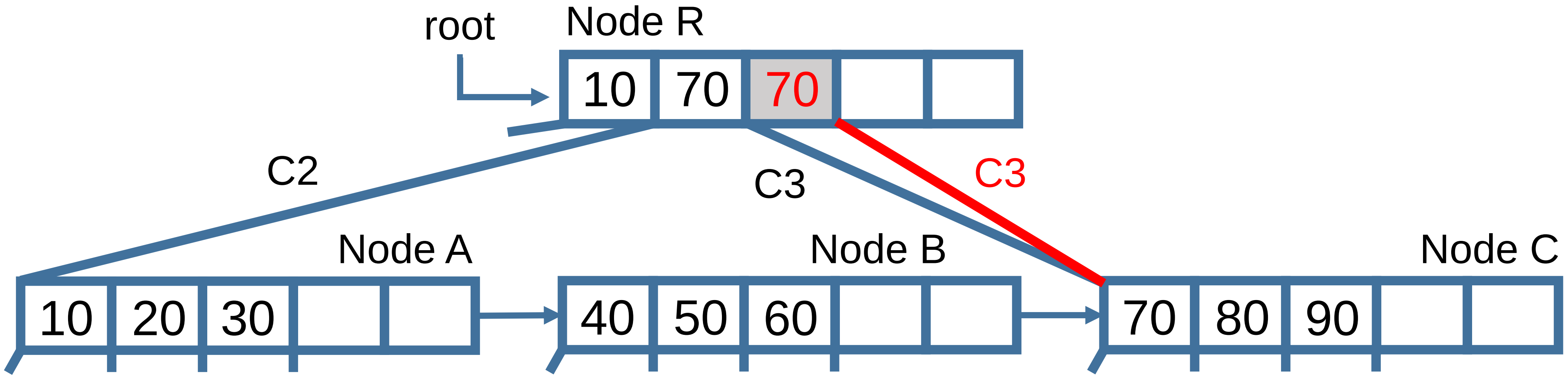
Failure-Atomic In-place Rebalancing (FAIR)

FAIR split a node



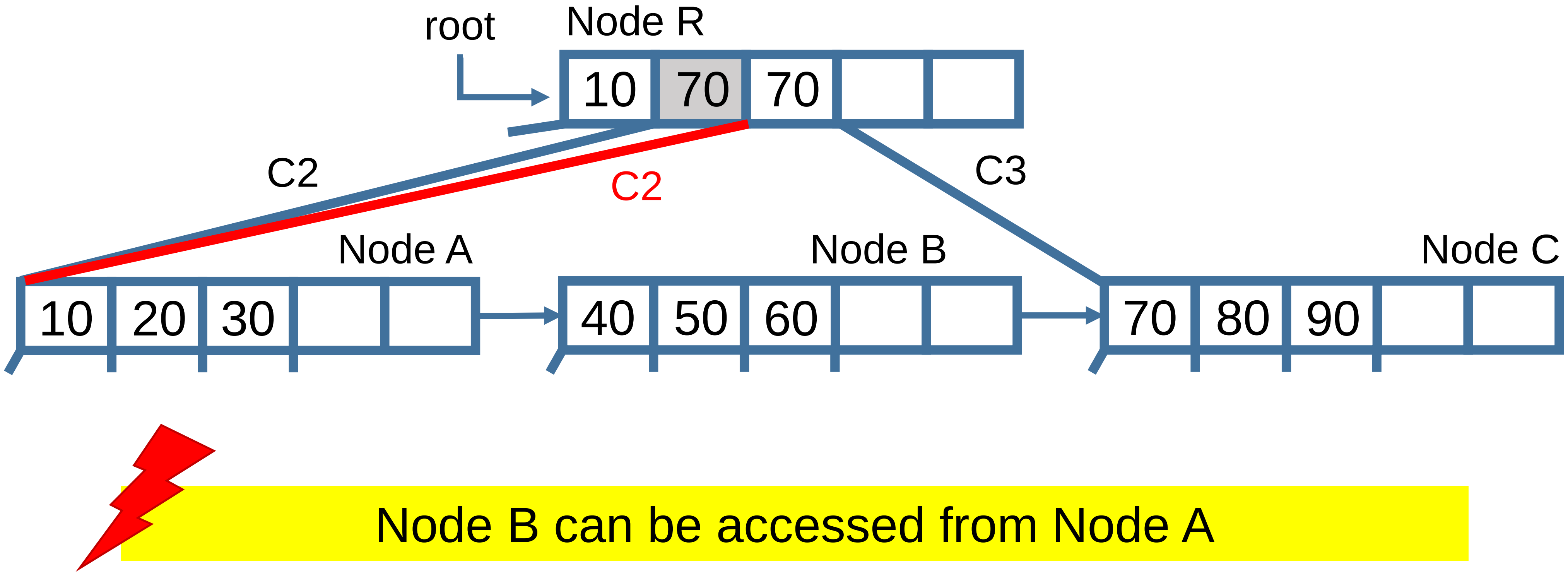
Failure-Atomic In-place Rebalancing (FAIR)

Insert a key into the parent node using FAST after FAIR split



Failure-Atomic In-place Rebalancing (FAIR)

Insert a key into the parent node using FAST after FAIR split

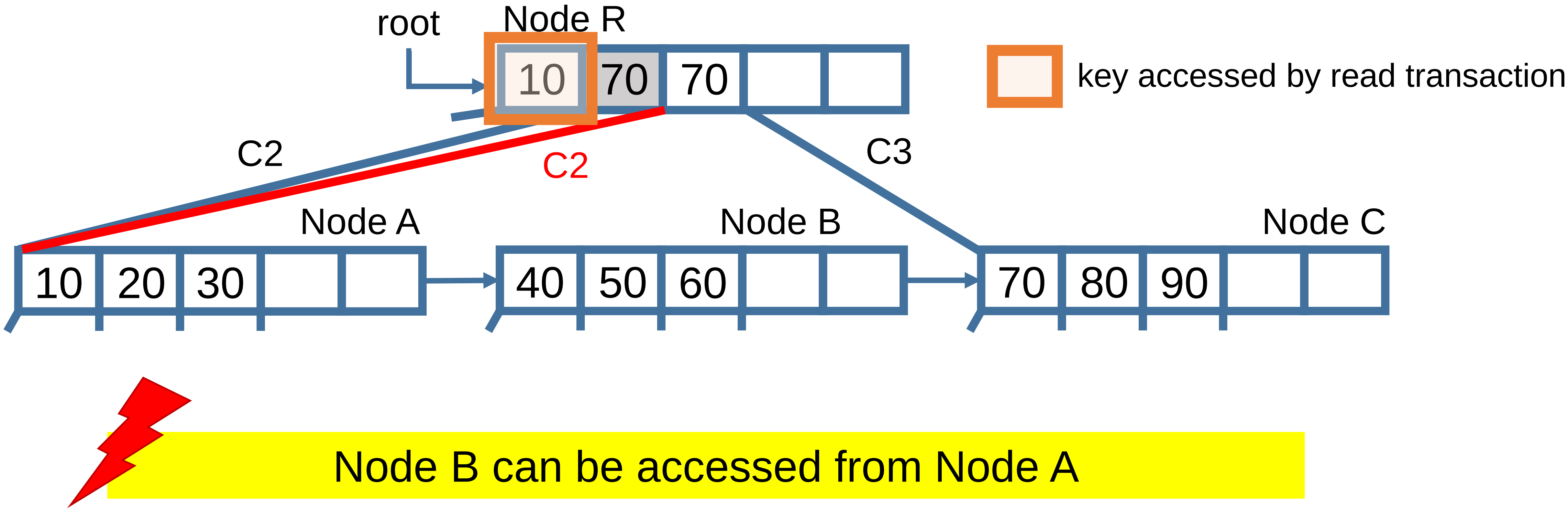


Node B can be accessed from Node A

Failure-Atomic In-place Rebalancing (FAIR)

Insert a key into the parent node using FAST after FAIR split

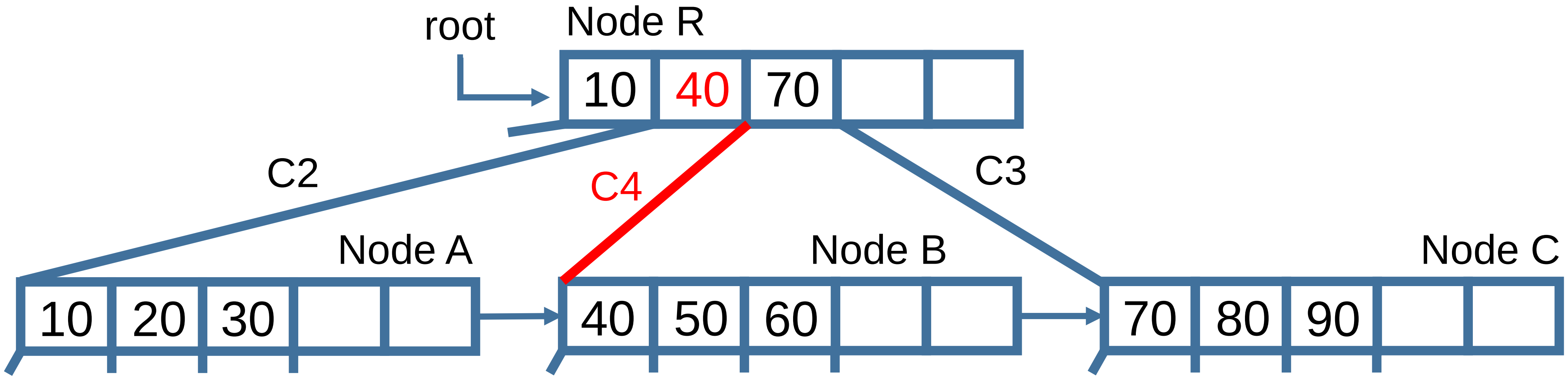
- Searching the key 50 from the root after a system crash



Node B can be accessed from Node A

Failure-Atomic In-place Rebalancing (FAIR)

Insert a key into the parent node using FAST after FAIR split



FAST inserting makes Node B visible atomically

Lock-Free Search

Read transactions can tolerate any inconsistency caused by write transactions

↓

Read transactions can access the transient inconsistent tree node being modified by a write transaction

↓

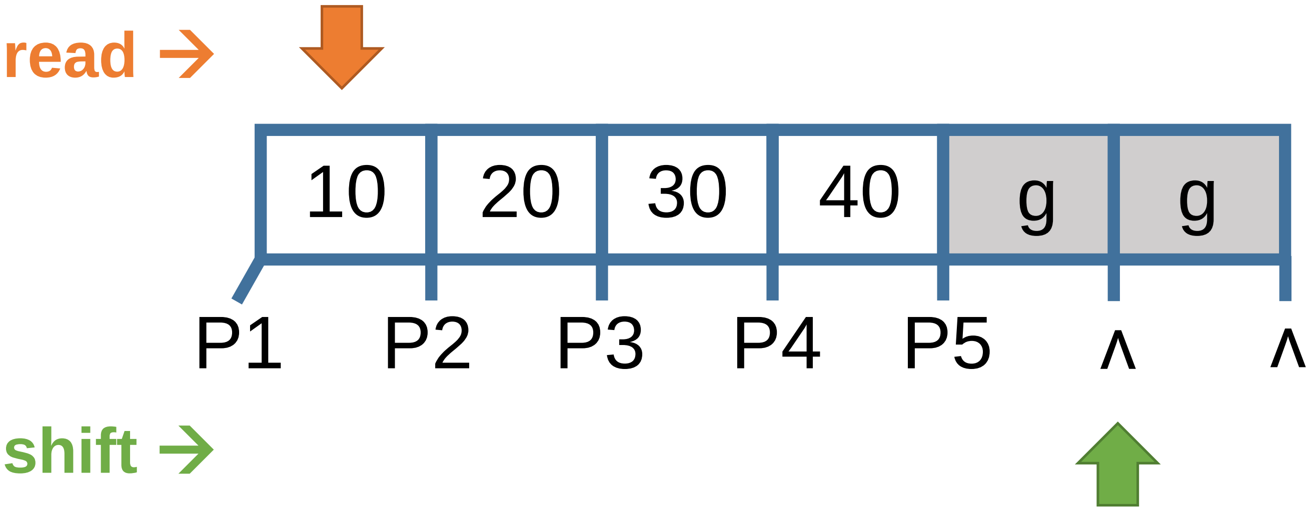
Lock-Free Search

Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)

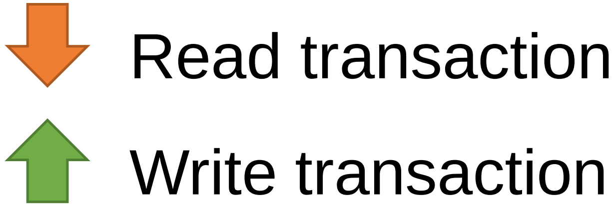
⬇ Read transaction

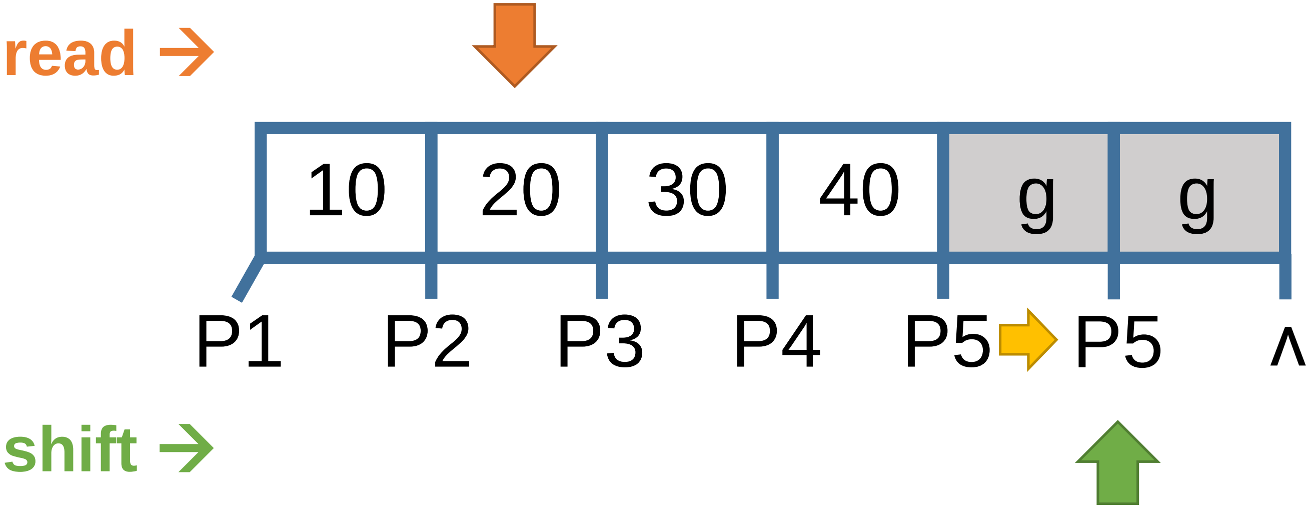
⬆ Write transaction



Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)



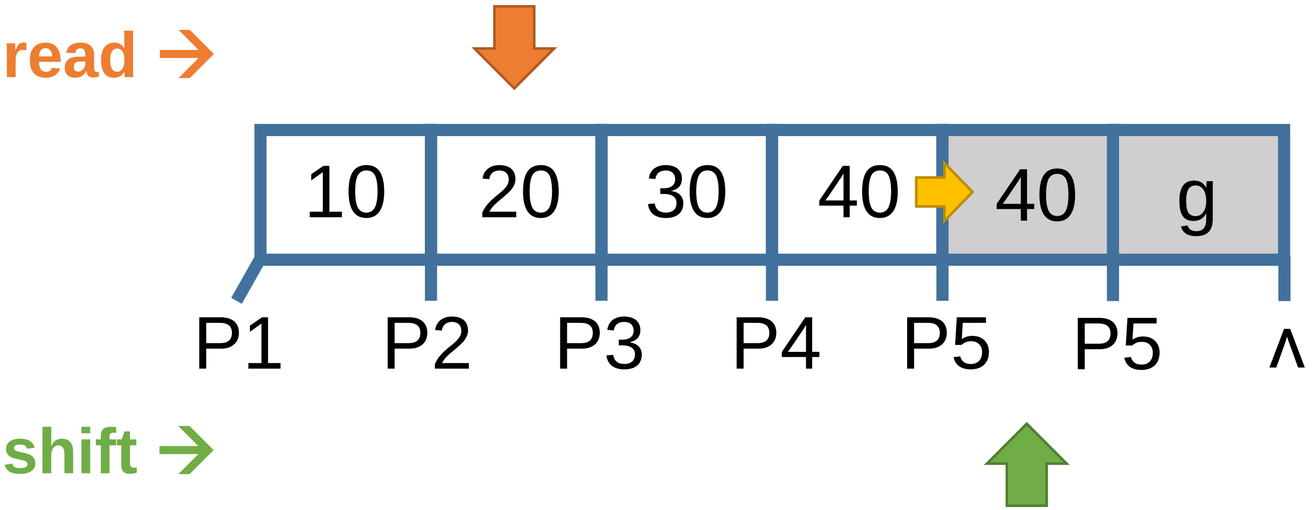


Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)

↓ Read transaction

↑ Write transaction

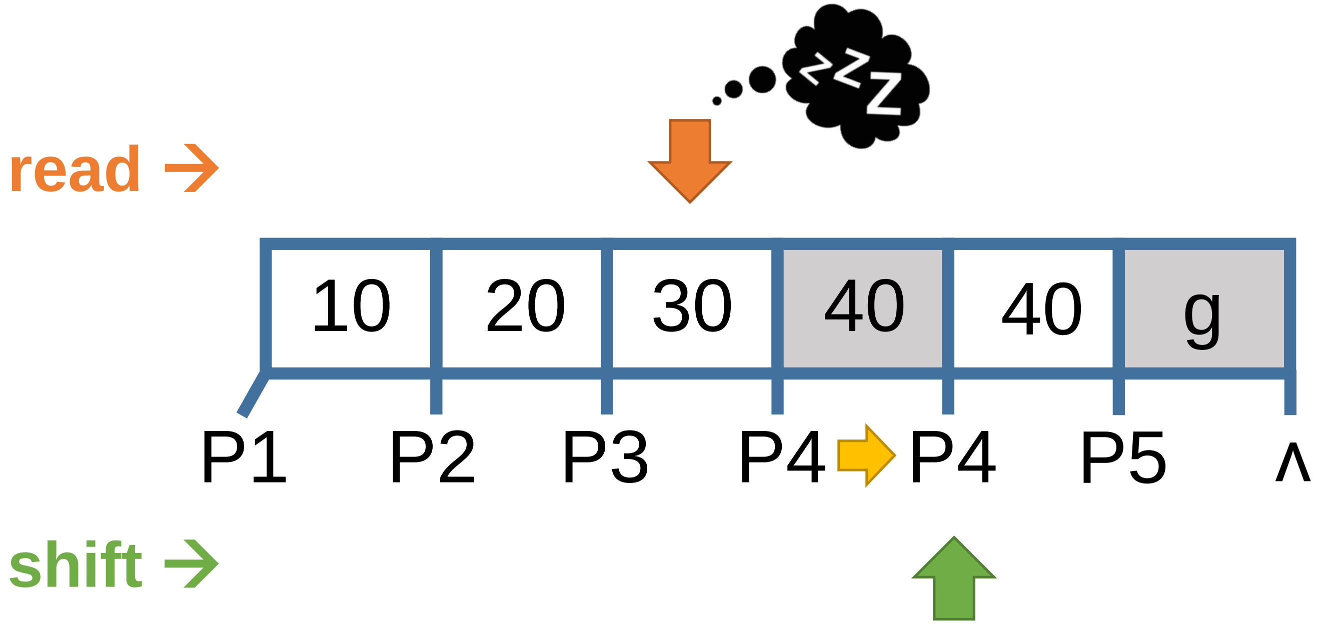


Lock-Free Search

⬇ Read transaction

⬆ Write transaction

[Example 1] Searching 30 while inserting (15, P6)



read →

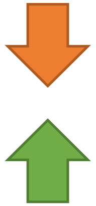
10	20	30	40	40	g

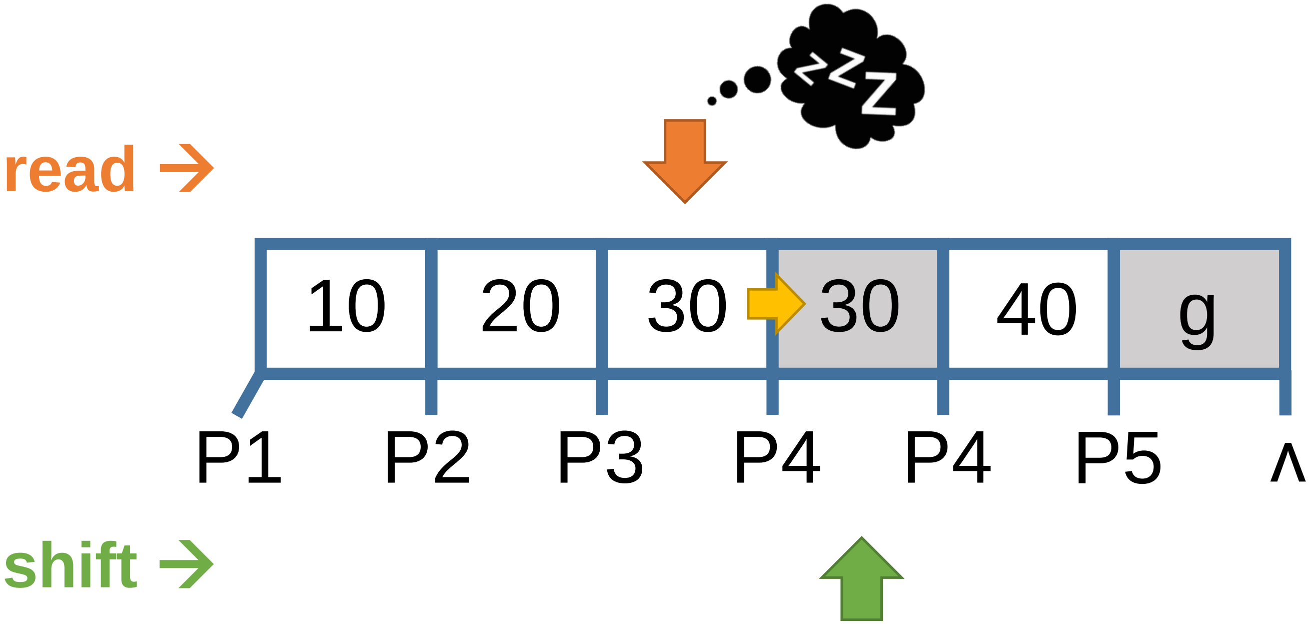
P1 P2 P3 P4 ⇨ P4 P5 ∧

shift →

Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)



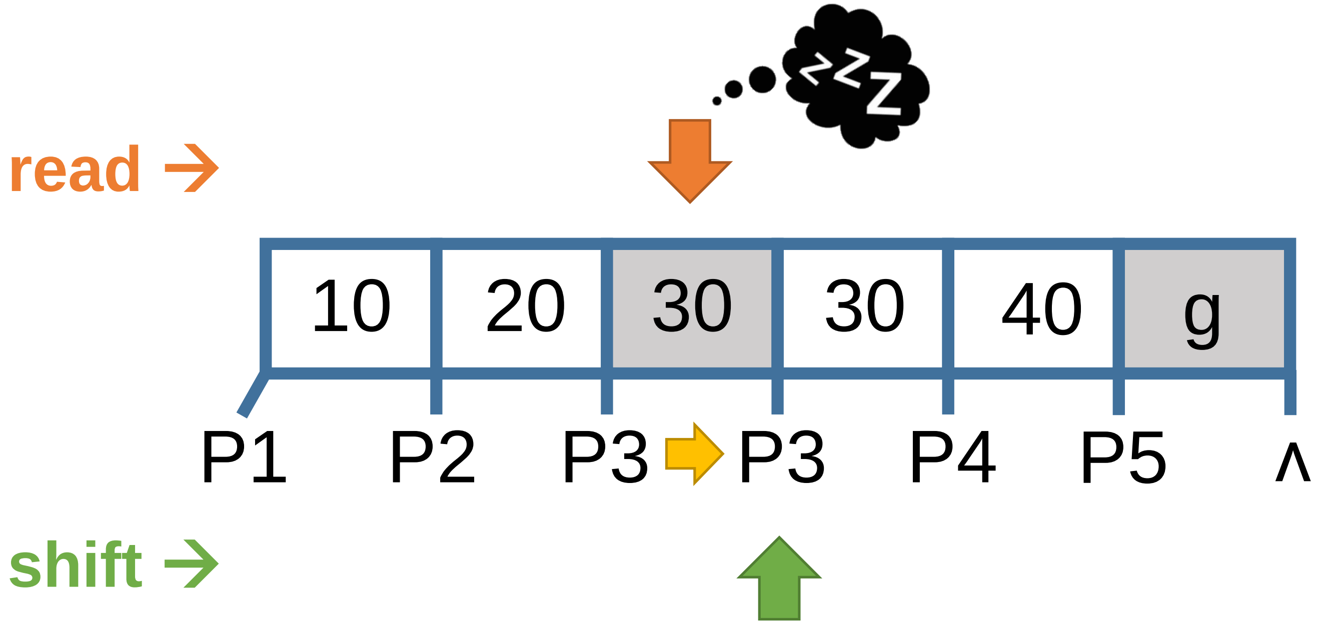


Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)

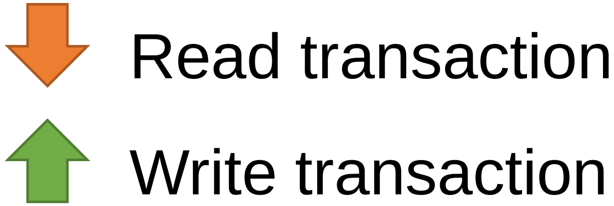
⬇ Read transaction

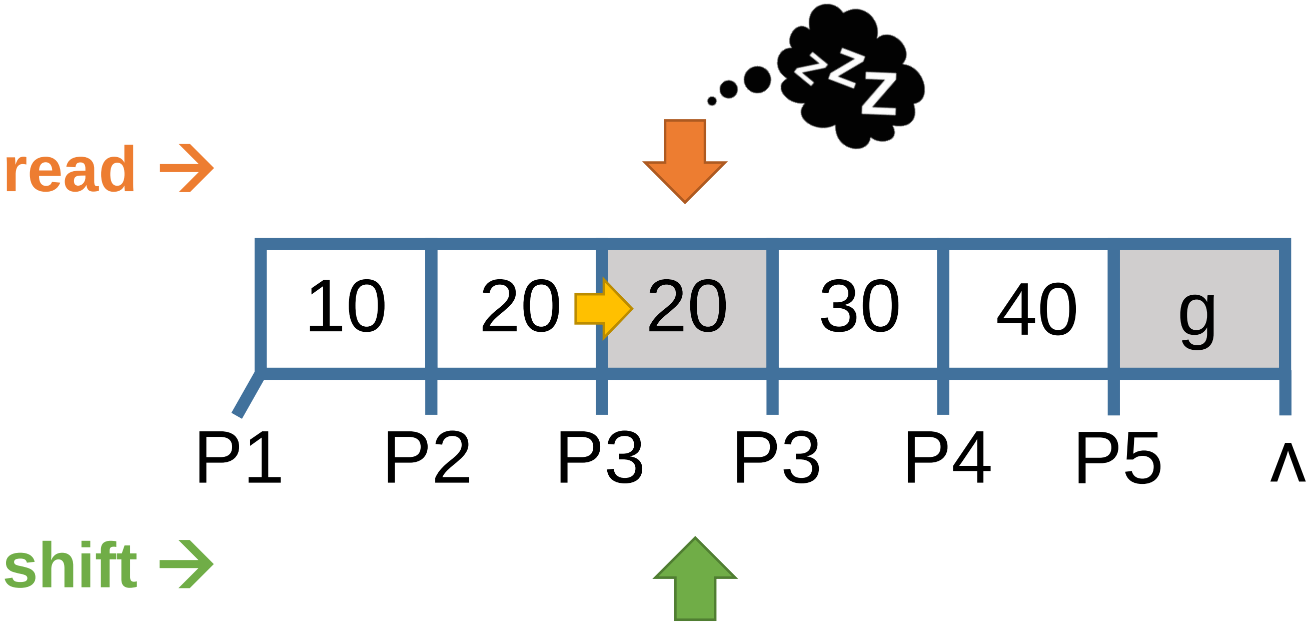
⬆ Write transaction



Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)



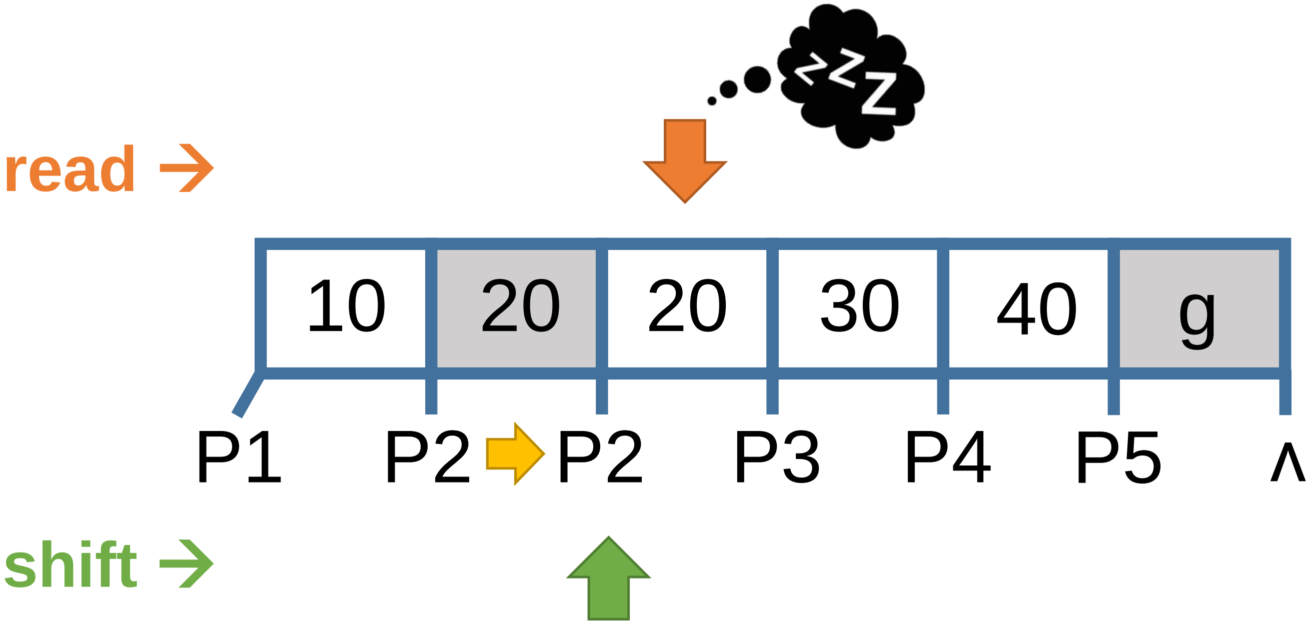


Lock-Free Search

↓ Read transaction

↑ Write transaction

[Example 1] Searching 30 while inserting (15, P6)

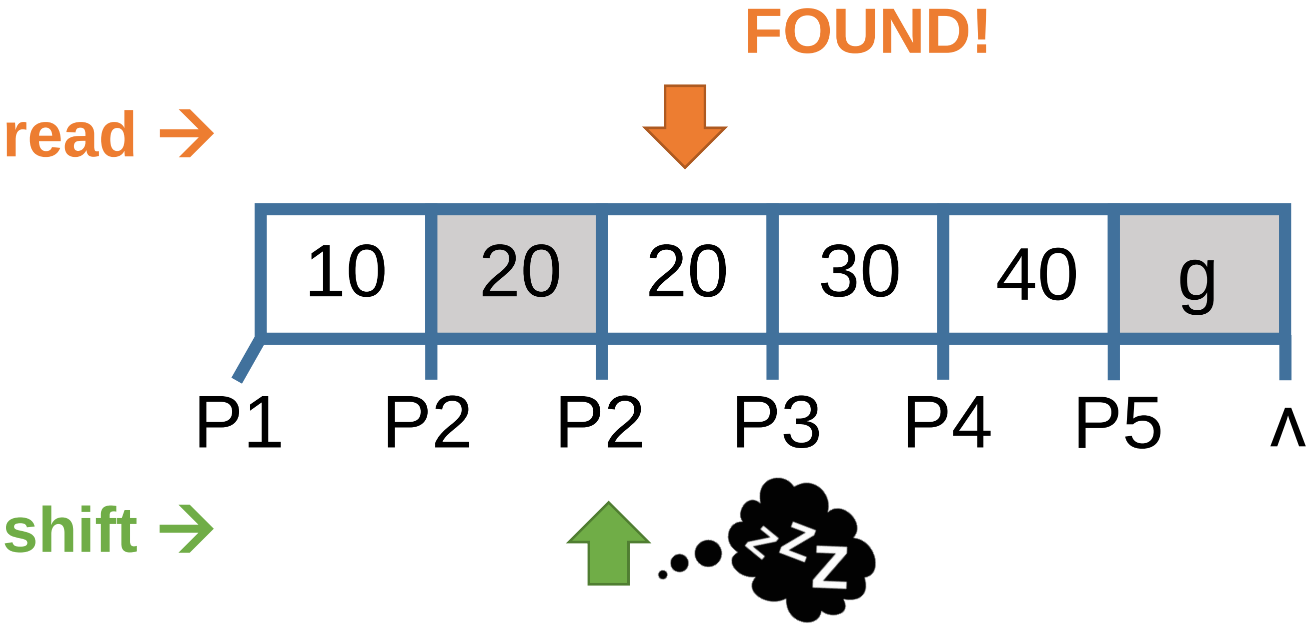


Lock-Free Search

[Example 1] Searching 30 while inserting (15, P6)

⬇ Read transaction

⬆ Write transaction



FOUND!

read →

10	20	20	30	40	g

P1 P2 P2 P3 P4 P5 ∧

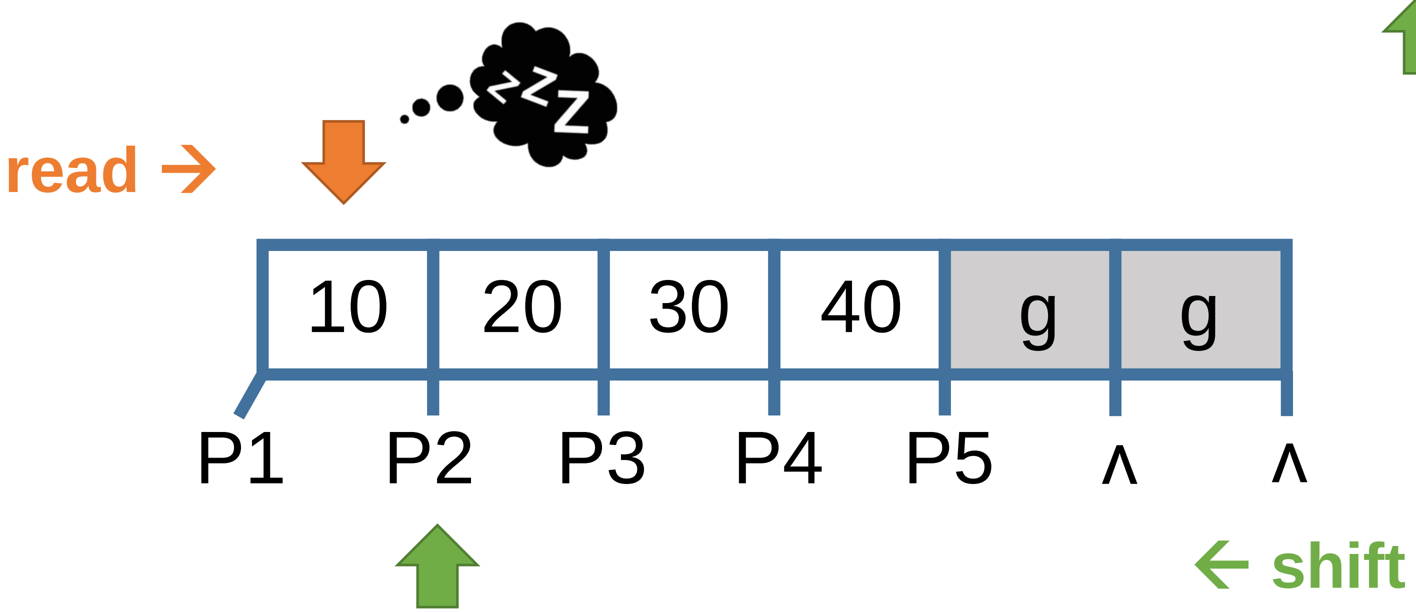
shift →

Lock-Free Search

[Example 2] Searching 30 while deleting (20, P2)

⬇ Read transaction

⬆ Write transaction

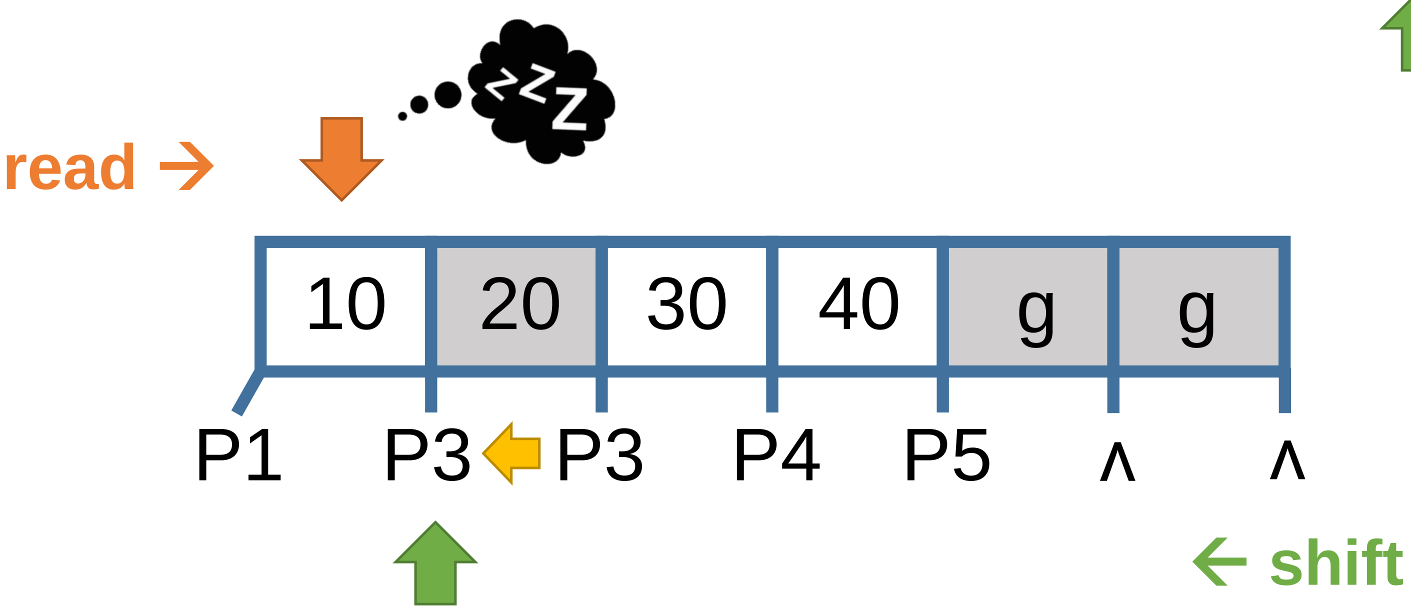


Lock-Free Search

[Example 2] Searching 30 while deleting (20, P2)

↓ Read transaction

↑ Write transaction

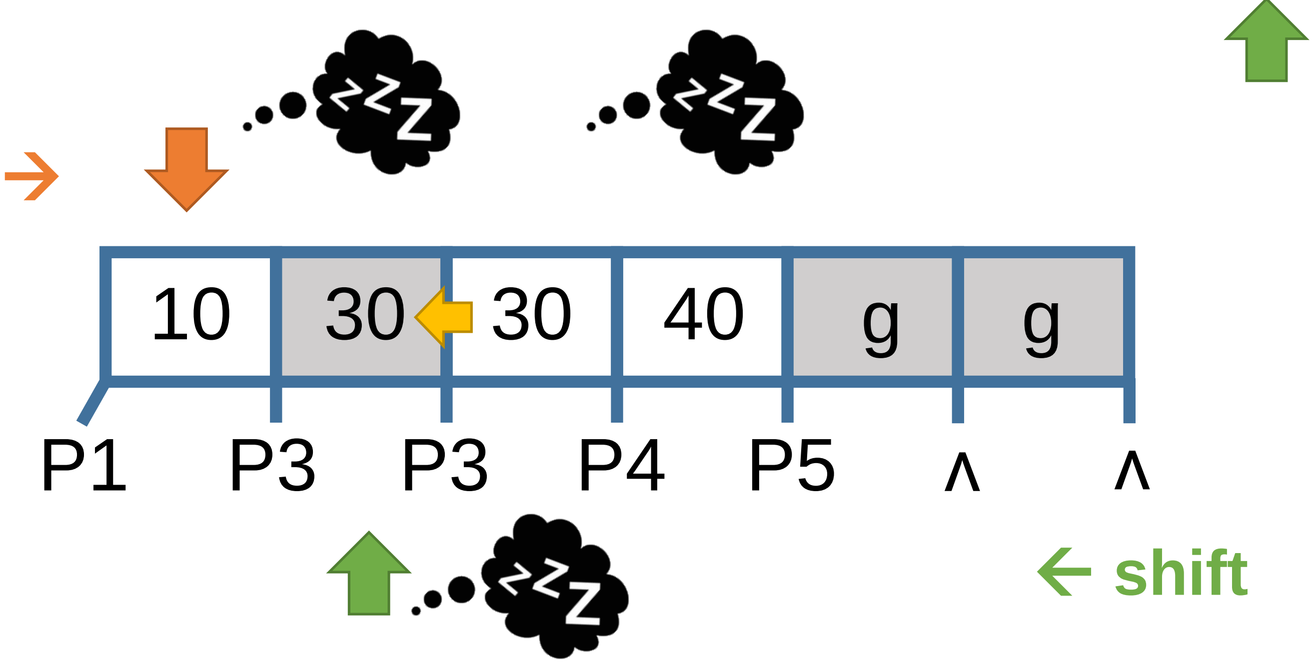


Lock-Free Search

[Example 2] Searching 30 while deleting (20, P2)

⬇ Read transaction

⬆ Write transaction

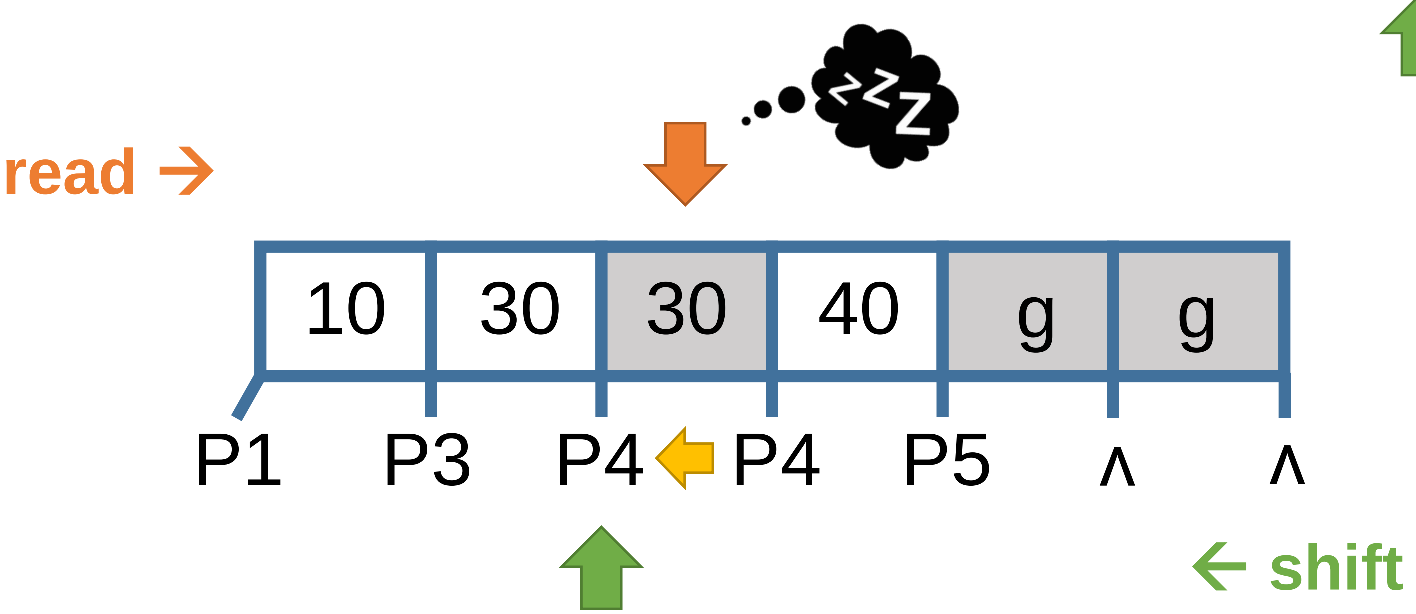


Lock-Free Search

[Example 2] Searching 30 while deleting (20, P2)

⬇ Read transaction

⬆ Write transaction

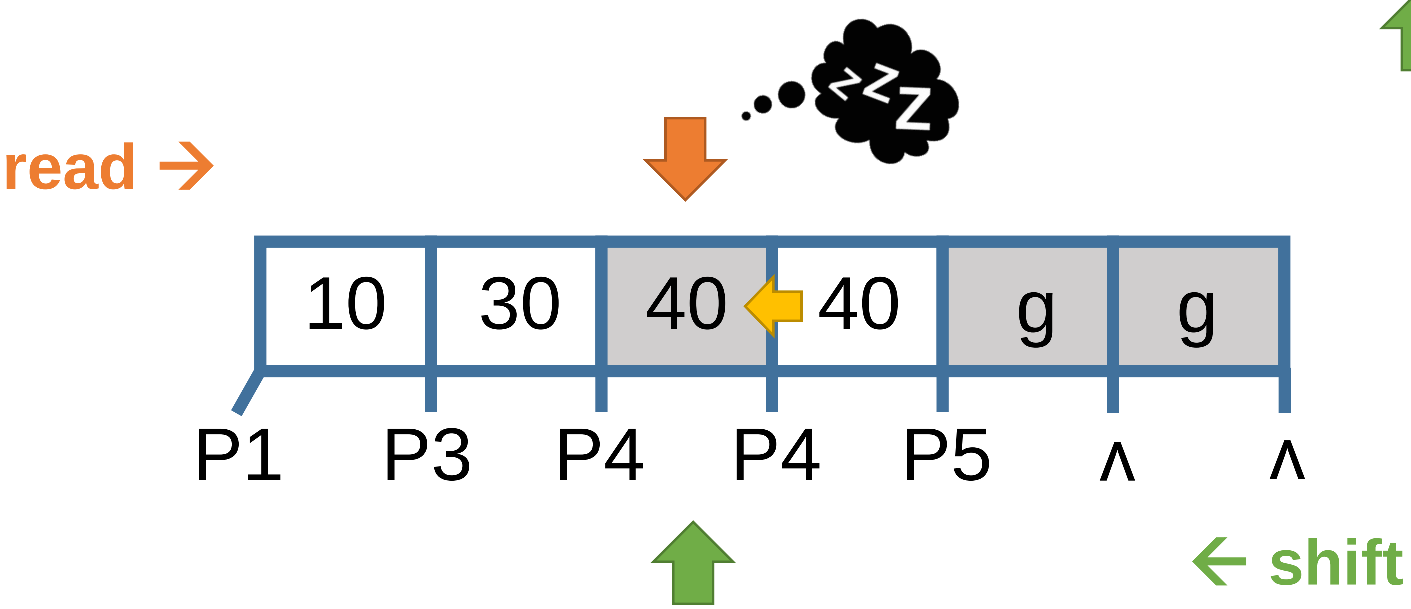


Lock-Free Search

[Example 2] Searching 30 while deleting (20, P2)

↓ Read transaction

↑ Write transaction

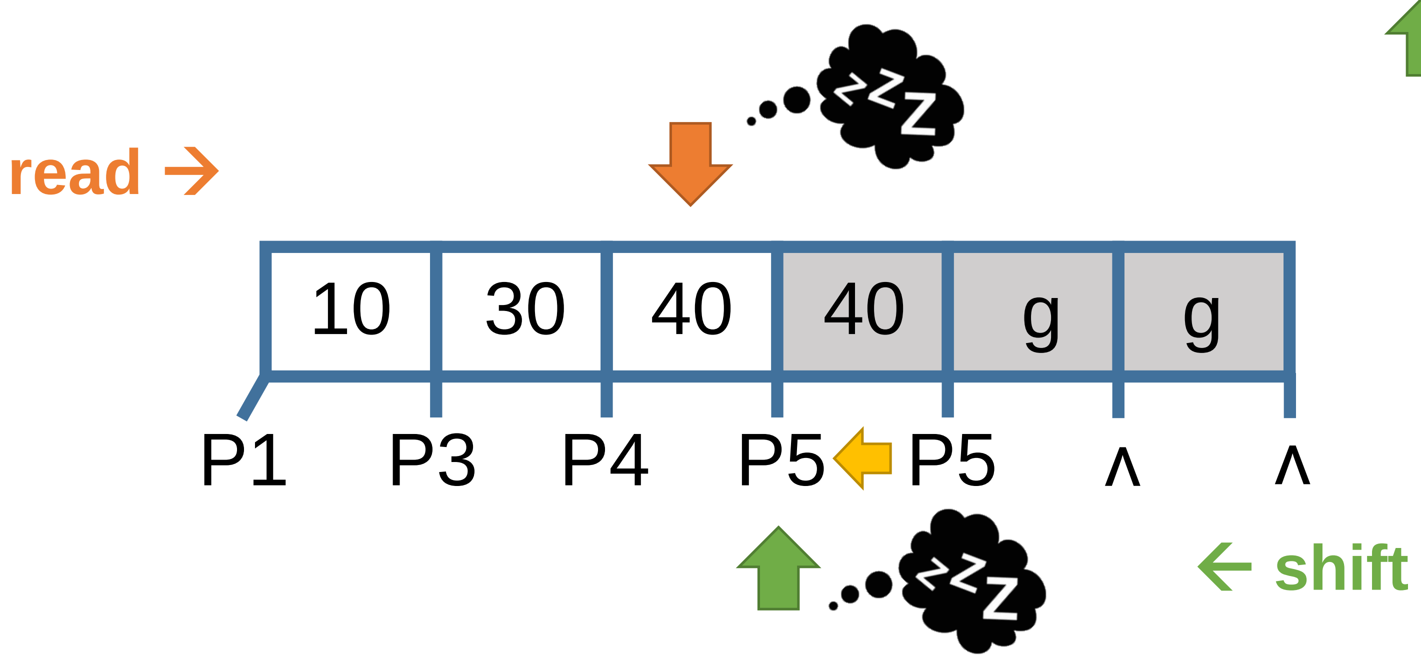


Lock-Free Search

[Example 2] Searching 30 while deleting (20, P2)

⬇ Read transaction

⬆ Write transaction



read →

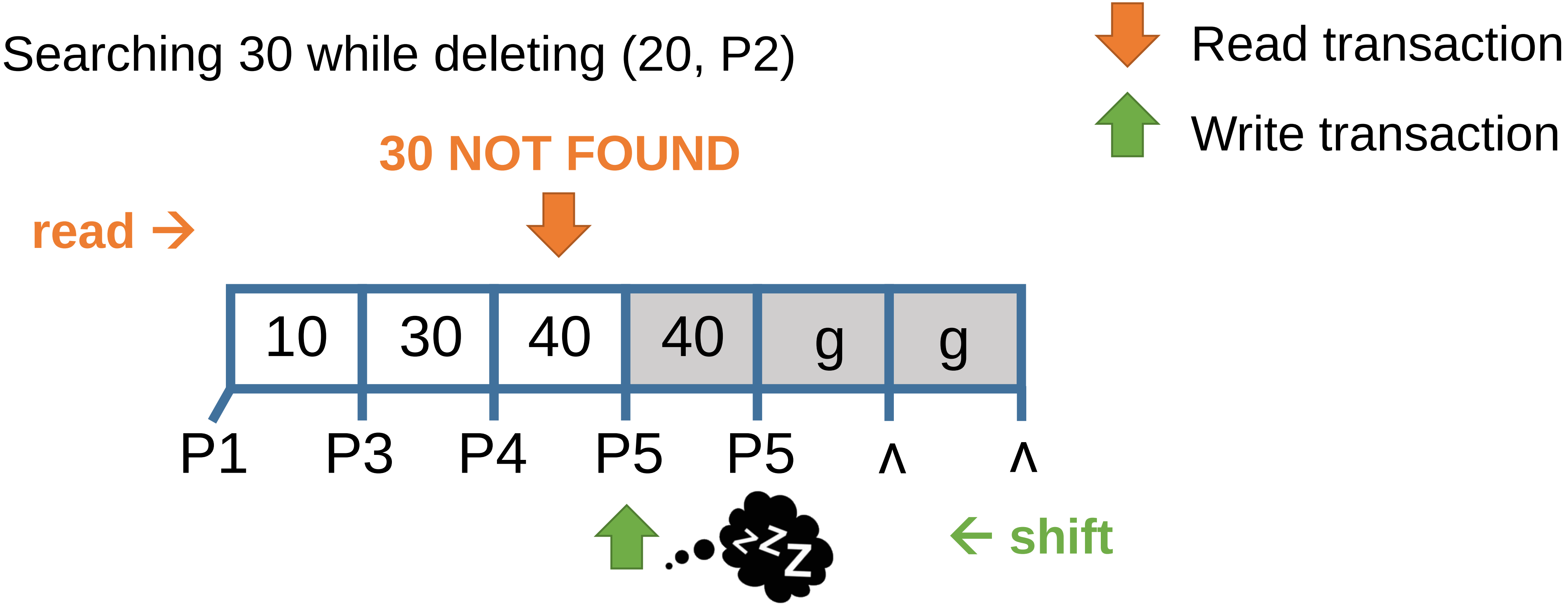
10	30	40	40	g	g

P1 P3 P4 P5 ⬅ P5 ∧ ∧

← shift

Lock-Free Search

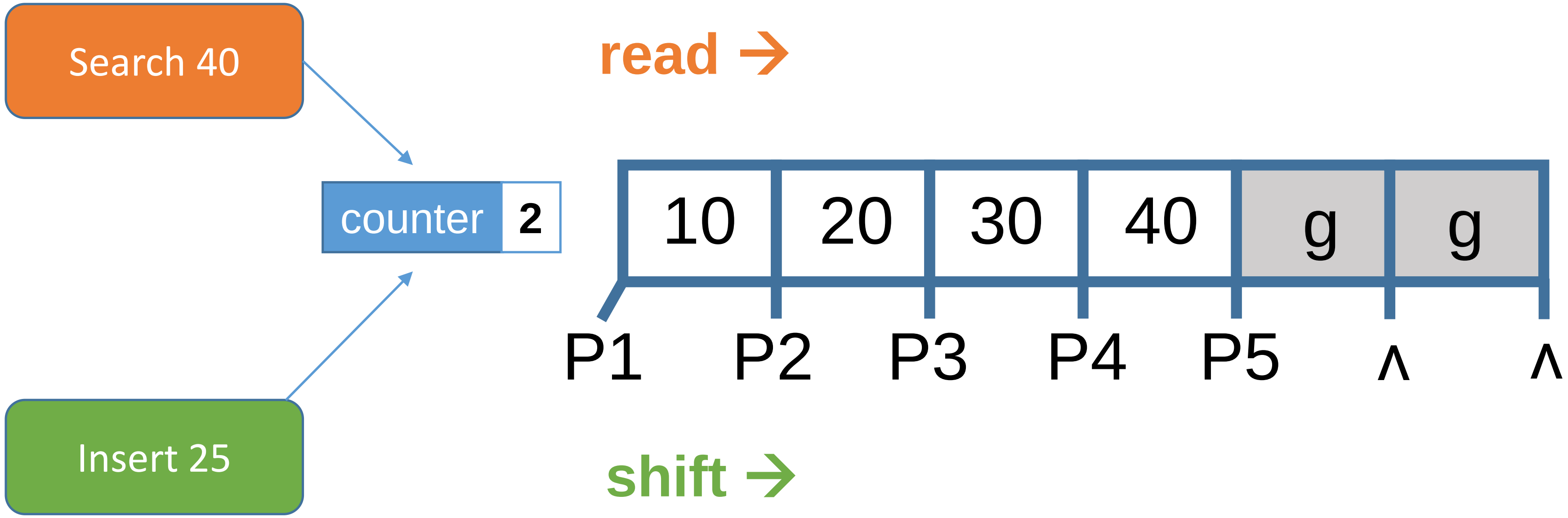
[Example 2] Searching 30 while deleting (20, P2)



The read transaction cannot find the key 30 due to shift operation

Lock-Free Search

- Direction flag:
  - Even Number
    - Insertion shifts to the right.
    - Search must scan from Left to Right
  - Odd Number
    - Deletion shifts to the left.
    - Search must scan from Right to Left



Search 40

read →

counter 2

10	20	30	40	g	g

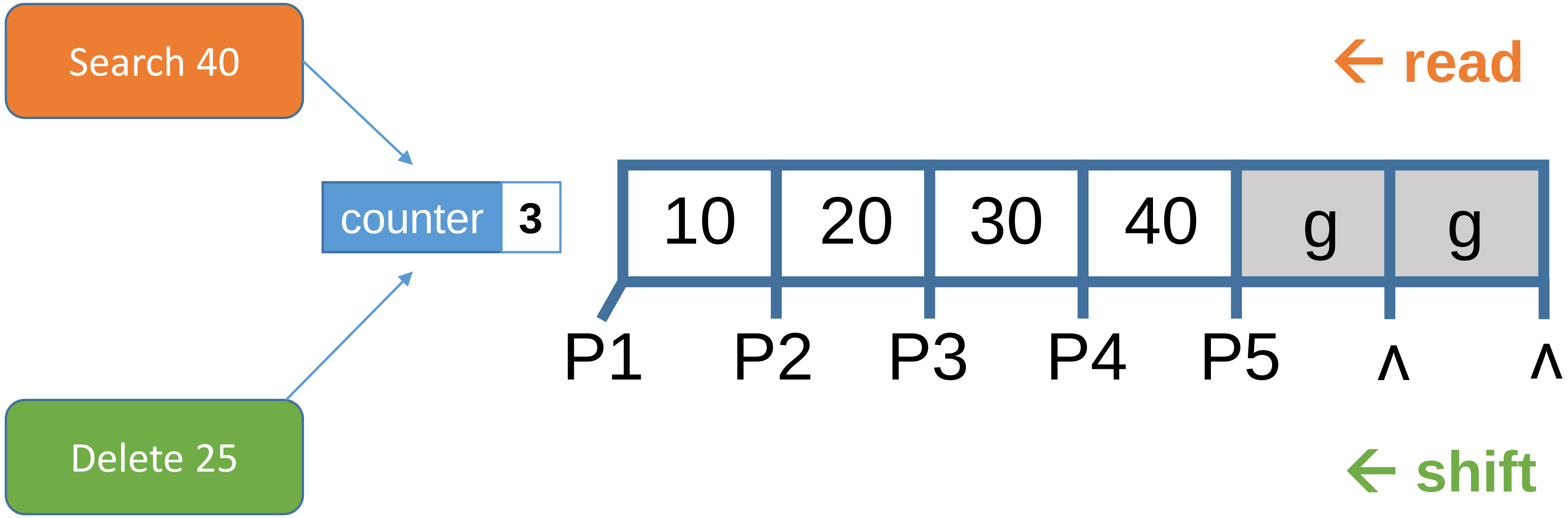
P1 P2 P3 P4 P5 ∧ ∧

shift →

Insert 25

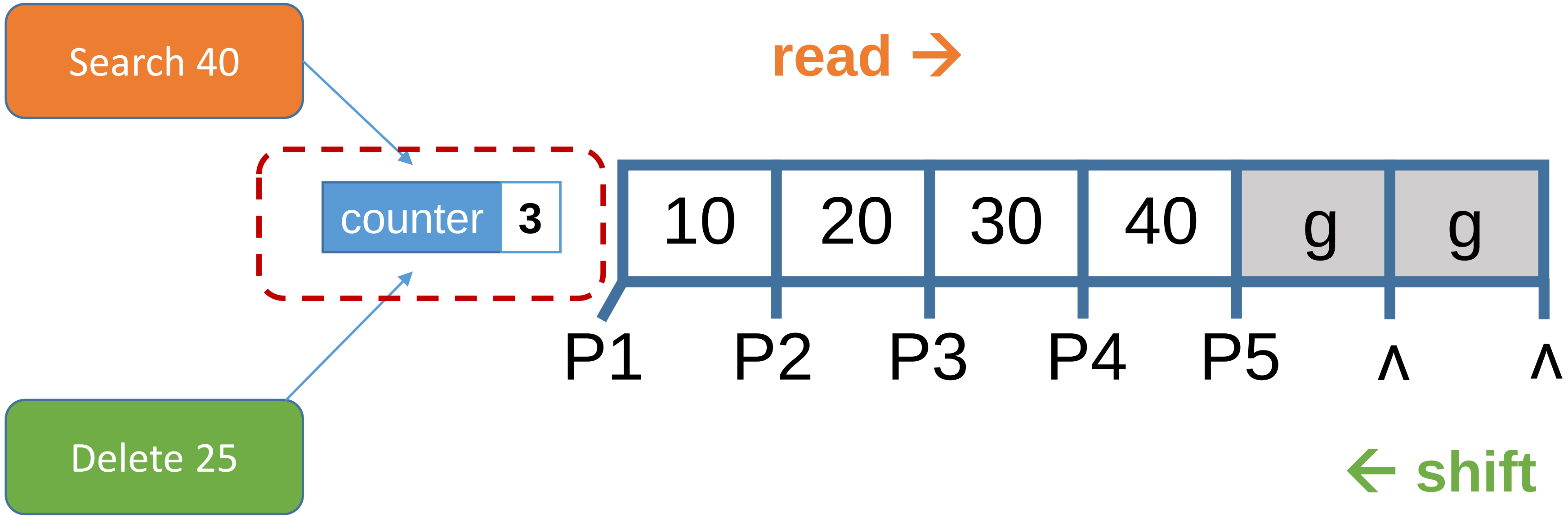
Lock-Free Search

- Direction flag:
  - Even Number
    - Insertion shifts to the right.
    - Search must scan from Left to Right
  - Odd Number
    - Deletion shifts to the left.
    - Search must scan from Right to Left



Lock-Free Search

- Direction flag:
  - Even Number
    - Insertion shifts to the right.
    - Search must scan from Left to Right
  - Odd Number
    - Deletion shifts to the left.
    - Search must scan from Right to Left



The read transaction has to check the counter once again to make sure the counter has not changed. Otherwise, search the node again.

Lock-Free Search Consistency Model

Transaction A

```
BEGIN
INSERT 10
SUSPENDED
```

```
WAKE UP
```

ABORT

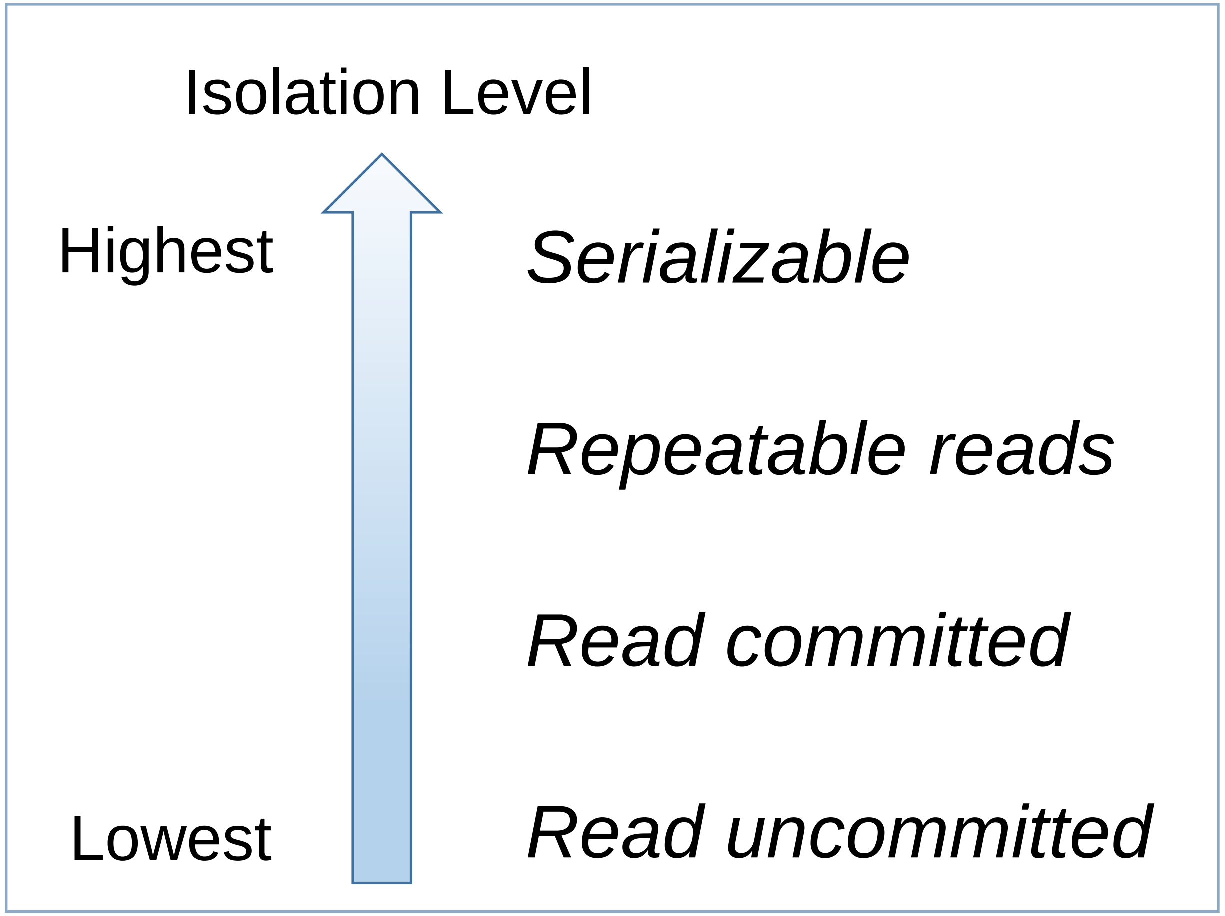
Transaction B

```
BEGIN
SEARCH 10(FOUND)
COMMIT
```

Dirty reads problem

The ordering of Transaction A and Transaction B cannot be determined

Lock-Free Search Consistency Model



Isolation Level

Highest

Lowest

Serializable

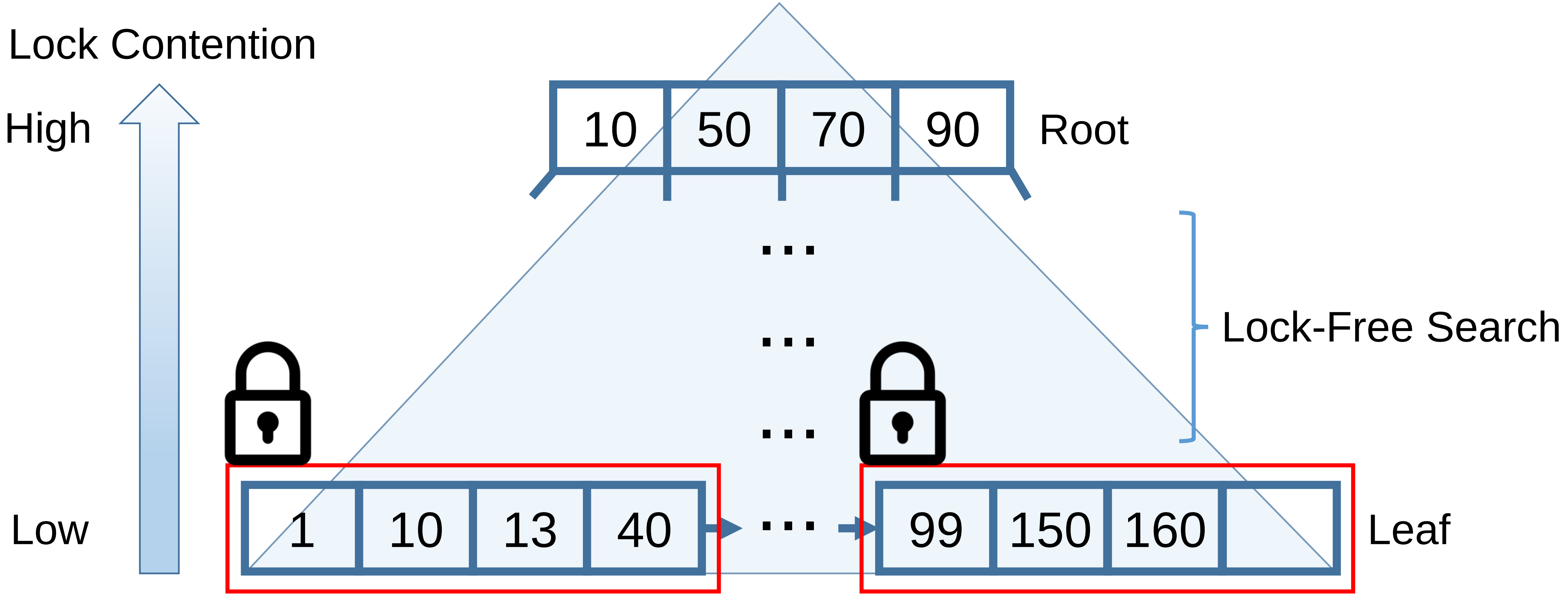
Repeatable reads

Read committed

Read uncommitted

Our Lock-Free Search supports low isolation level

Lock-Free Search Consistency Model

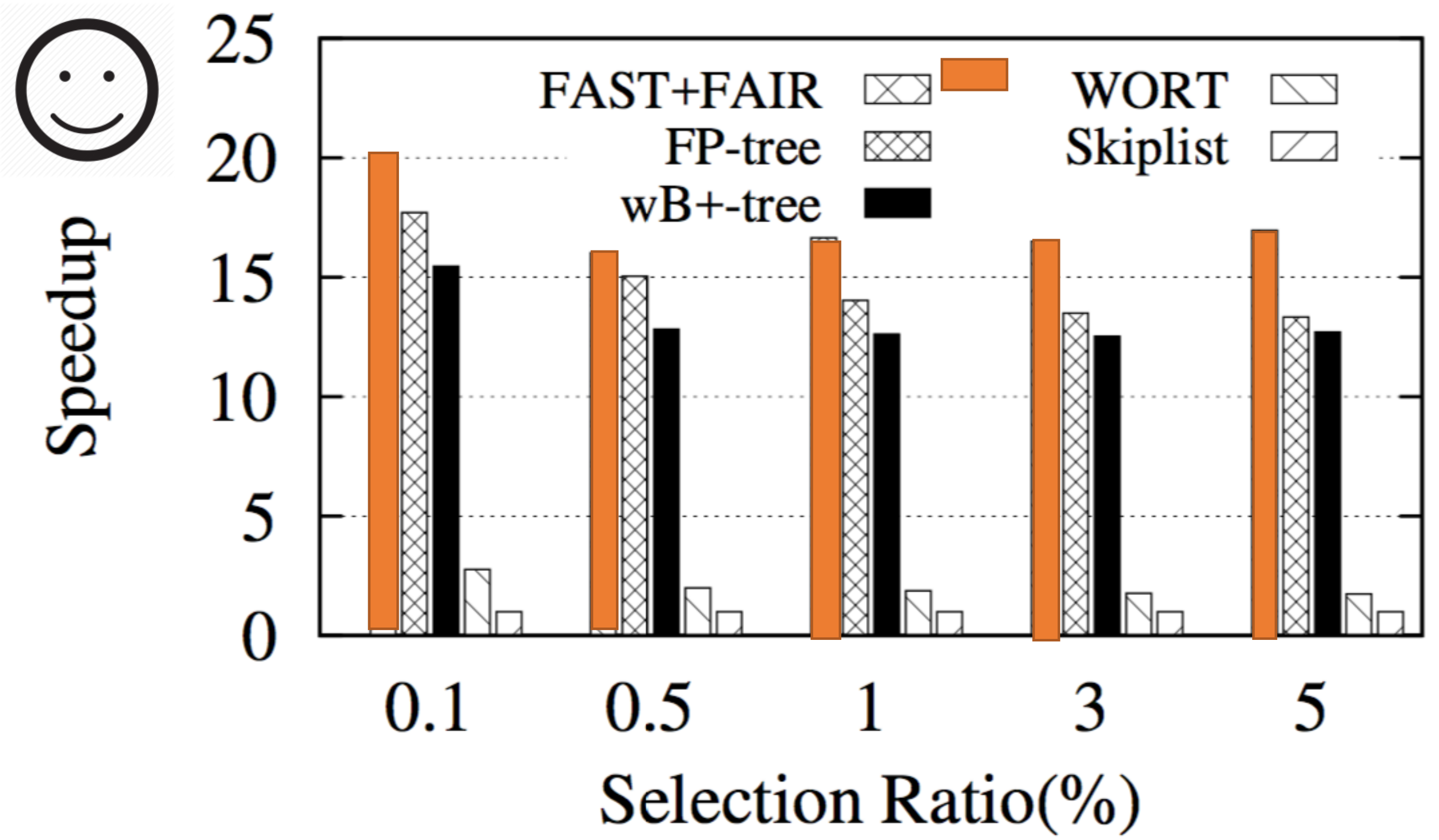


For higher isolation level, read lock is necessary for leaf nodes

Experimental Environments

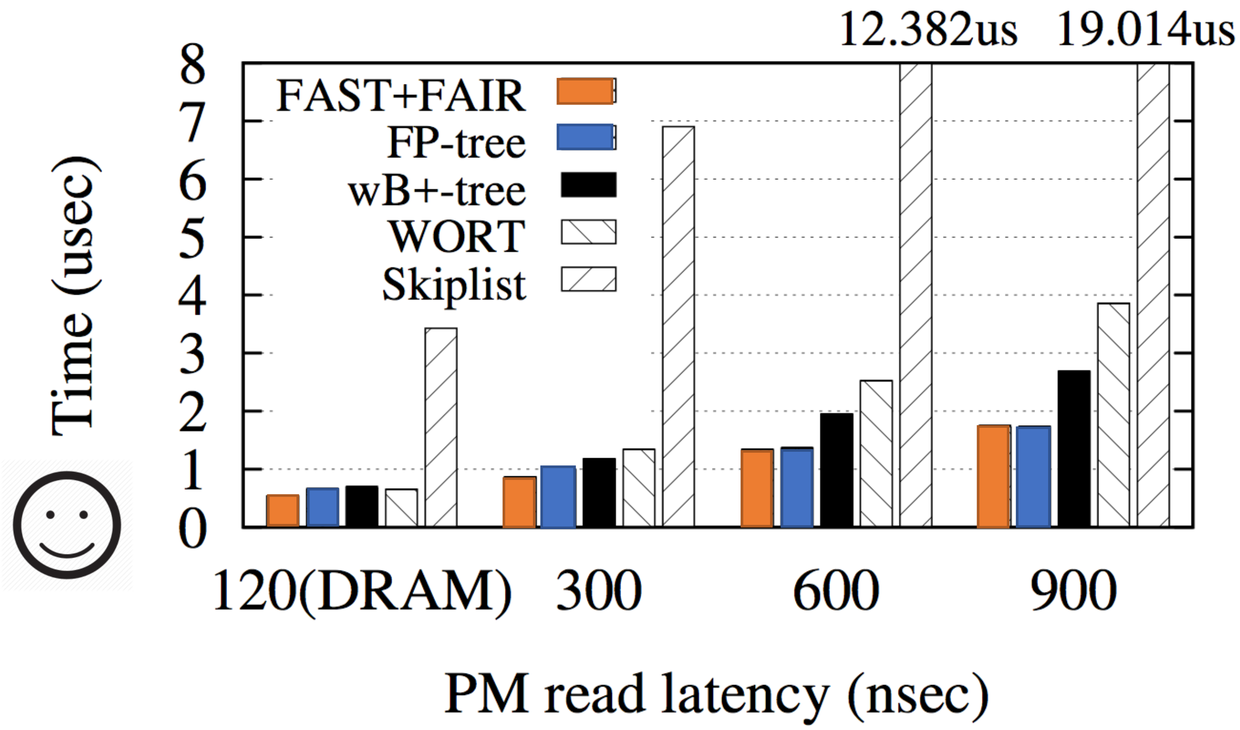
- Xeon Haswell-Ex E7-4809 v3 processors
  - 2.0 GHz, 16 vCPUs with hyper-threading enabled, and 20 MB L3 cache
  - Total Store Ordering (TSO) is guaranteed
- g++ 4.8.2 with -O3
- PM latency
  - Read latency
    - A DRAM-based PM latency emulator, Quartz
  - Write latency
    - Injecting delay

Range Query Performance Improvement over Skiplist



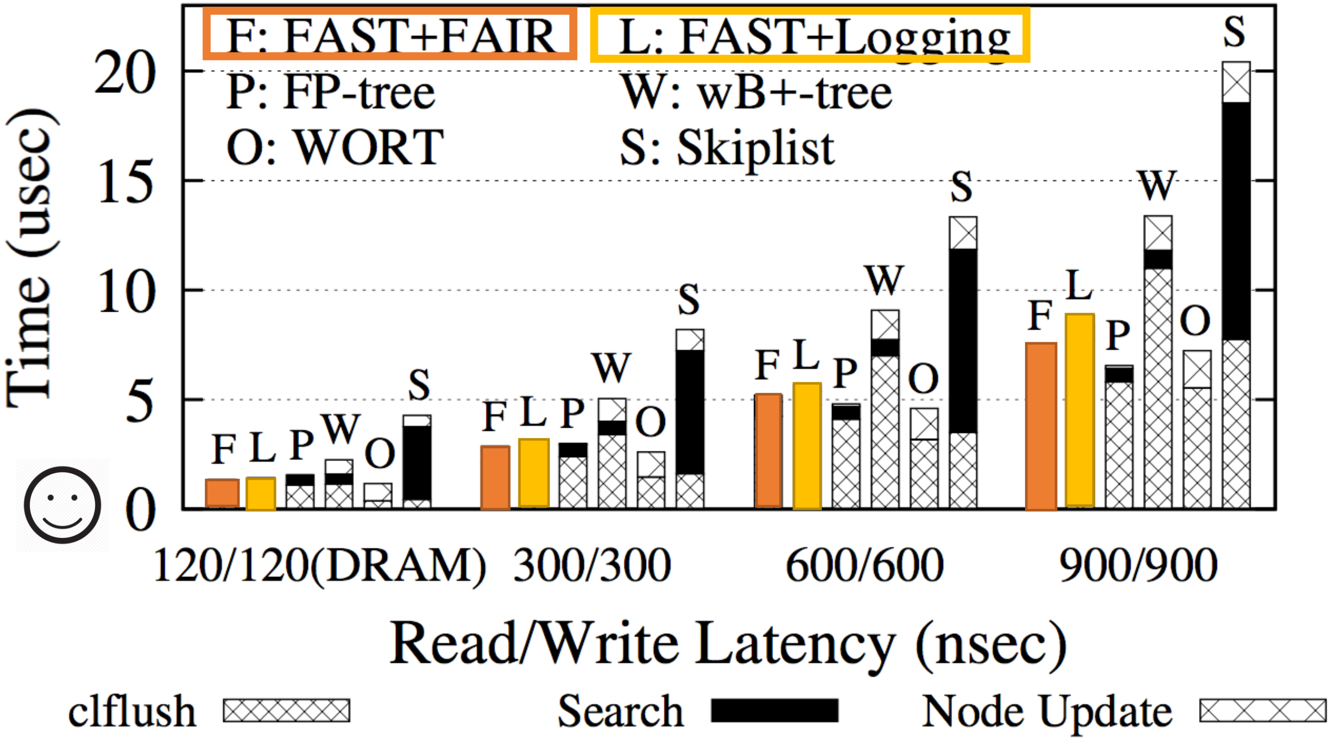
- Sorted keys, cache locality, and memory level parallelism
  → up to 20X speed up

Exact Match Query Performance with varying PM Latencies



FAST+FAIR→ FP-Tree → wB+-Tree → WORT → Skiplist

Breakdown of Time spent for Insertion

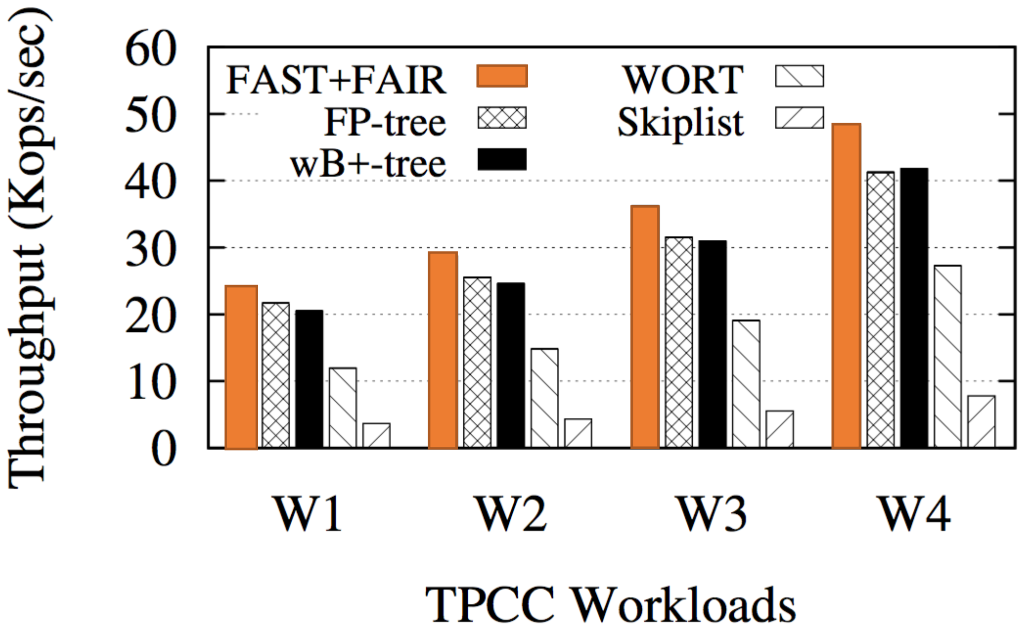


- clflush: I/O time
- Search: Tree traversal time
- Node Update: Computation time

WORT, FAST+FAIR, FP-Tree → FAST+Logging → wB+-Tree → Skiplist

- *FAST+Logging* uses logging instead of *FAIR* when splitting a node

TPC-C Benchmark



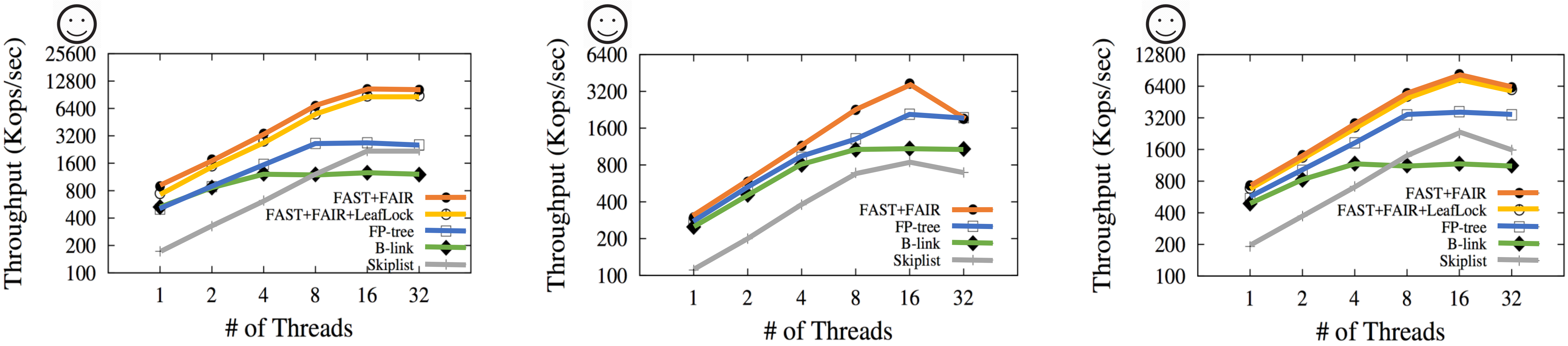
	New Order	Payment	Order Status	Delivery	Stock Level
W1	34%	43%	5%	4%	14%
W2	27%	43%	15%	4%	11%
W3	20%	43%	25%	4%	8%
W4	13%	43%	35%	4%	5%

Specification of TPCC workloads

More Range Queries

- FAST+FAIR consistently outperforms other indexes because of its good insertion performance and superior range query performance

Lock-Free Search with concurrent threads



(a) 50M Search

(b) 50M Insertion

(c) 200M Search / 50M Insertion / 12.5M Deletion

- Lock-free search with FAST+FAIR shows high scalability and performance
- FAST+FAIR+LeafLock shows comparable scalability and provides high concurrency level

Conclusion

- We designed a byte addressable persistent B+-Tree that
  - stores keys in order
  - avoids expensive logging

- FAST and FAIR always transform B+-Trees into consistent/transient inconsistent B+-Trees

- Lock-Free search
  - By tolerating transient inconsistency

Thank you

Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

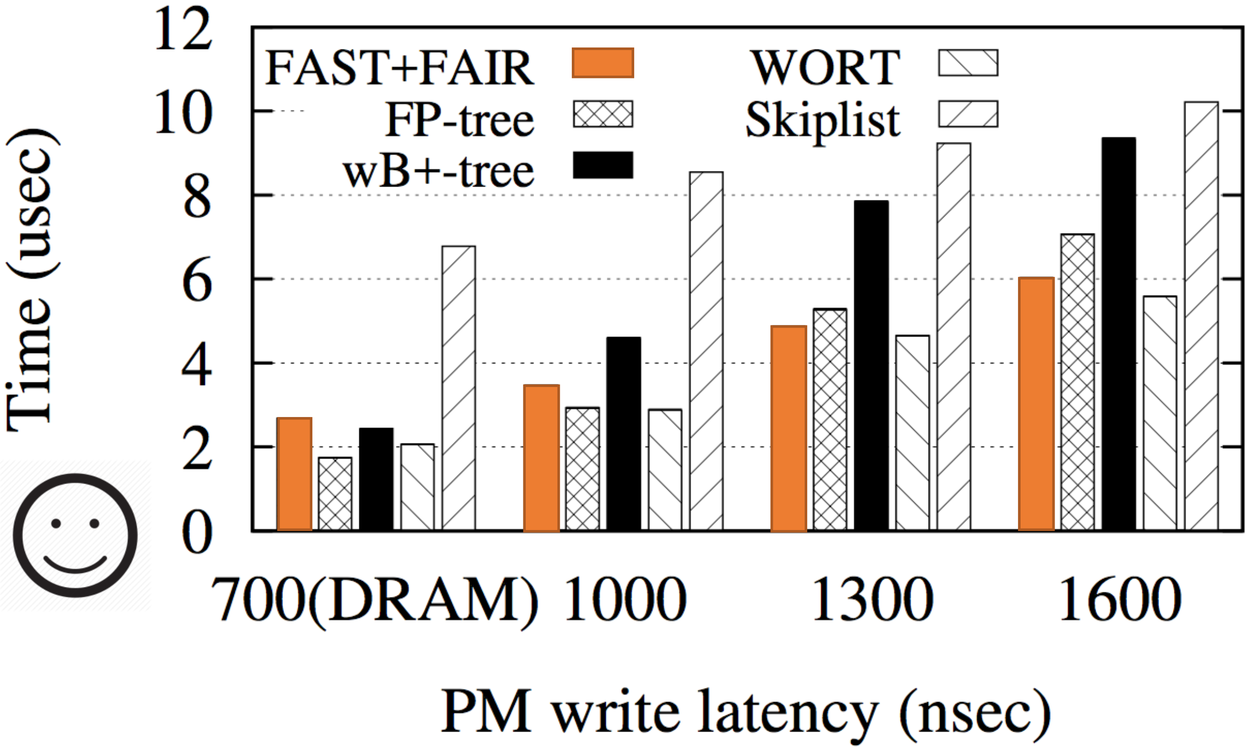
Deukyeon Hwang UNIST

Wook-Hee Kim UNIST

Youjip Won Hanyang Univ.

Beomseok Nam UNIST

Insertion for Non-TSO with varying write latency



- To guarantee the order of instructions, the *dmb* instruction is used for FAST+FAIR
- Although there is an overhead by *dmb*, FAST+FAIR is less affected by latency