WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems
Lecture 13: Blink Tree

Xiangyao Yu
10/20/2020

Announcement

Project proposal deadline: **Oct. 25**

- Submission website: https://wisc-cs764-f21.hotcrp.com
- Format: VLDB or SIGMOD preferred
- Make sure to include: project name, author list, background and motivation, task plan, timeline

Guest lecture next Monday (Oct. 25) from Amazon

- The lecture is offered in **online mode**
- No round-table discussion

Today’s Paper: B-tree Locking

Efficient Locking for Concurrent Operations on B-Trees

PHILIP L. LEHMAN
Carnegie-Mellon University
and
S. BING YAO
Purdue University

The B-tree and its variants have been found to be highly useful (both theoretically and in practice) for storing large amounts of information, especially on secondary storage devices. We examine the problem of overcoming the inherent difficulty of concurrent operations on such structures, using a practical storage model. A single additional "link" pointer in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Our solution compares favorably with earlier solutions in that the locking scheme is simpler (no read-locks are used) and only a (small) constant number of nodes are locked by any update process at any given time. An informal correctness proof for our system is given.

Key Words and Phrases: database, data structures, B-tree, index organizations, concurrent algorithms, concurrency controls, locking protocols, correctness, consistency, multiway search trees
CR Categories: 3.73, 3.74, 4.32, 4.33, 4.34, 5.24

1. INTRODUCTION

The B-tree [2] and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [7]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database applications.

A topic of current interest in database design is the construction of databases that can be manipulated concurrently and correctly by several processes. In this

ACM Trans. Database Syst. 1981

Agenda

B-Tree Index

Lock coupling

B^{link}-tree

– Search

– Insert

Optimistic lock coupling (OLC)

Index

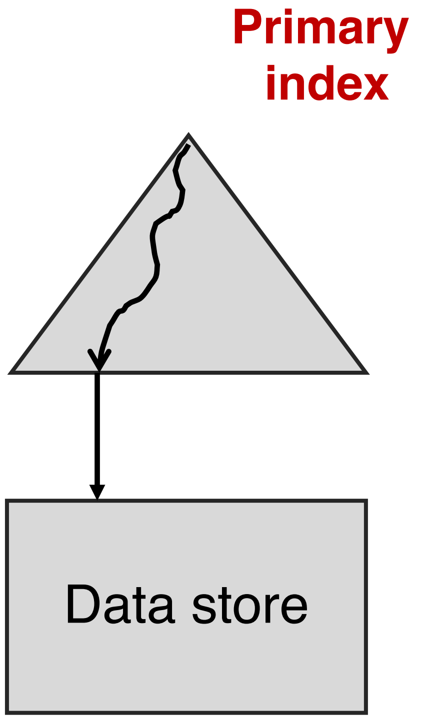
Index: Accelerate data retrieval operations in a database table

- E.g., random lookup, range scan

Index

Index: Accelerate data retrieval operations in a database table

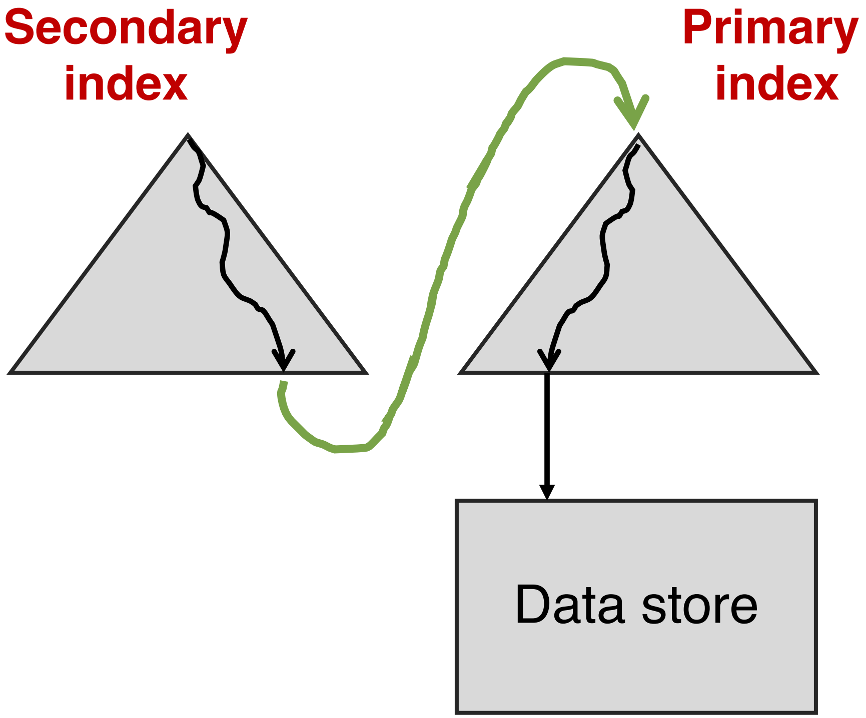
- E.g., random lookup, range scan



Index

Index: Accelerate data retrieval operations in a database table

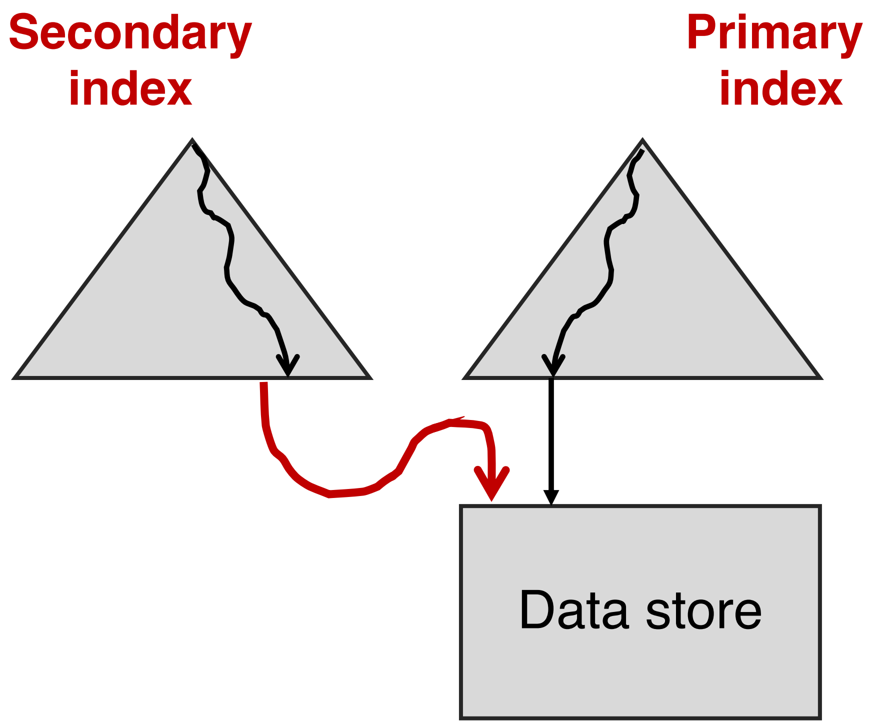
- E.g., random lookup, range scan



Index

Index: Accelerate data retrieval operations in a database table

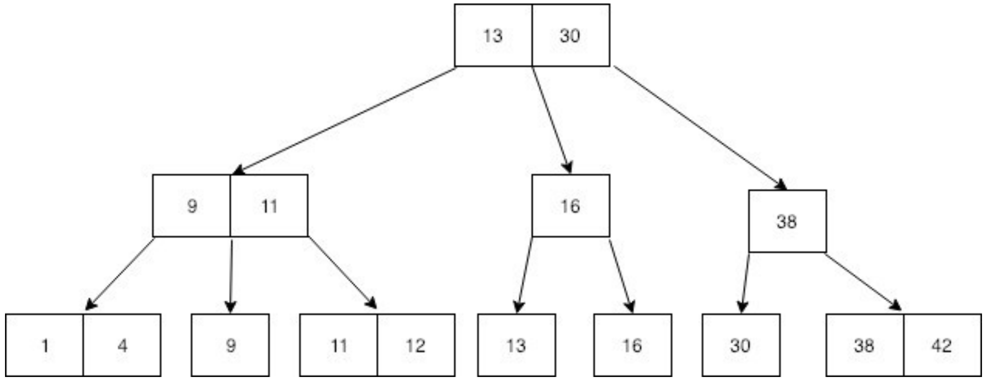
- E.g., random lookup, range scan



B-tree

Balanced tree data structure

- Data is sorted
- Supports: search, sequential scan, inserts, and deletes



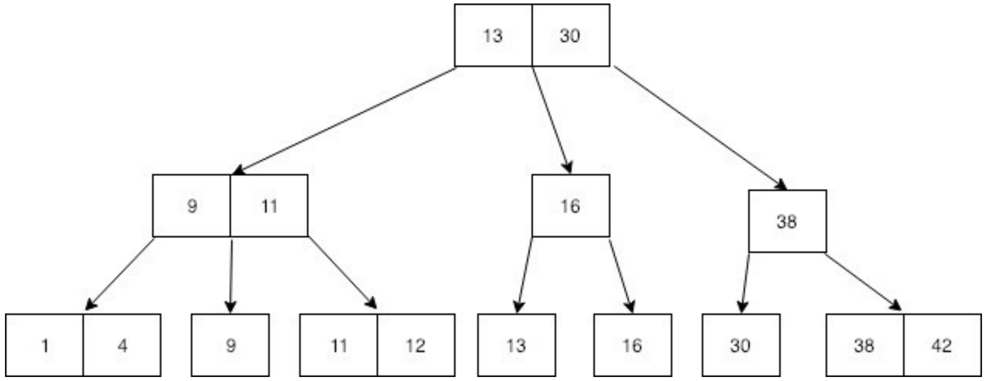
B-tree

Balanced tree data structure

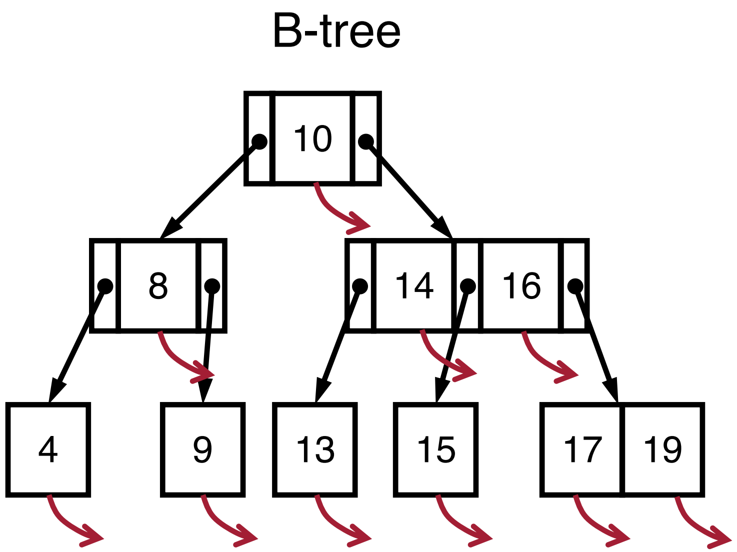
- Data is sorted
- Supports: search, sequential scan, inserts, and deletes

Properties

- Every node contains *k* to *2k* keys (except root)
- All leaf nodes are at the same level
- *k* is typically large; a lookup traverses a small number of levels

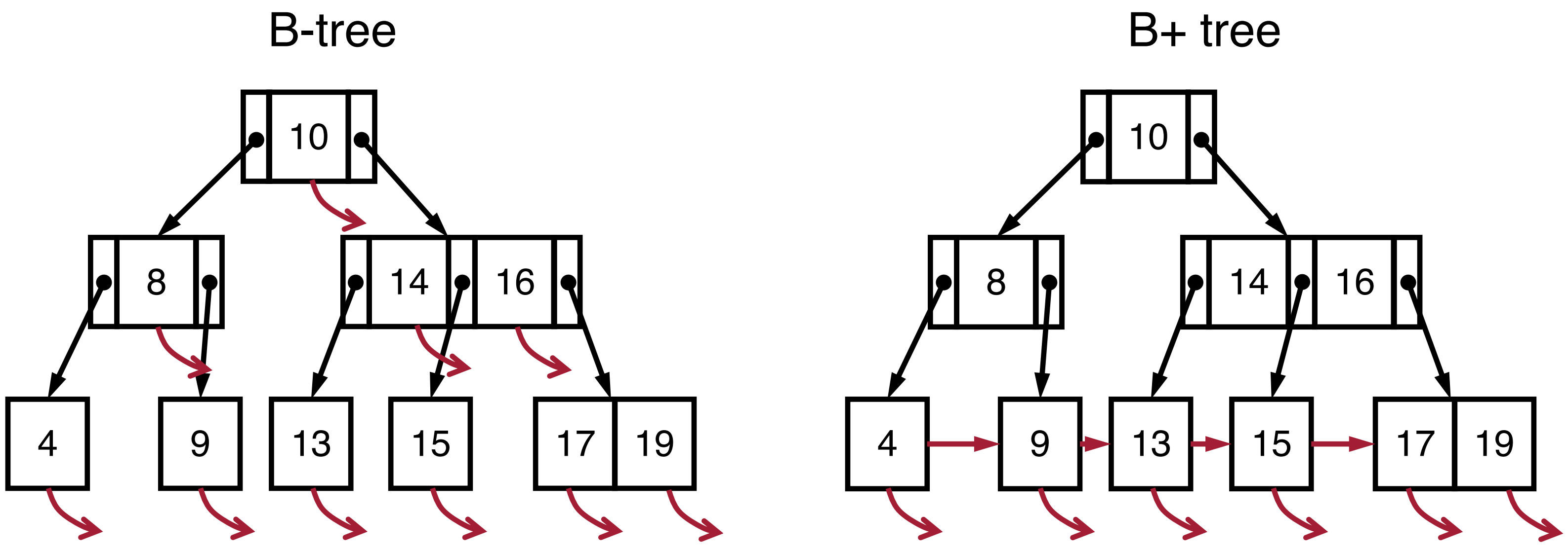


B-tree vs. B+ Tree vs. B* Tree



B-tree: data pointers stored in all nodes

B-tree vs. B+ Tree vs. B* Tree

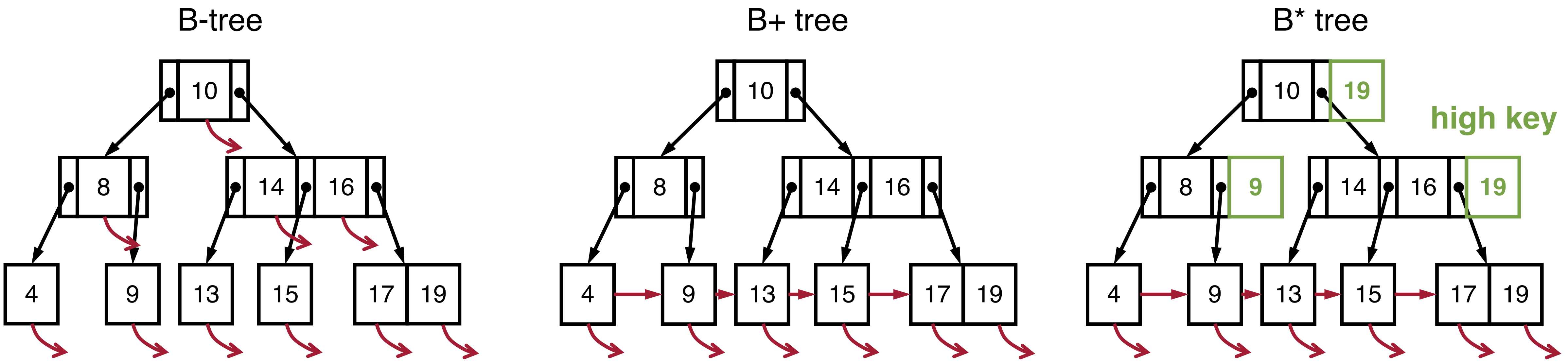


B-tree: data pointers stored in all nodes

B+ tree:

- Data pointers stored only in leaf nodes
- The leaf nodes are linked

B-tree vs. B+ Tree vs. B* Tree



B-tree: data pointers stored in all nodes

B+ tree:

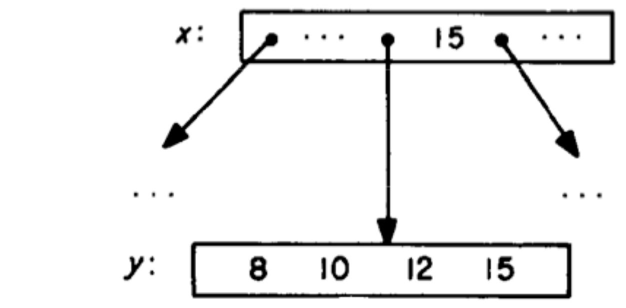
- Data pointers stored only in leaf nodes
- The leaf nodes are linked

B* tree is a misused term in B-tree literature

- Typically means a variant of B+ tree in which each node is least 2/3 full
- In this paper: B+ tree with high key appended to non-leaf nodes (upper bound on values)

Insert Example

Assume k = 2 (at most 4 keys per node)

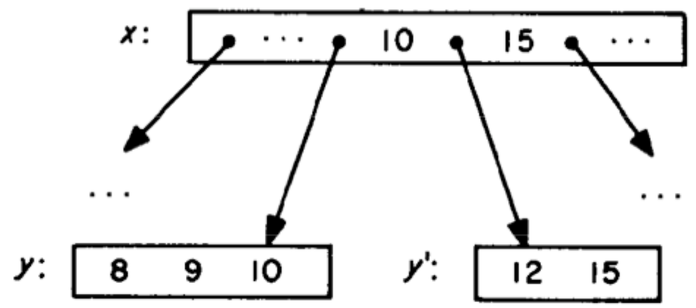


(a)

<u>insert(9)</u>

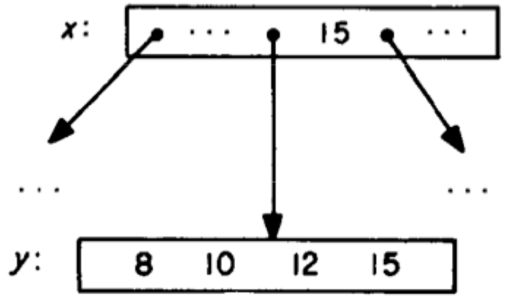
$A \leftarrow \text{read}(x)$

examine A; get ptr to y
$A \leftarrow \text{read}(y)$
insert 9 into A; must split into A, B
put(B, y')
put(A, y)
Add to node x a pointer to node y'.



Search Example

Assume k = 2 (at most 4 keys per node)



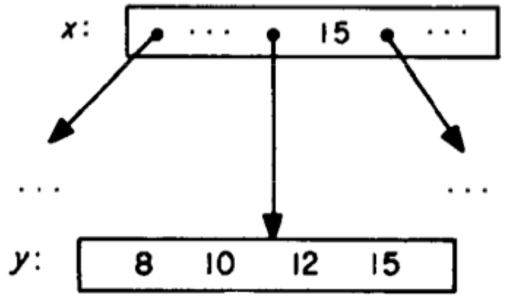

```
search(15)
C ← read(x)

examine C; get ptr to y






C ← read(y)
```

Concurrency Challenge

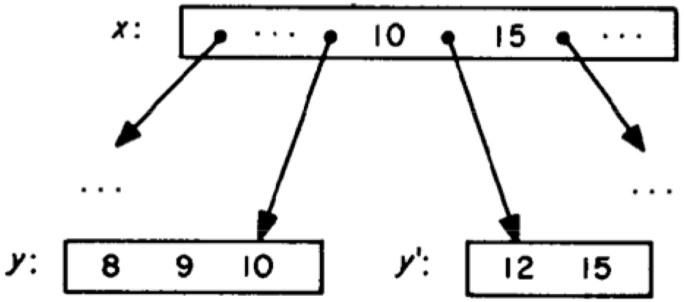
Assume k = 2 (at most 4 keys per node)

Concurrent search and insert can cause problems

	search(15)	insert(9)
1.	$C \leftarrow \text{read}(x)$	
2.		$A \leftarrow \text{read}(x)$
3.	examine C; get ptr to y	
4.		examine A; get ptr to y
5.		$A \leftarrow \text{read}(y)$
6.		insert 9 into A; must split into A, B
7.		put(B, y')
8.		put(A, y)
9.		Add to node x a pointer to node y'.
10.	$C \leftarrow \text{read}(y)$	
11.	*error*: 15 not found!	



(a)



Lock Coupling

A node is **unsafe** (wrt. insertion) if it is full (i.e., contains 2k keys)

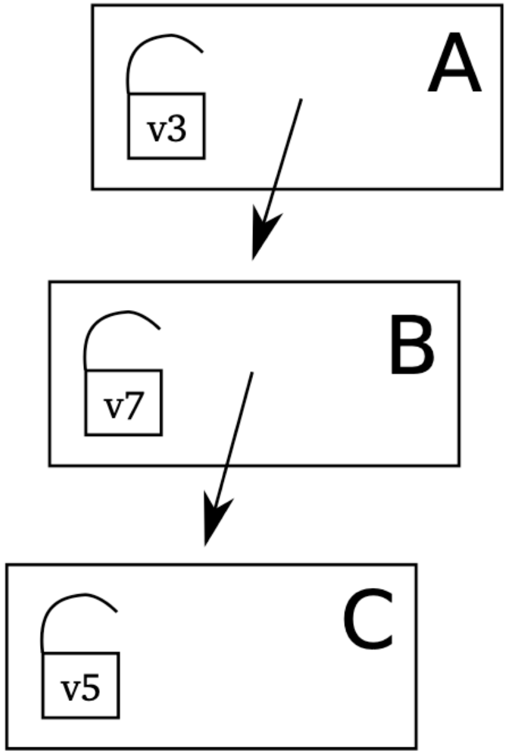
Lock Coupling

A node is **unsafe** (wrt. insertion) if it is full (i.e., contains 2k keys)

Lock coupling (aka. lock crabbing)

- – Lock parent
- – Access parent
- – Lock child
- – Release parent if child is safe

1. lock node A
2. access node A
3. lock node B
4. unlock node A
5. access node B
6. lock node C
7. unlock node B
8. access node C
9. unlock node C



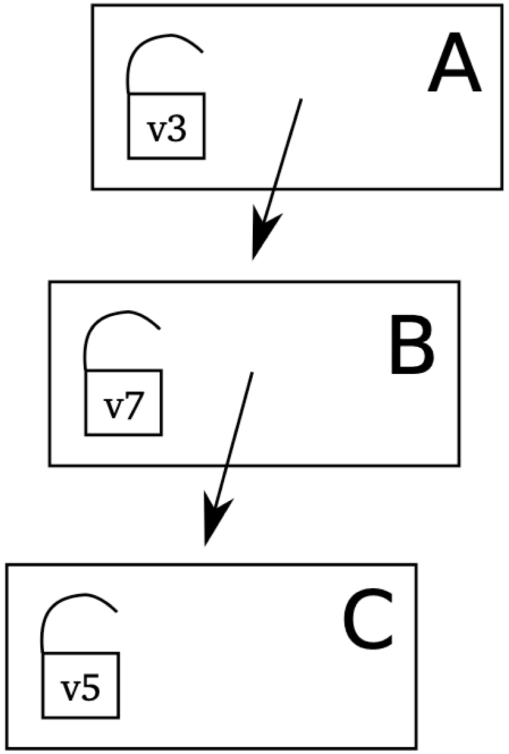
Lock Coupling

A node is **unsafe** (wrt. insertion) if it is full (i.e., contains 2k keys)

Lock coupling (aka. lock crabbing)

- Lock parent
- Access parent
- Lock child
- Release parent if child is safe

1. lock node A
2. access node A
3. lock node B
4. unlock node A
5. access node B
6. lock node C
7. unlock node B
8. access node C
9. unlock node C



What if the child is unsafe?

- One solution: split immediately if child is unsafe

Limitation of Lock Coupling

The root is locked for every index access and becomes a scalability bottleneck

Observation: root and upper levels are rarely changed; lock coupling is too conservative

Limitation of Lock Coupling

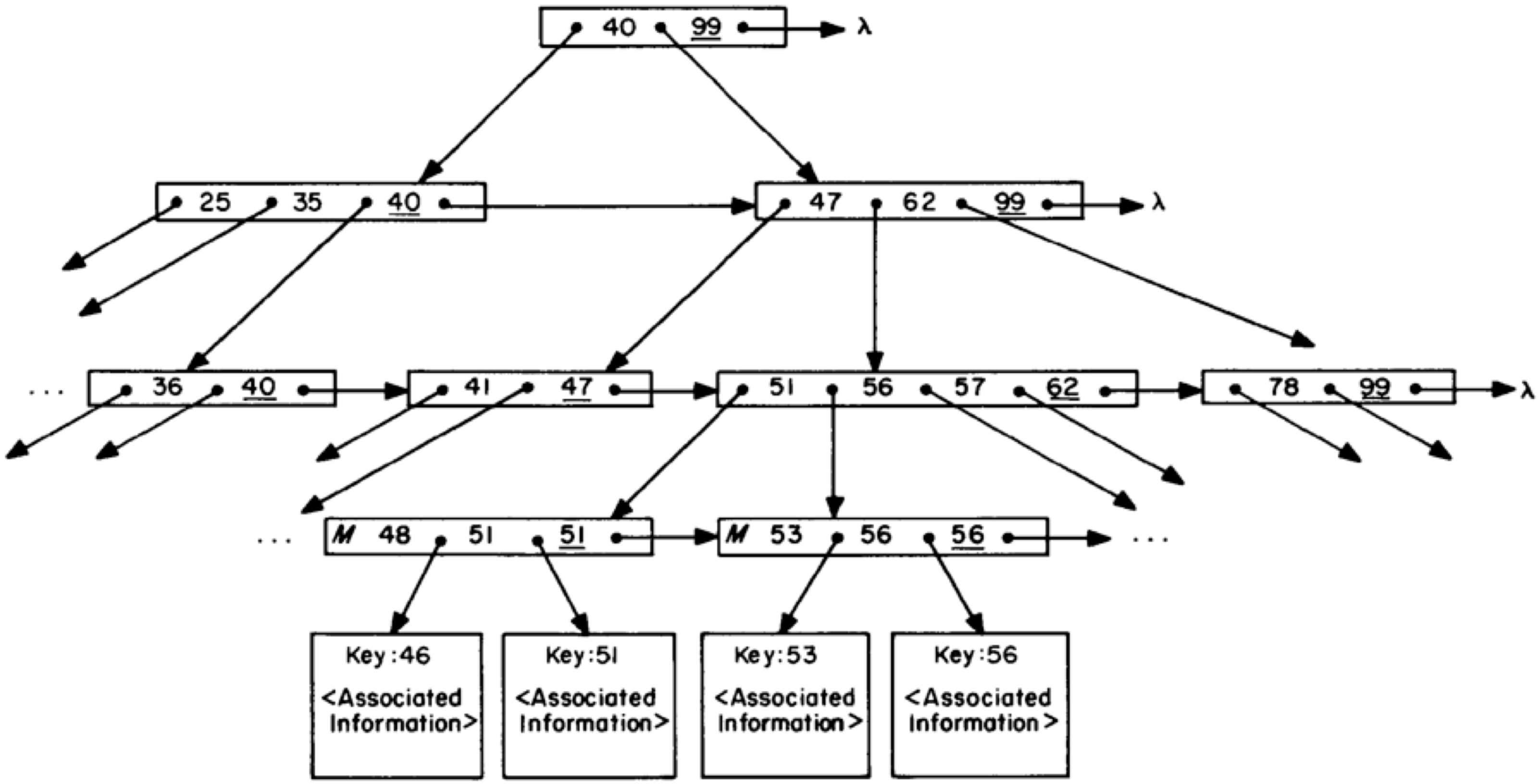
The root is locked for every index access and becomes a scalability bottleneck

Observation: root and upper levels are rarely changed; lock coupling is too conservative

Concurrency challenge: search may read wrong node due to split

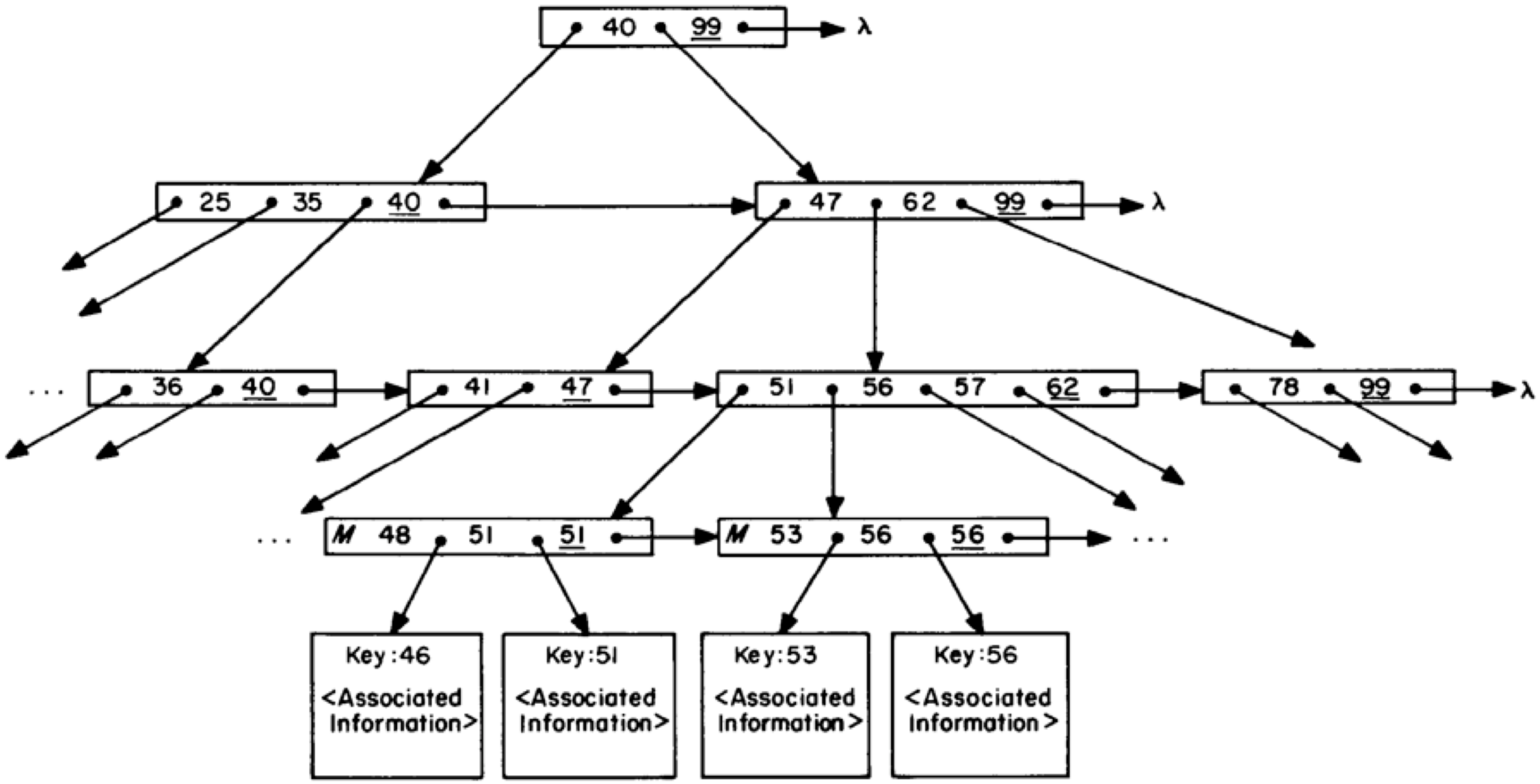
- **Lock coupling solution**: guard split using a lock
- **B^{link} tree solution**: allow search to find the right node

B^{link}-Tree



Feature 1: **link pointer** to next node at each level → **key idea**

B^{link}-Tree



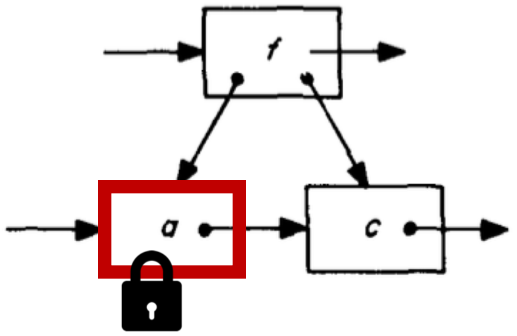
Feature 1: **link pointer** to next node at each level → **key idea**

Feature 2: **high key** for each node

B^{link}-Tree: Insert Algorithm

Insert to leaf if the leaf node if not full

Illustration of node split (node *a* is split into *a'* and *b'*)

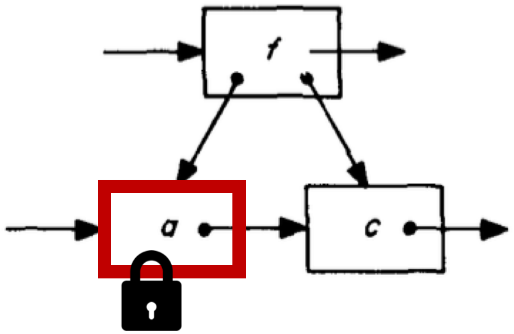


Before split

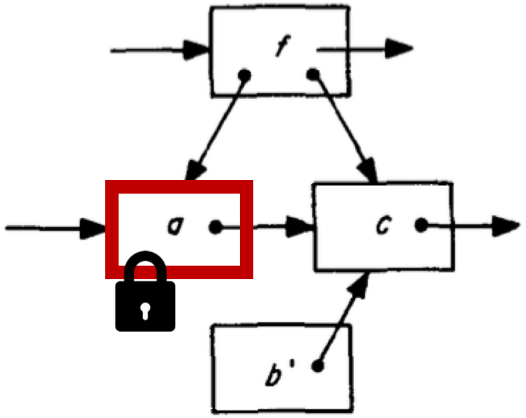
B^{link}-Tree: Insert Algorithm

Insert to leaf if the leaf node if not full

Illustration of node split (node *a* is split into *a'* and *b'*)



Before split

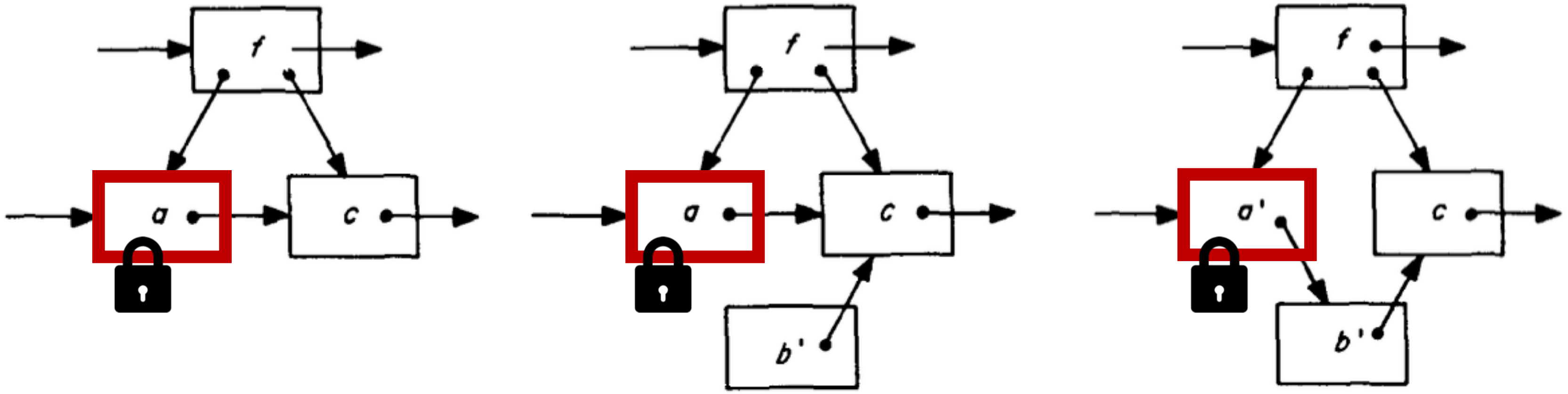


Step 1

B^{link}-Tree: Insert Algorithm

Insert to leaf if the leaf node if not full

Illustration of node split (node *a* is split into *a'* and *b'*)

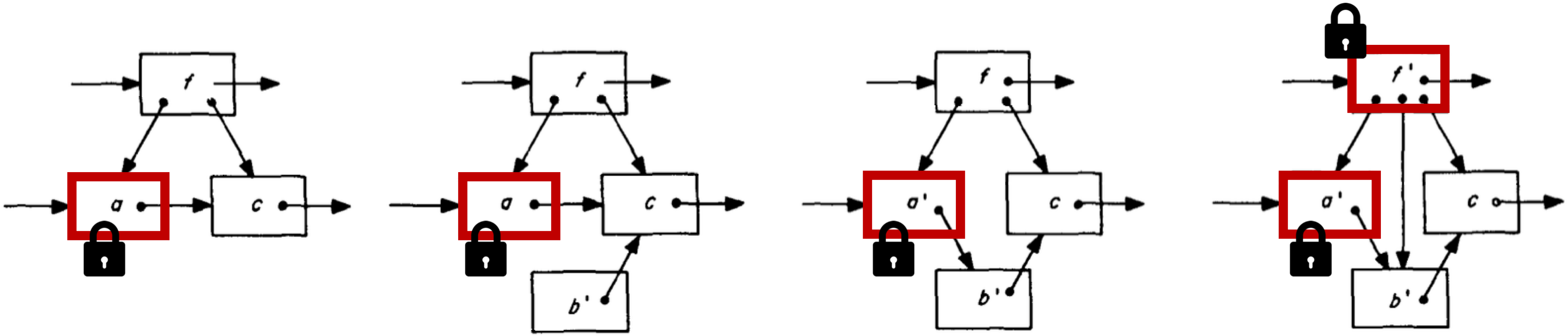


Before split Step 1 Step 2

B^{link}-Tree: Insert Algorithm

Insert to leaf if the leaf node if not full

Illustration of node split (node *a* is split into *a'* and *b'*)

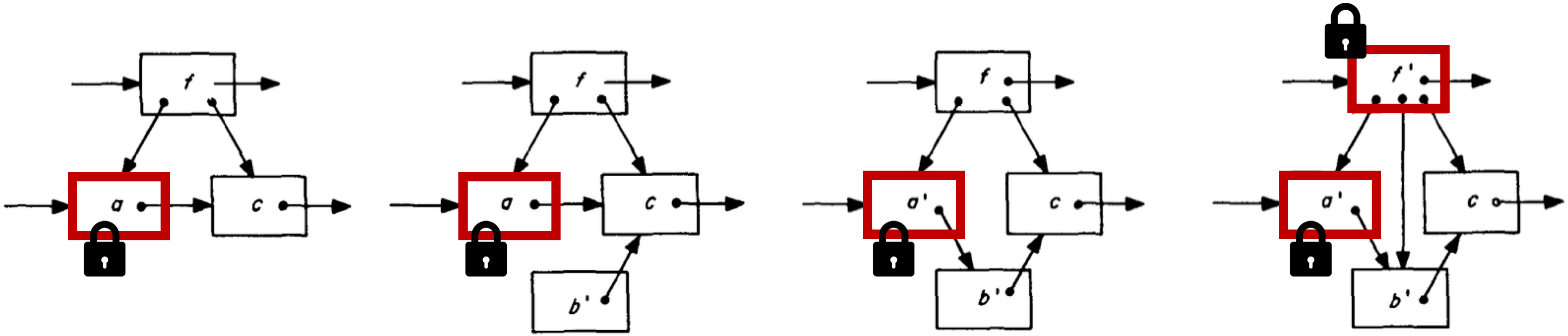


Before split | Step 1 | Step 2 | Step 3

B^{link}-Tree: Insert Algorithm

Insert to leaf if the leaf node if not full

Illustration of node split (node *a* is split into *a'* and *b'*)



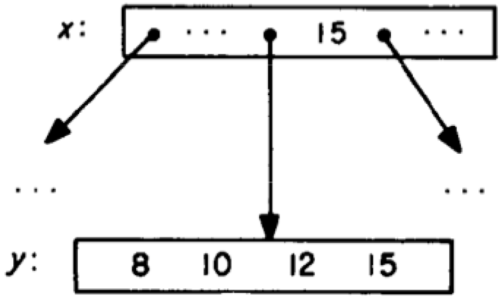
Before split | Step 1 | Step 2 | Step 3

Q: What if another txn searches a key in b' before step 3 finishes?

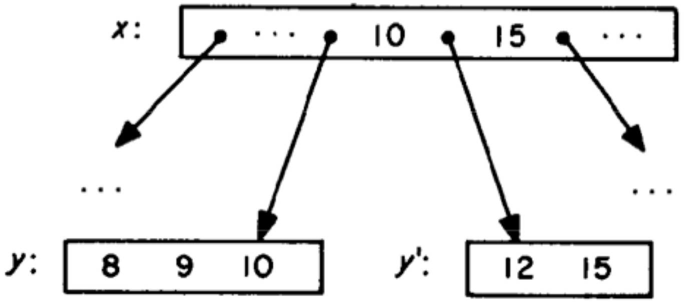
Concurrent Search & Insert

Assume k = 2 (at most 4 keys per node)

Concurrency problem is solved in B^{link} tree



(a)



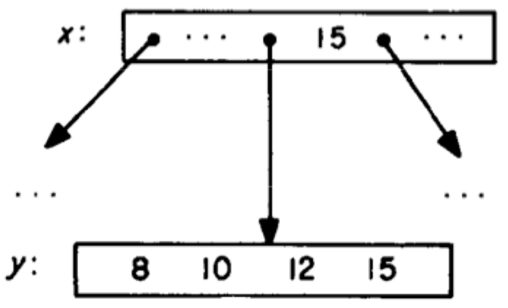
	search(15)	insert(9)
1.	$C \leftarrow$ read(x)	
2.		$A \leftarrow$ read(x)
3.	examine C; get ptr to y	
4.		examine A; get ptr to y
5.		$A \leftarrow$ read(y)
6.		insert 9 into A; must split into A, B
7.		put(B, y')
8.		put(A, y)
9.		Add to node x a pointer to node y'.
10.	$C \leftarrow$ read(y)	
11.	*error*: 15 not found!	

Concurrent Search & Insert

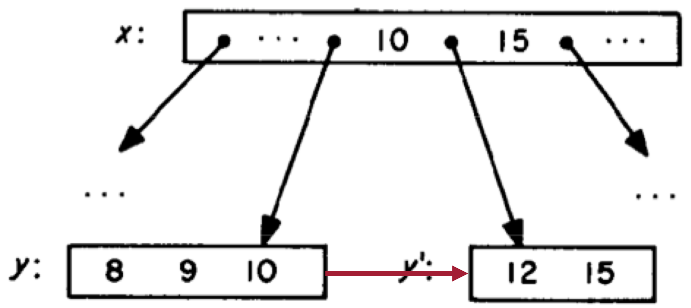
Assume k = 2 (at most 4 keys per node)

Concurrency problem is solved in B^{link} tree

High key indicates when to follow link pointer



(a)



	search(15)	insert(9)
1.	$C \leftarrow$ read(x)	
2.		$A \leftarrow$ read(x)
3.	examine C; get ptr to y	
4.		examine A; get ptr to y
5.		$A \leftarrow$ read(y)
6.		insert 9 into A; must split into A, B
7.		put(B, y')
8.		put(A, y)
9.		Add to node x a pointer to node y'.
10.	$C \leftarrow$ read(y)	
~~11.~~	~~error: 15 not found!~~	

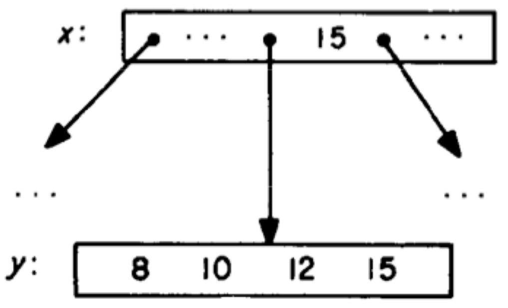
15 is found following the link pointer

Concurrent Search & Insert

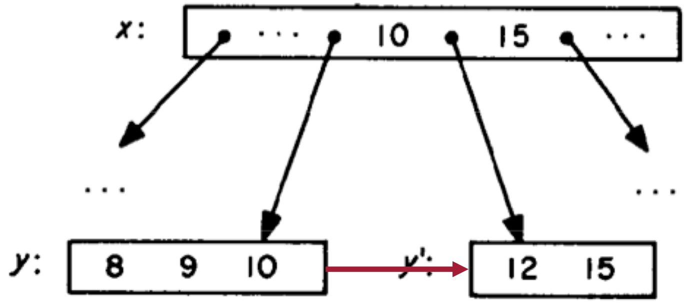
Assume k = 2 (at most 4 keys per node)

Concurrency problem is solved in B^{link} tree

High key indicates when to follow link pointer



(a)



	search(15)	insert(9)
1.	$C \leftarrow$ read(x)	
2.		$A \leftarrow$ read(x)
3.	examine C; get ptr to y	
4.		examine A; get ptr to y
5.		$A \leftarrow$ read(y)
6.		insert 9 into A; must split into A, B
7.		put(B, y')
8.		put(A, y)
9.		Add to node x a pointer to node y'.
10.	$C \leftarrow$ read(y)	
~~11.~~	~~*error*: 15 not found!~~	

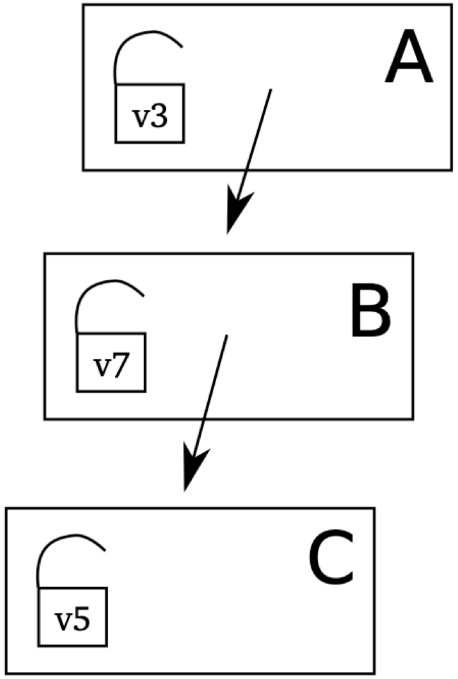
15 is found following the link pointer

Q: Can the algorithm work without high keys

Optimistic Lock Coupling (OLC)

Each tuple contains a 64-bit version counter

1. lock node A
2. access node A
3. lock node B
4. unlock node A
5. access node B
6. lock node C
7. unlock node B
8. access node C
9. unlock node C



1. read version v3
2. access node A
3. read version v7
4. validate version v3
5. access node B
6. read version v5
7. validate version v7
8. access node C
9. validate version v5

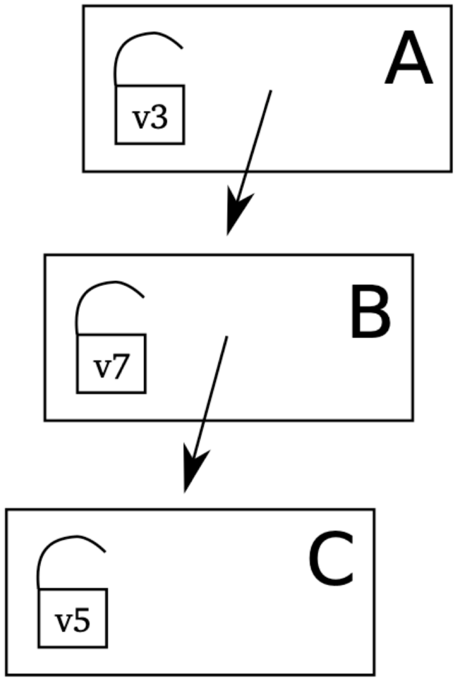
Optimistic Lock Coupling (OLC)

Each tuple contains a 64-bit version counter

1. lock node A
2. access node A

3. lock node B
4. unlock node A
5. access node B

6. lock node C
7. unlock node B
8. access node C
9. unlock node C



1. read version v3
2. access node A

3. read version v7
4. validate version v3
5. access node B

6. read version v5
7. validate version v7
8. access node C
9. validate version v5

No scalability bottleneck

- No write to shared memory during traversal
- Upon conflict, retry from root
- Performance similar to B^{link} tree

Evaluation

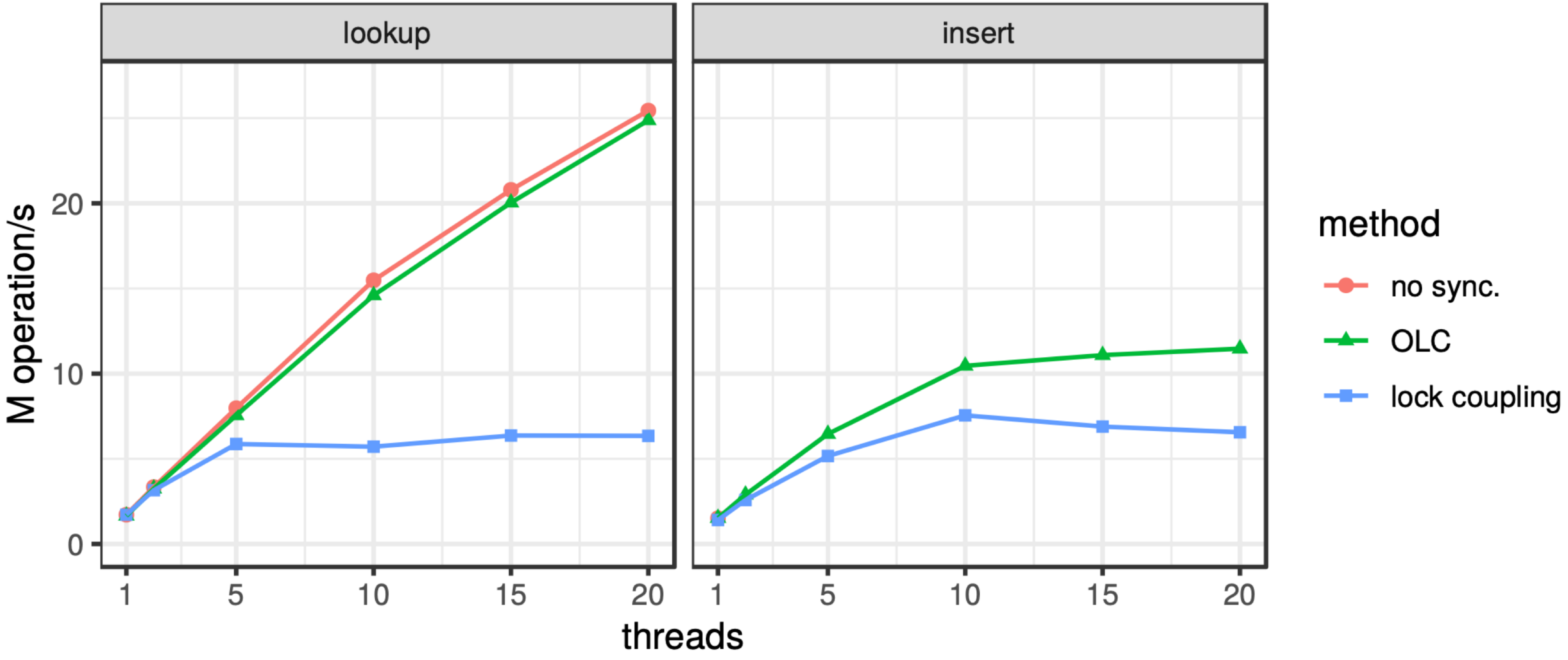


Figure 3: Scalability on 10-core system for B-tree operations (100M values).

Leis, Viktor et al. *Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. IEEE Data Eng. Bull.* 42 (2019): 73-84.

Q/A – Blink Tree

Why at most three locks are used? (why not two?)

Implemented on real systems?

What isolation level is Blink tree?

How is this different from B+ tree?

How does the algorithm perform in in-memory multicore processors?

How are the database indexes selected for a given workload?

Before Next Wednesday

Submit review for

- Viktor Leis, et al., The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. ICDE, 2013