15-721 DATABASE SYSTEMS

Lecture #06 – Indexing (Locking & Latching)

Andy Pavlo // Carnegie Mellon University // Spring 2016

TODAY'S AGENDA

Order Preserving Indexes

Index Locking & Latching

Prison Gang Tattoos

DATABASE INDEX

A data structure that improves the speed of data retrieval operations on a table at the cost of additional writes and storage space.

Indexes are used to quickly locate data without having to search every row in a table every time a table is accessed.

DATA STRUCTURES

Order Preserving Indexes

→ A tree-like structure that maintains keys in some sorted order.

→ Supports all possible predicates with O(log n) searches.

Hashing Indexes

→ An associative array that maps a hash of the key to a particular record.

→ Only supports equality predicates with O(1) searches.

DATA STRUCTURES

Order Preserving Indexes

→ A tree-like structure that maintains keys in some sorted order.

→ Supports all possible predicates with O(log n) searches.

Hashing Indexes

→ An associative array that maps a hash of the key to a particular record.

→ Only supports equality predicates with O(1) searches.

B-TREE VS. B+TREE

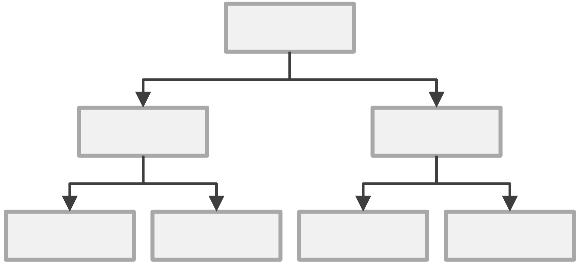
The original **B-tree** from 1972 stored keys + values in all nodes in the tree.

→ More memory efficient since each key only appears once in the tree.

A **B+tree** only stores values in leaf nodes. Inner nodes only guide the search process.

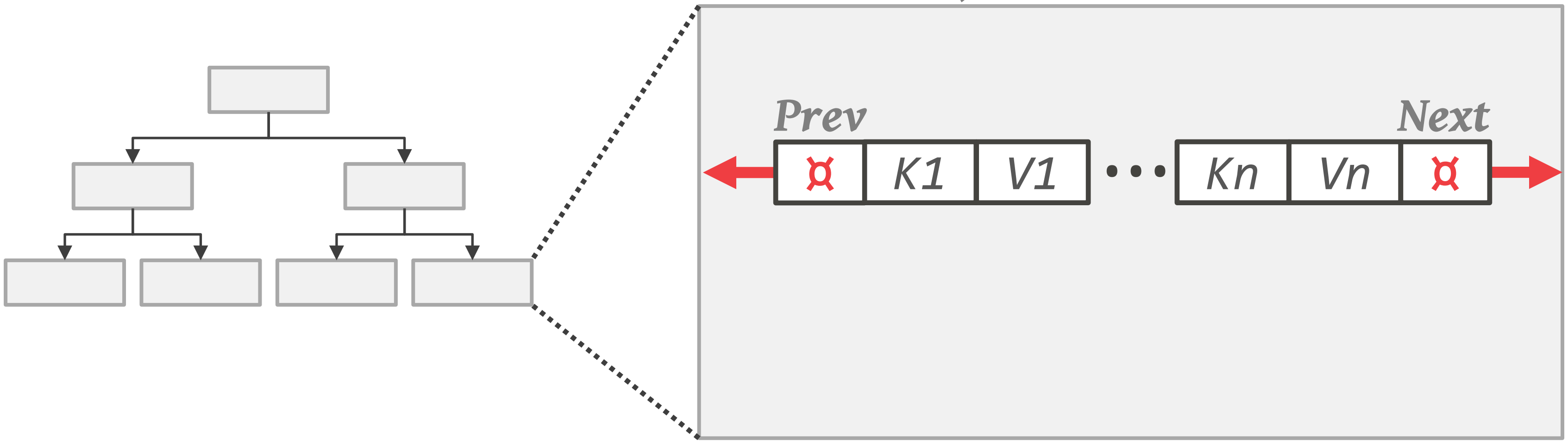
→ Easier to manage concurrent index access when the values are only in the leaf nodes.

B+TREE

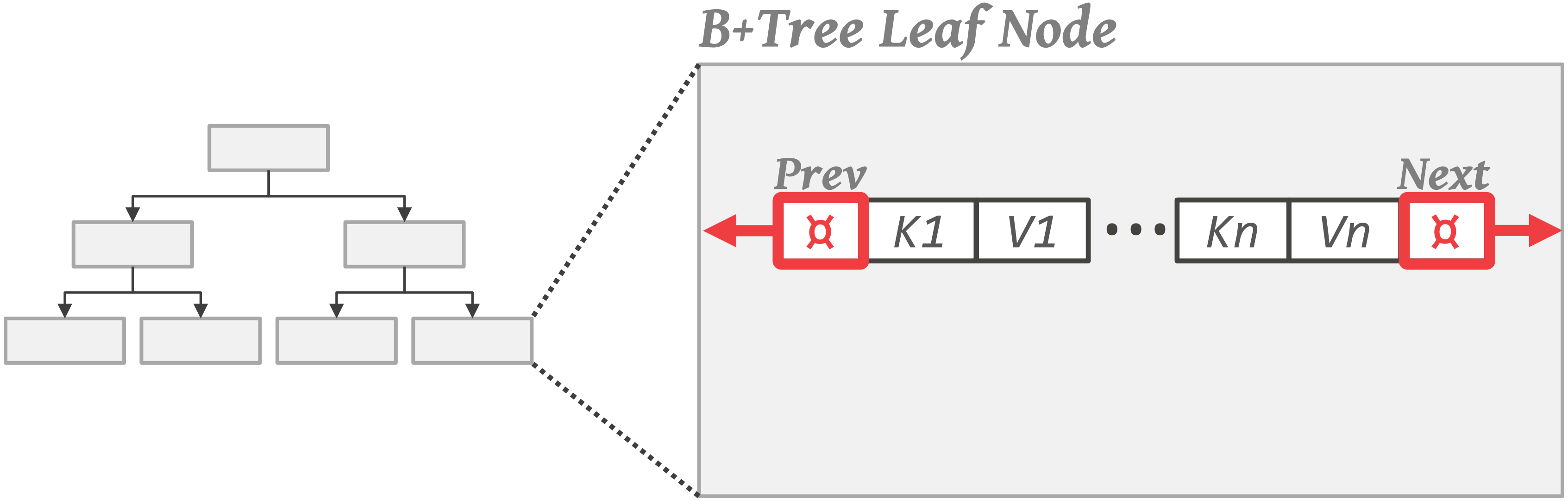


B+TREE

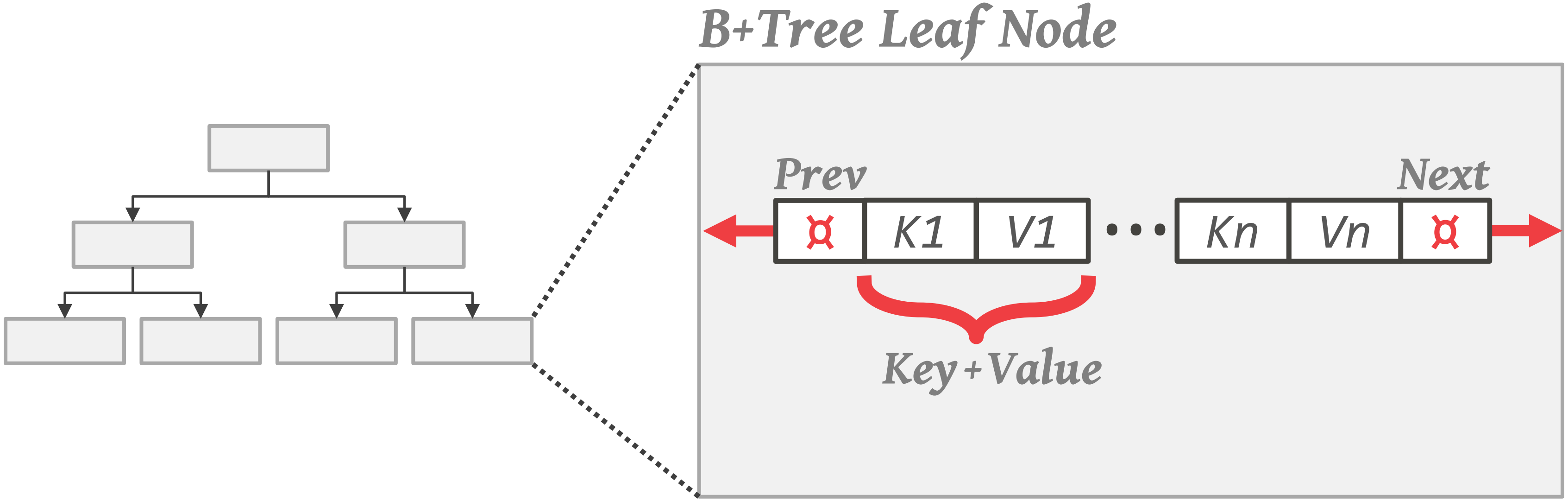
B+Tree Leaf Node



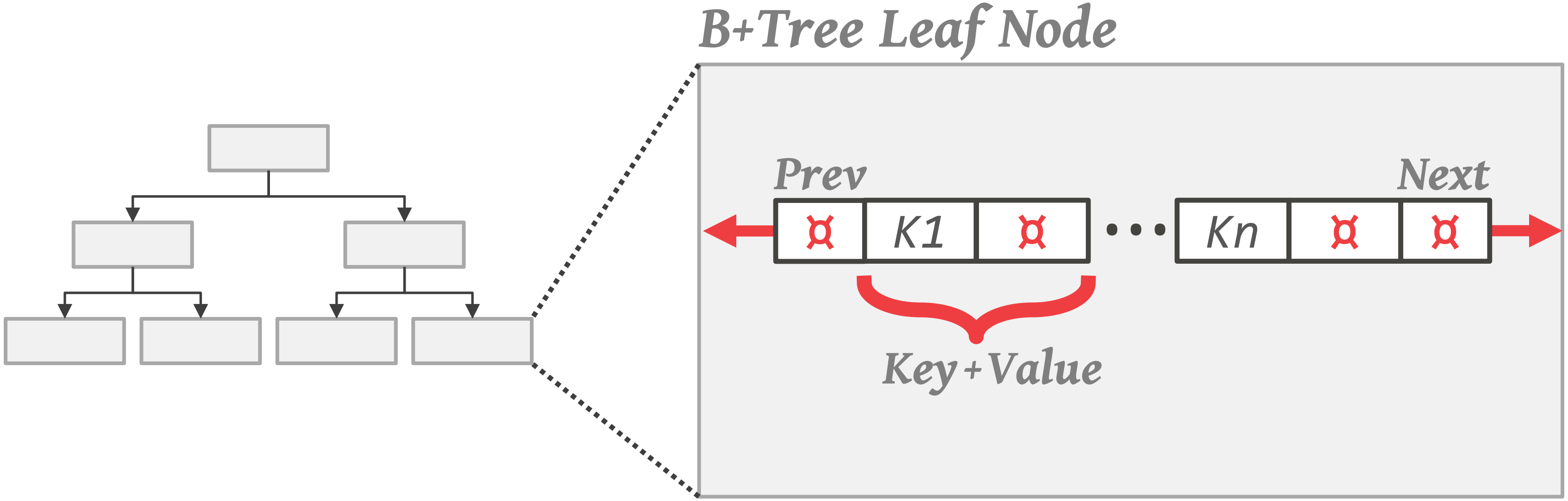
B+TREE



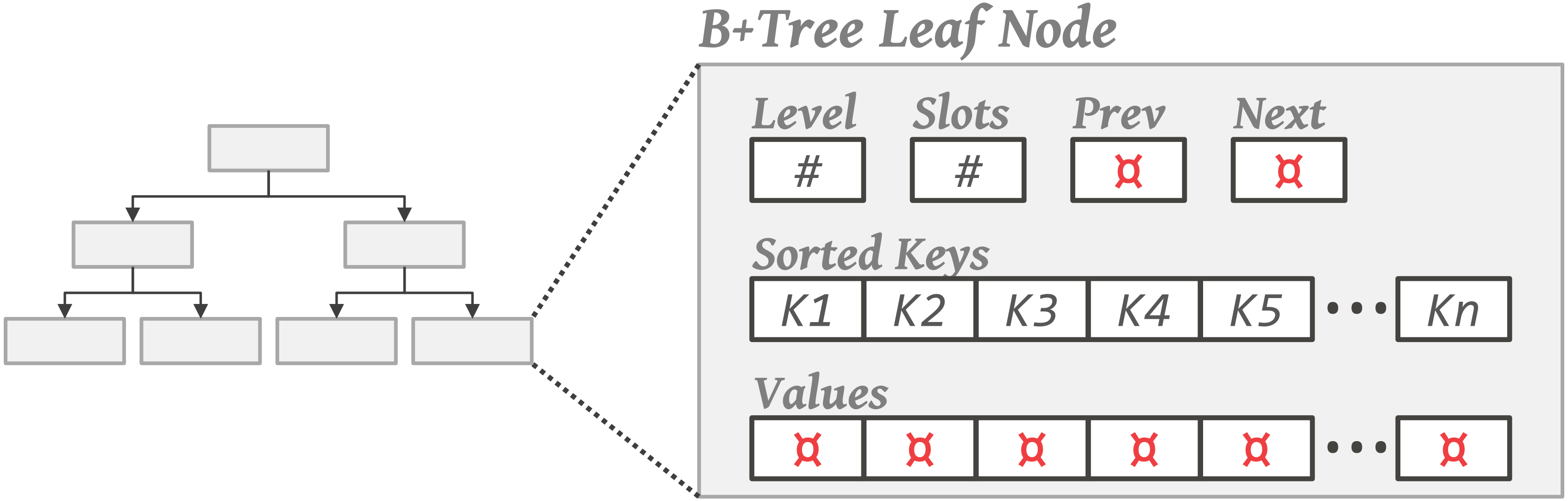
B+TREE



B+TREE



B+TREE



B+Tree Leaf Node

Level | Slots | Prev | Next

# | # | ¤ | ¤

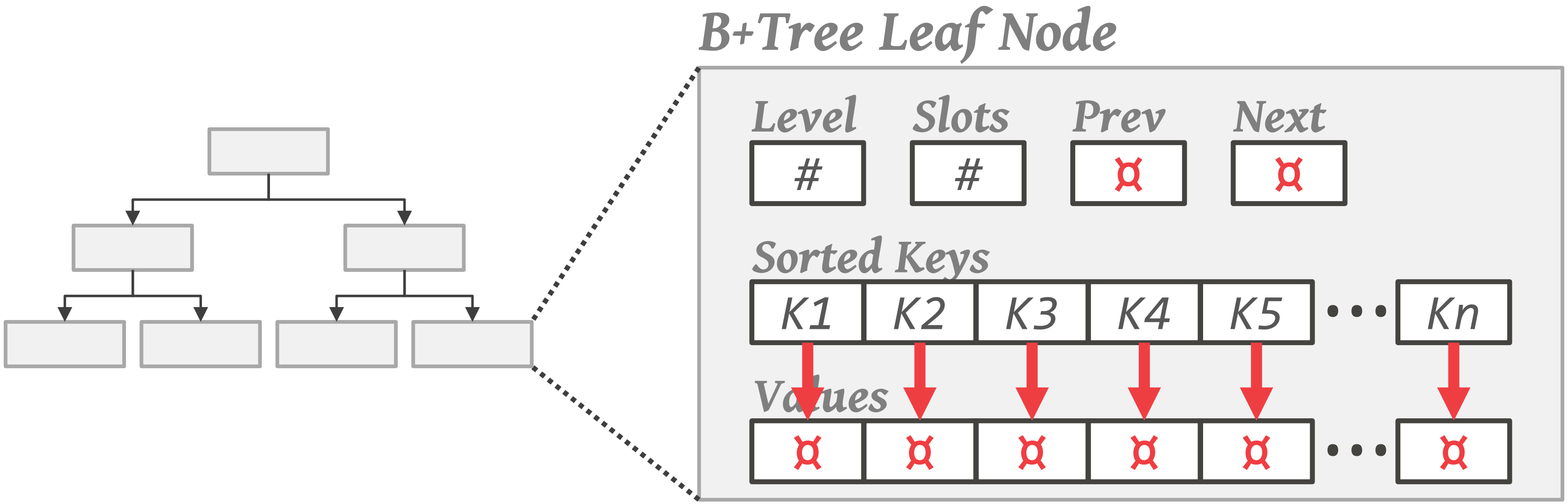
Sorted Keys

K1 | K2 | K3 | K4 | K5 | • • • | Kn

Values

¤ | ¤ | ¤ | ¤ | ¤ | • • • | ¤

B+TREE


B+Tree Leaf Node
Level
#
Slots
#
Prev
¤
Next
¤
Sorted Keys
K1
K2
K3
K4
K5
• • •
Kn
Values
¤
¤
¤
¤
¤
• • •
¤

B+TREE DESIGN CHOICES

Non-Unique Indexes: One key maps to multiple values.

Variable Length Keys: The size of each key is not the same.

B+TREE: NON-UNIQUE INDEXES

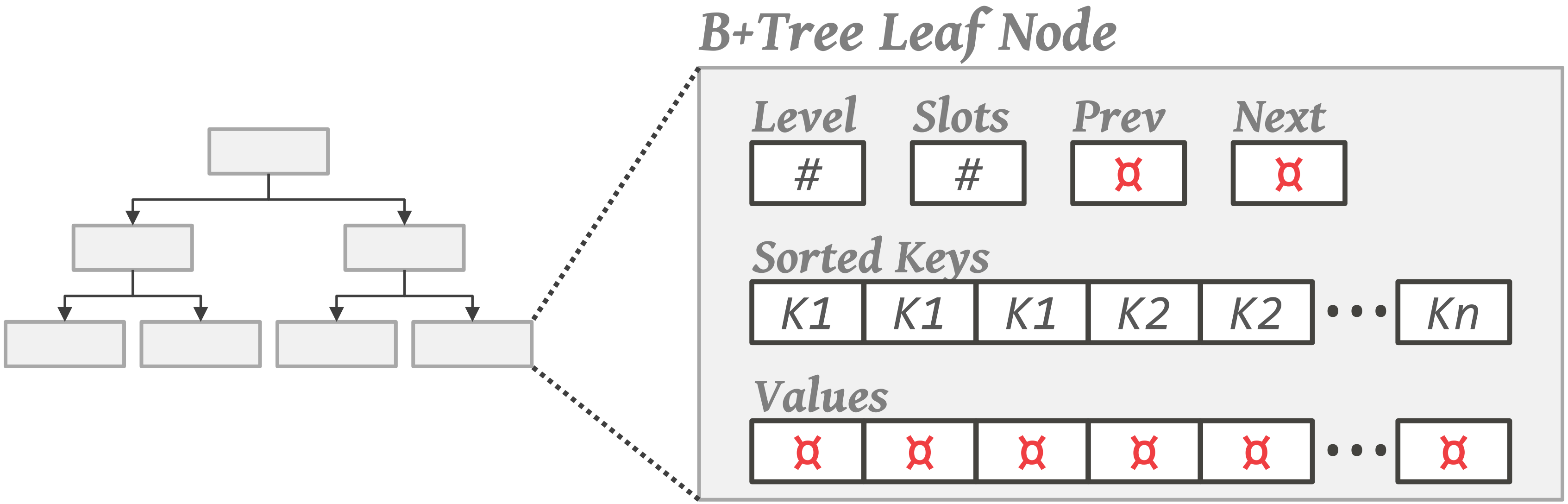
Approach #1: Duplicate Keys

→ Use the same leaf node layout but store duplicate keys multiple times.

Approach #2: Value Lists

→ Store each key only once and maintain a linked list of unique values.

B+TREE: DUPLICATE KEYS



B+Tree Leaf Node

Level | *Slots* | *Prev* | *Next*

# | # | ¤ | ¤

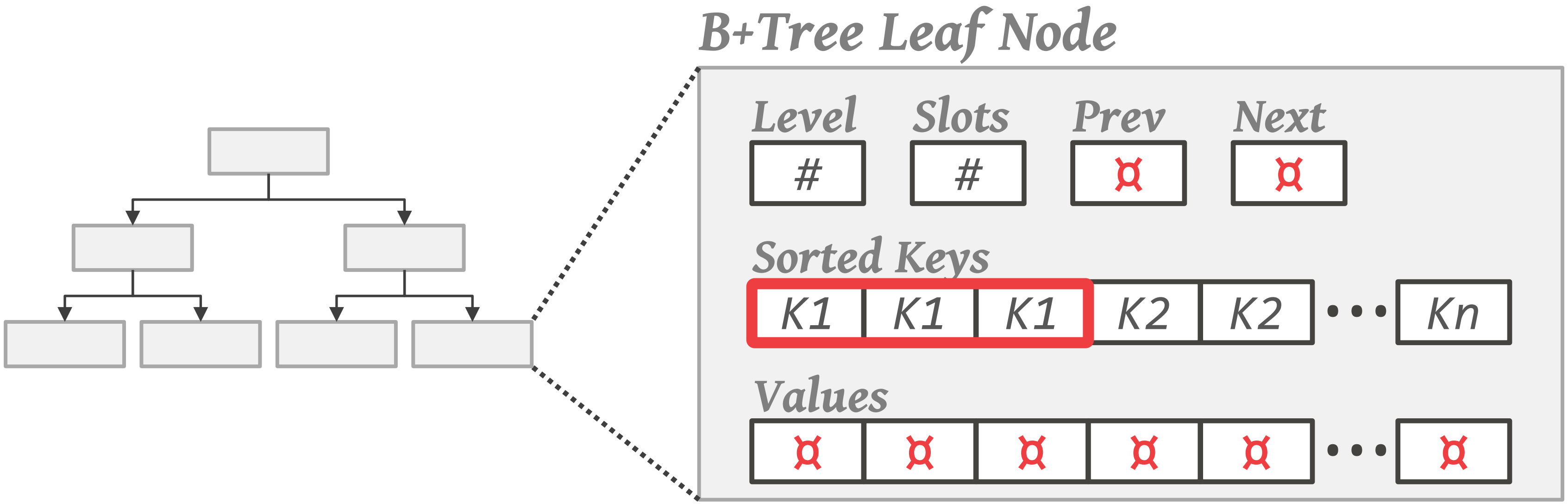
Sorted Keys

K1 | *K1* | *K1* | *K2* | *K2* | • • • | *Kn*

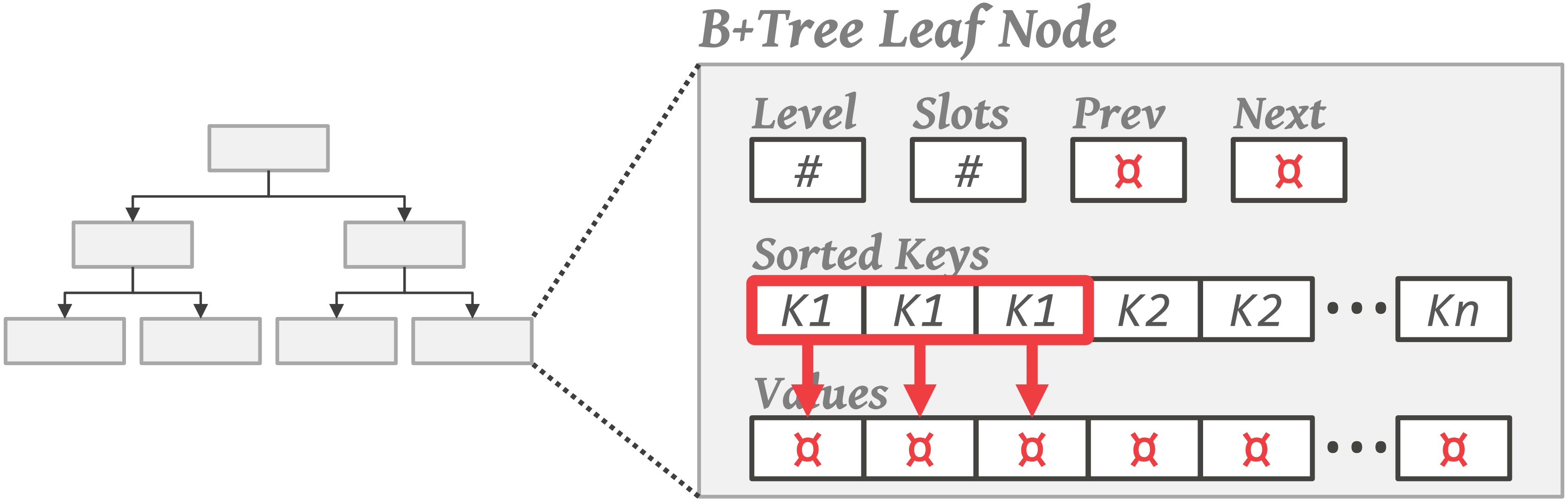
Values

¤ | ¤ | ¤ | ¤ | ¤ | • • • | ¤

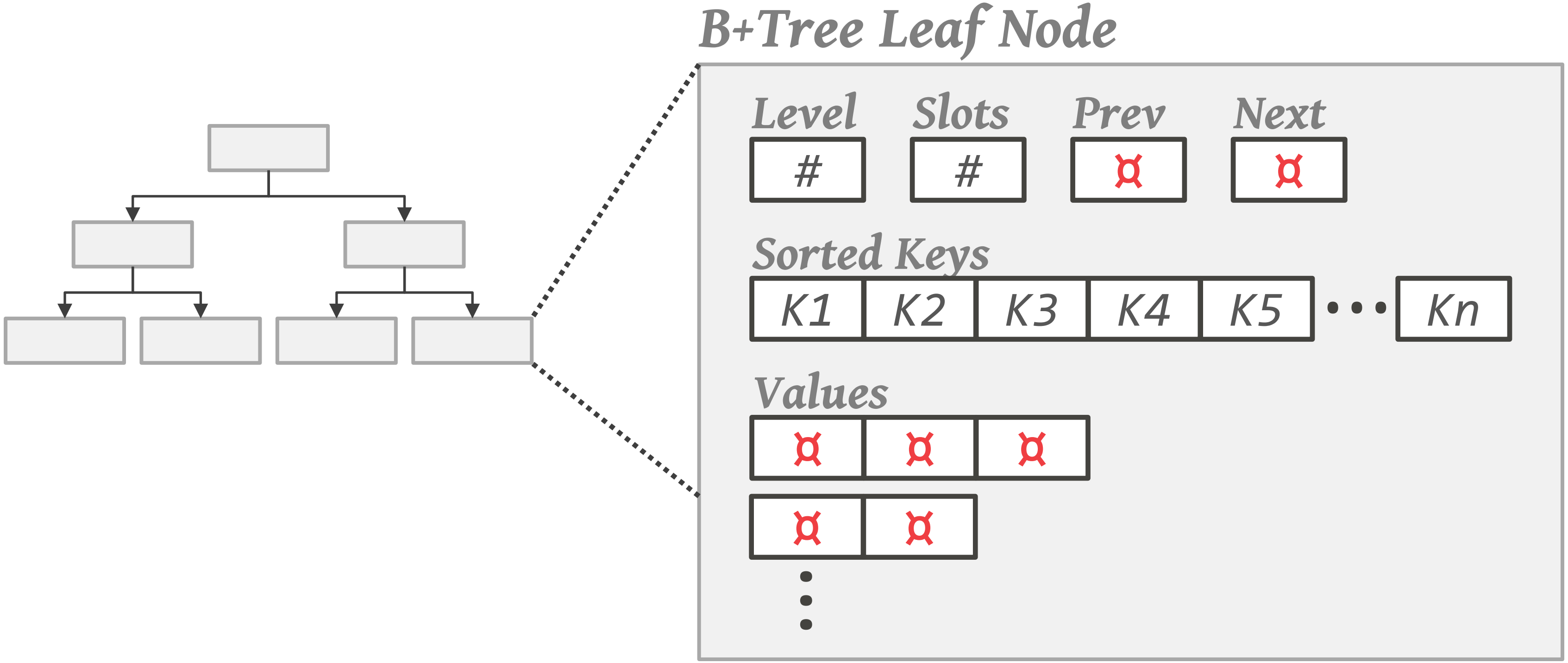
B+TREE: DUPLICATE KEYS



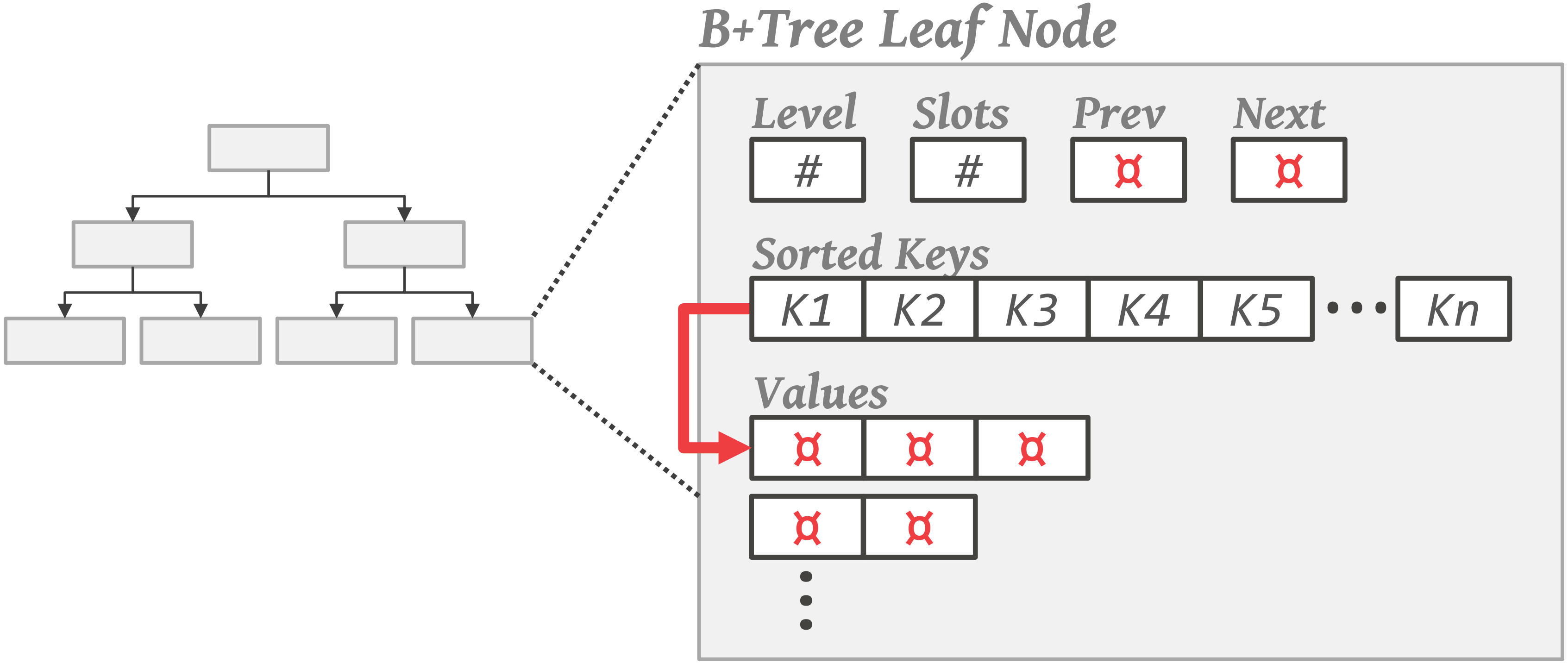
B+TREE: DUPLICATE KEYS



B+TREE: VALUE LISTS



B+TREE: VALUE LISTS



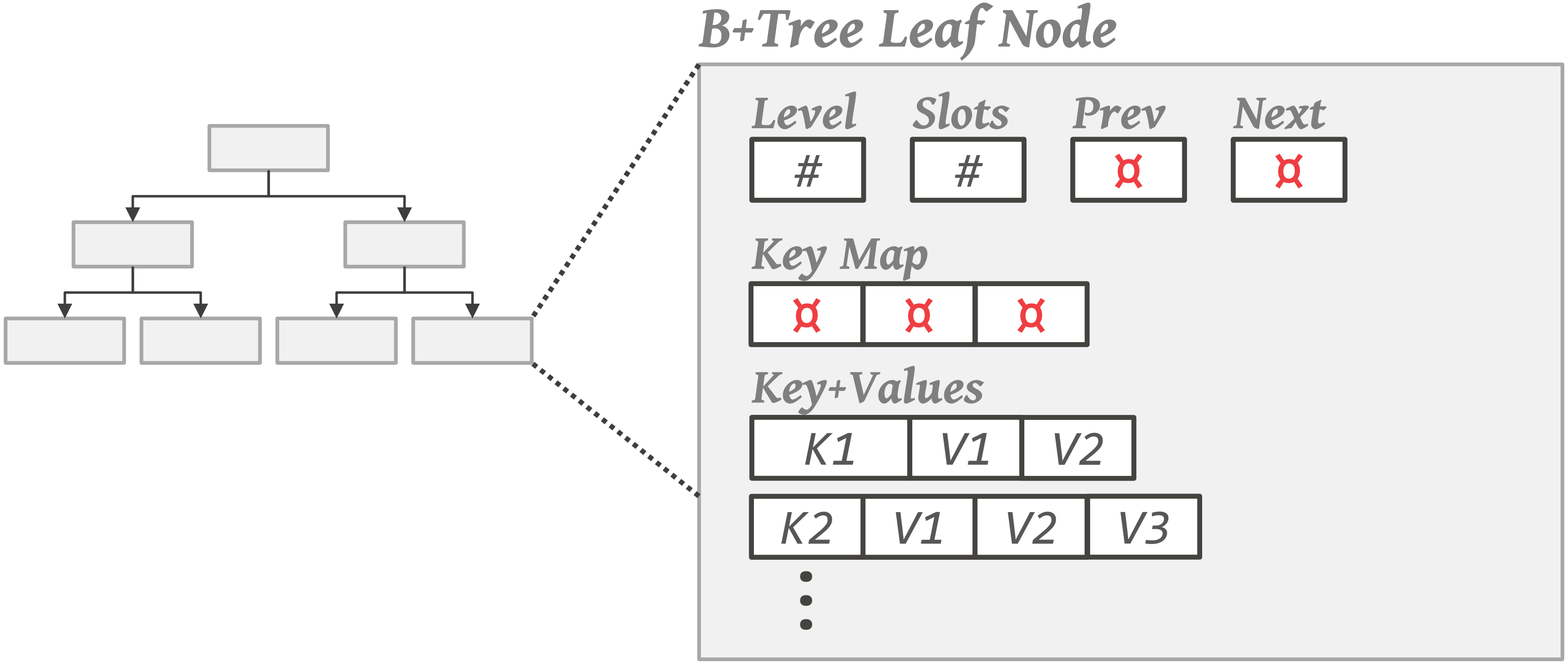
B+TREE: VARIABLE LENGTH KEYS

Approach #1: Pointers
→ Store the keys as pointers to the tuple's attribute.

Approach #2: Variable Length Nodes
→ The size of each node in the b+tree can vary.
→ Requires careful memory management.

Approach #3: Key Map
→ Embed an array of pointers that map to the key + value list within the node.

B+TREE: KEY MAP



B+Tree Leaf Node

Level | Slots | Prev | Next

# | # | ¤ | ¤

Key Map

¤ ¤ ¤

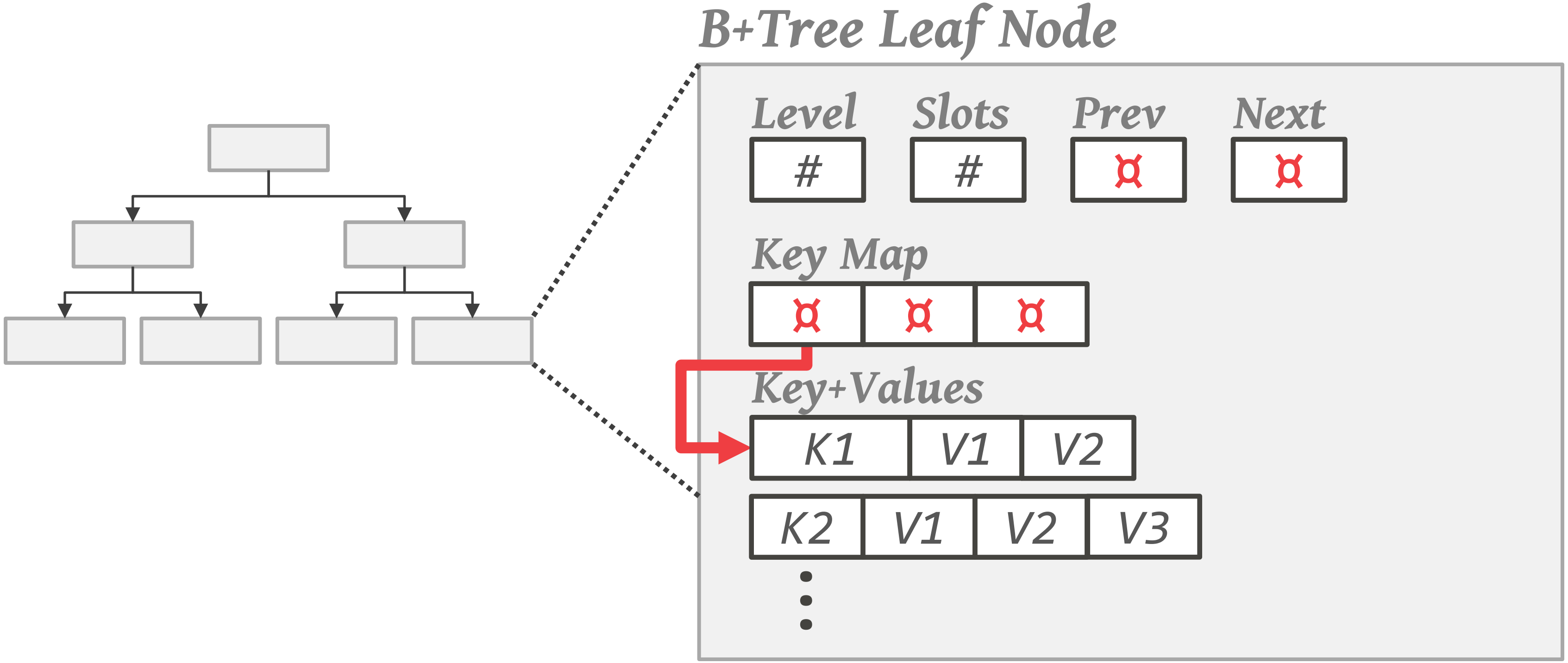
Key+Values

K1 V1 V2

K2 V1 V2 V3

⋮

B+TREE: KEY MAP



B+TREE ALTERNATIVES

T-Trees

Skip Lists

Radix Trees (aka Patricia Trees)

MassTree

Fractal Trees

B+TREE ALTERNATIVES

T-Trees

Skip Lists

Radix Trees (aka Patricia Trees)

MassTree

Fractal Trees

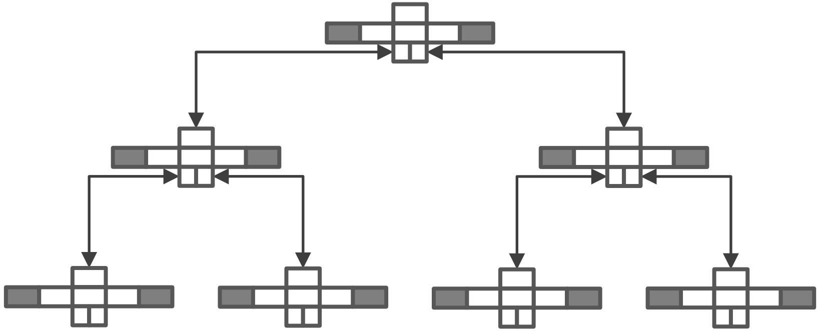
T-TREES

Based on AVL Trees. Instead of storing keys in nodes, store pointers to their original values.

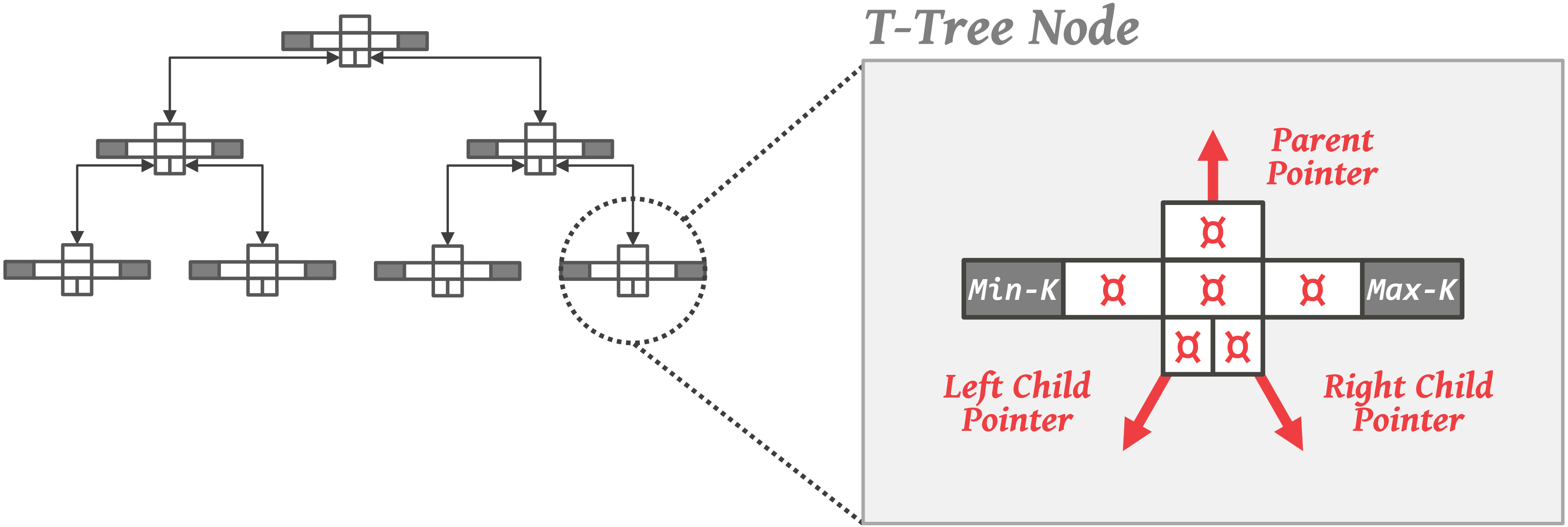
Proposed in 1986 from Univ. of Wisconsin Used in TimesTen and other early in-memory DBMSs during the 1990s.

A STUDY OF INDEX STRUCTURES FOR MAIN MEMORY DATABASE MANAGEMENT SYSTEMS
VLDB 1986

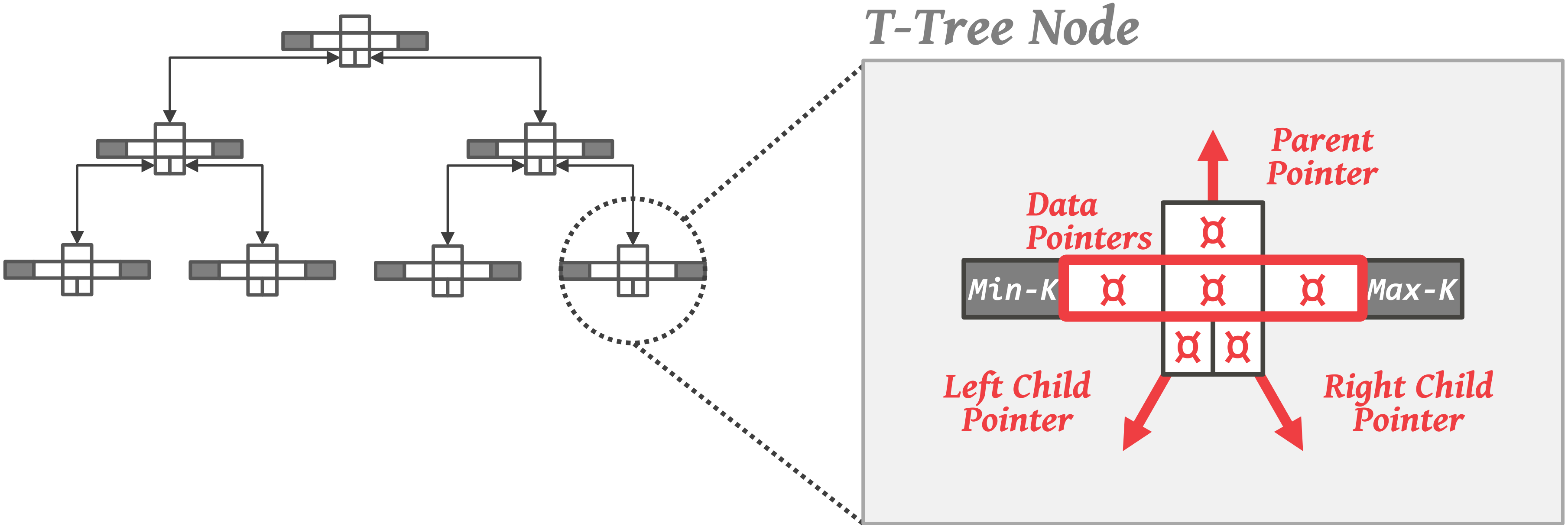
T-TREES



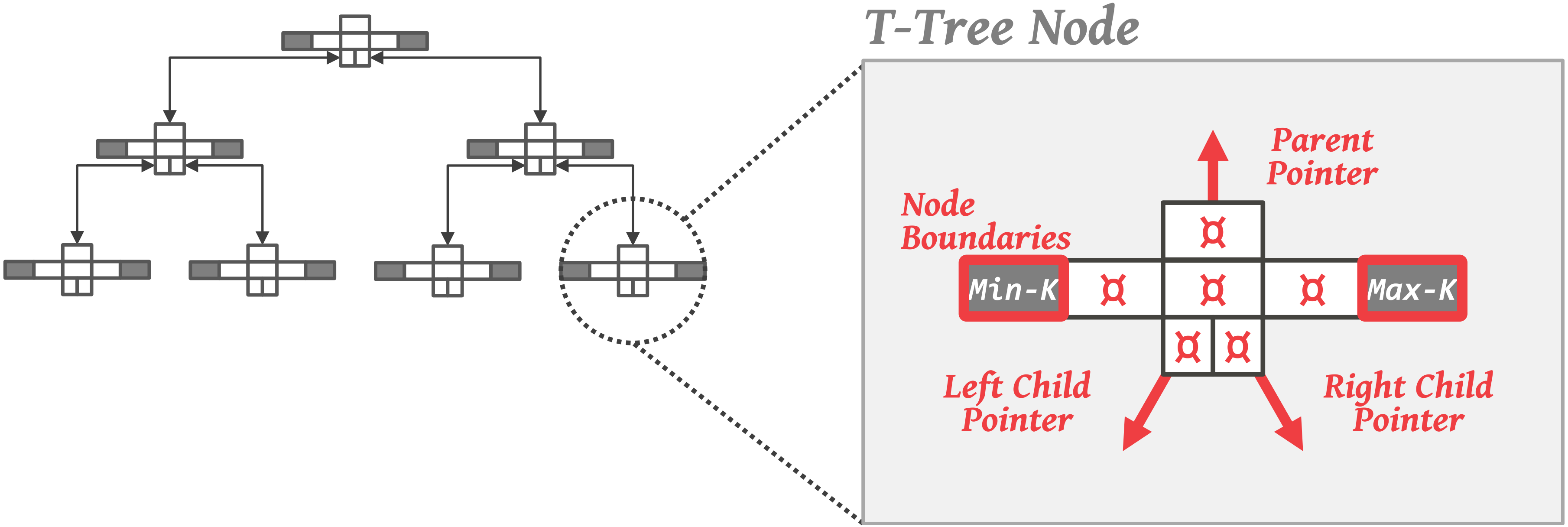
T-TREES


T-Tree Node
Parent Pointer
Min-K
Max-K
¤
Left Child Pointer
Right Child Pointer

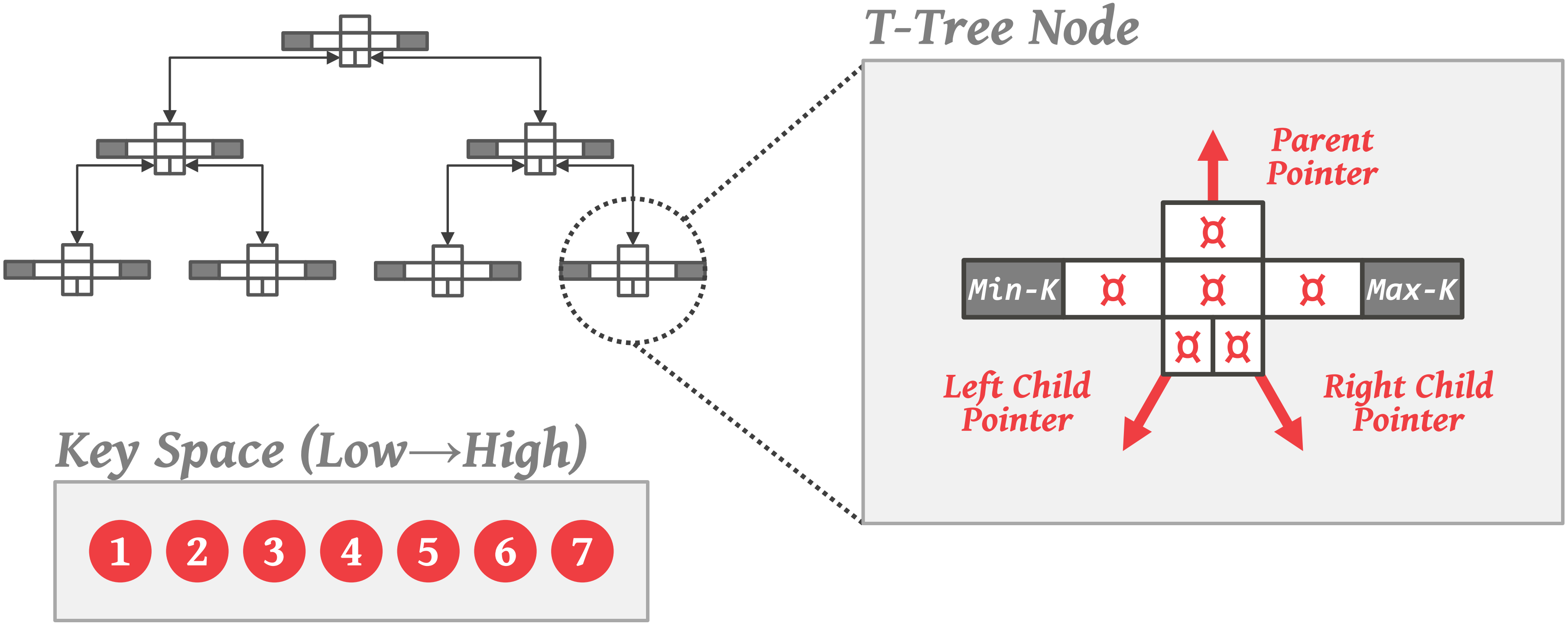
T-TREES


T-Tree Node
Parent Pointer
Data Pointers
Min-K
Max-K
Left Child Pointer
Right Child Pointer

T-TREES


T-Tree Node
Parent Pointer
Node Boundaries
Min-K
¤
¤
¤
¤
Max-K
¤
¤
Left Child Pointer
Right Child Pointer

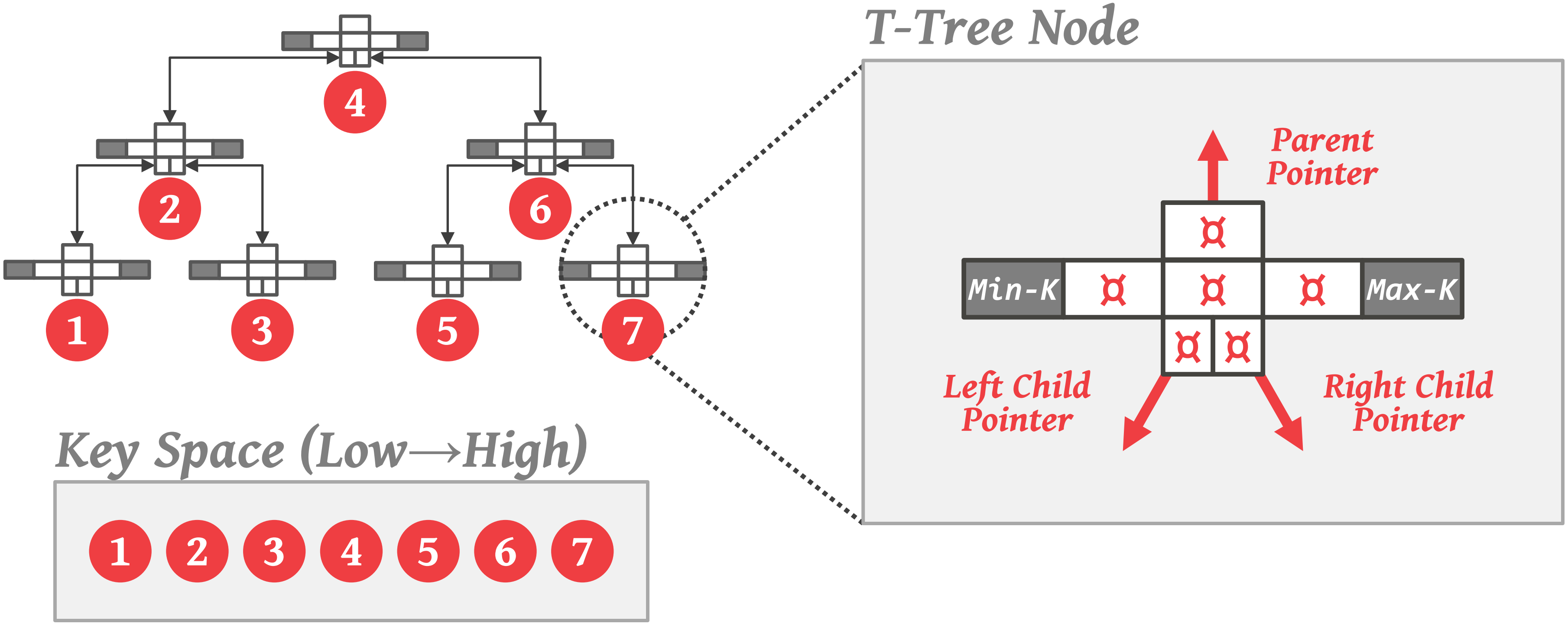
T-TREES



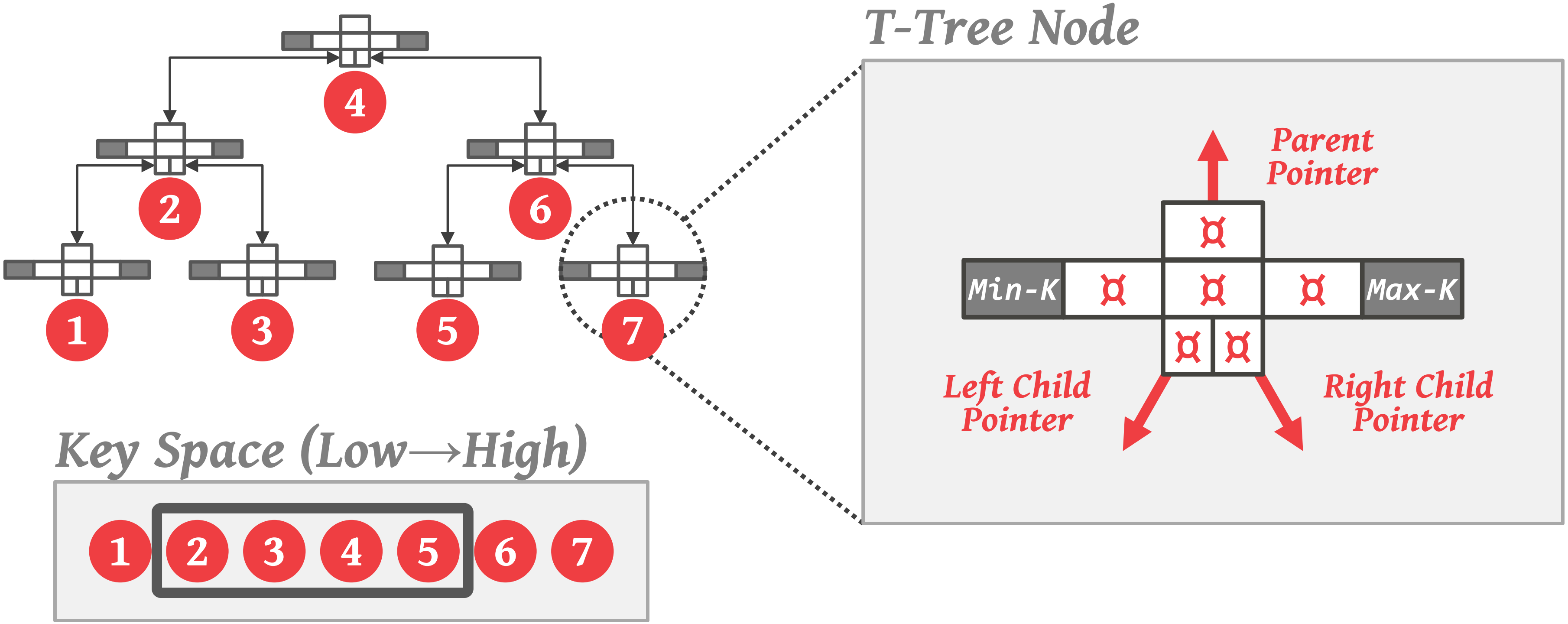
T-Tree Node

Key Space (Low→High)

T-TREES


4
2
6
1
3
5
7
Key Space (Low→High)
1
2
3
4
5
6
7
T-Tree Node
Parent Pointer
¤
Min-K
¤
¤
¤
Max-K
¤
¤
Left Child Pointer
Right Child Pointer

T-TREES


4
2
6
1
3
5
7
Key Space (Low→High)
1
2
3
4
5
6
7
T-Tree Node
Parent Pointer
¤
Min-K
¤
¤
¤
Max-K
¤
¤
Left Child Pointer
Right Child Pointer

T-TREES

Advantages

→ Uses less memory because it does not store keys inside of each node.

Disadvantages

→ Have to chase pointers when scanning range or performing binary search inside of a node.
→ Difficult to rebalance.
→ Difficult to implement safe concurrent access.

SKIP LISTS

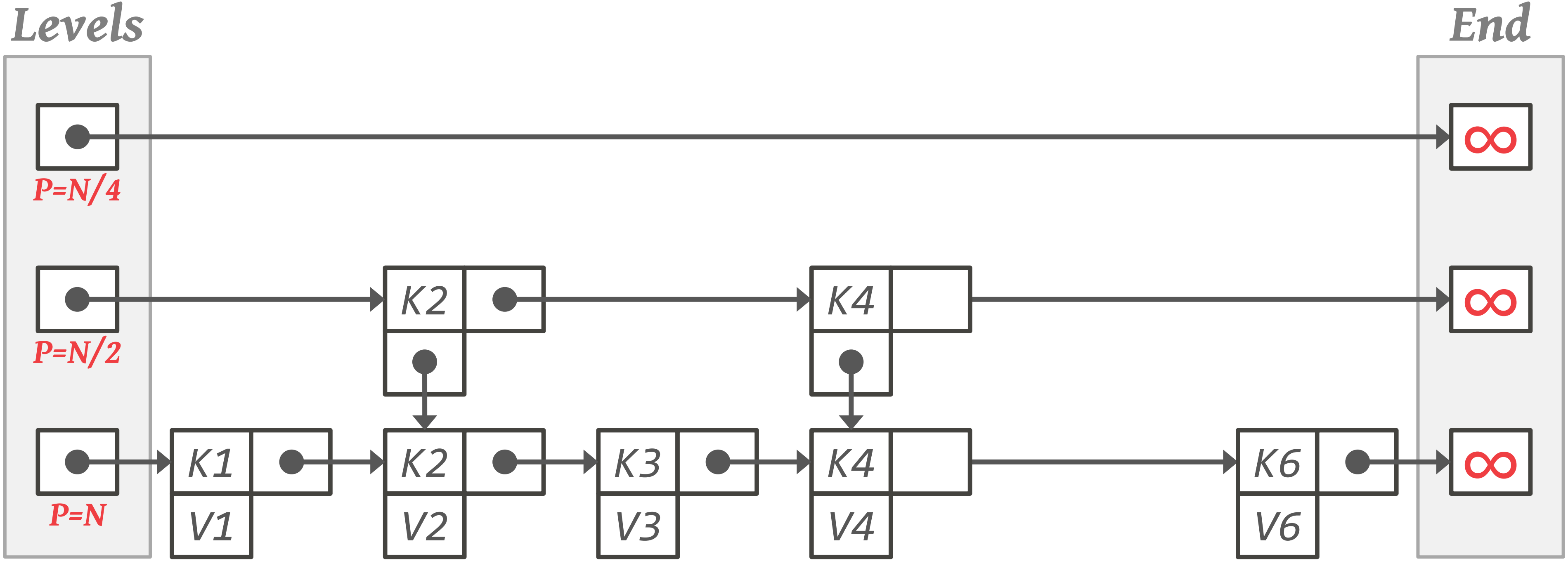
A collection of lists at different levels

→ Lowest level is a sorted, singly linked list of all keys
→ 2nd level links every other key
→ 3rd level links every fourth key
→ In general, a level has half the keys of one below it

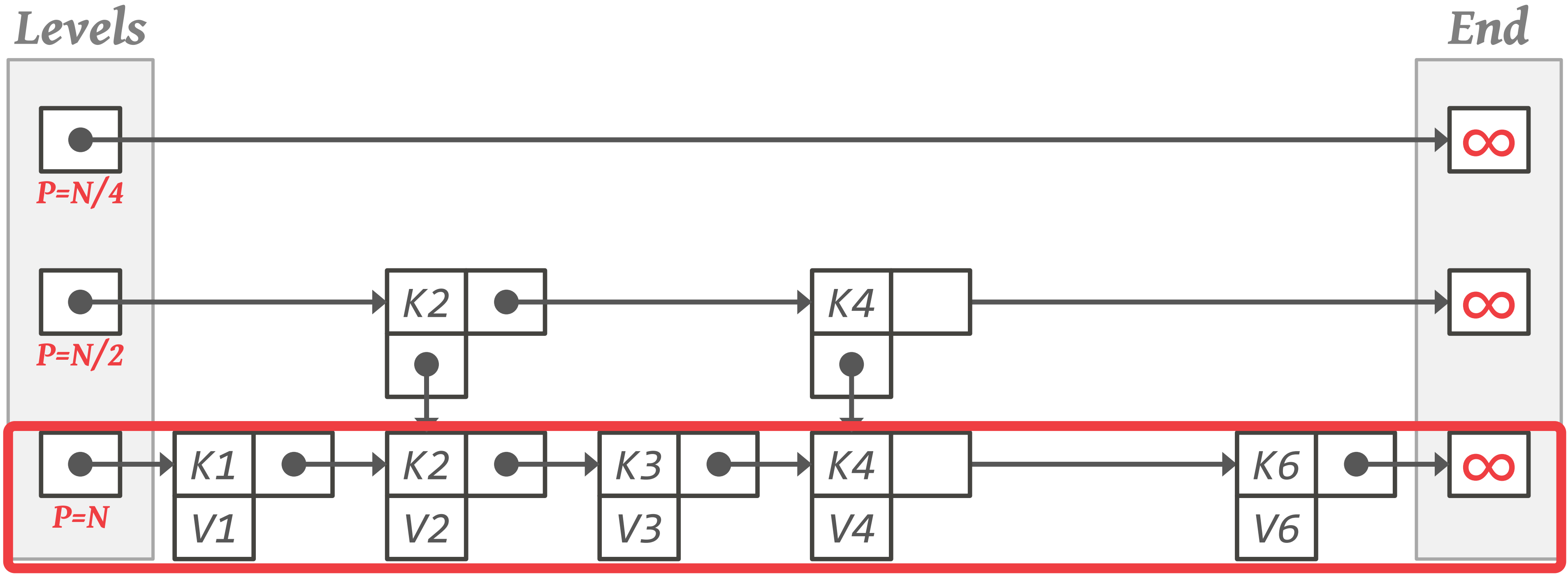
To insert a new key, flip a coin to decide how many levels to add the new key into. Provides approximate O(log n) search times.

CONCURRENT MAINTENANCE OF SKIP LISTS
Univ. of Maryland Tech Report 1990

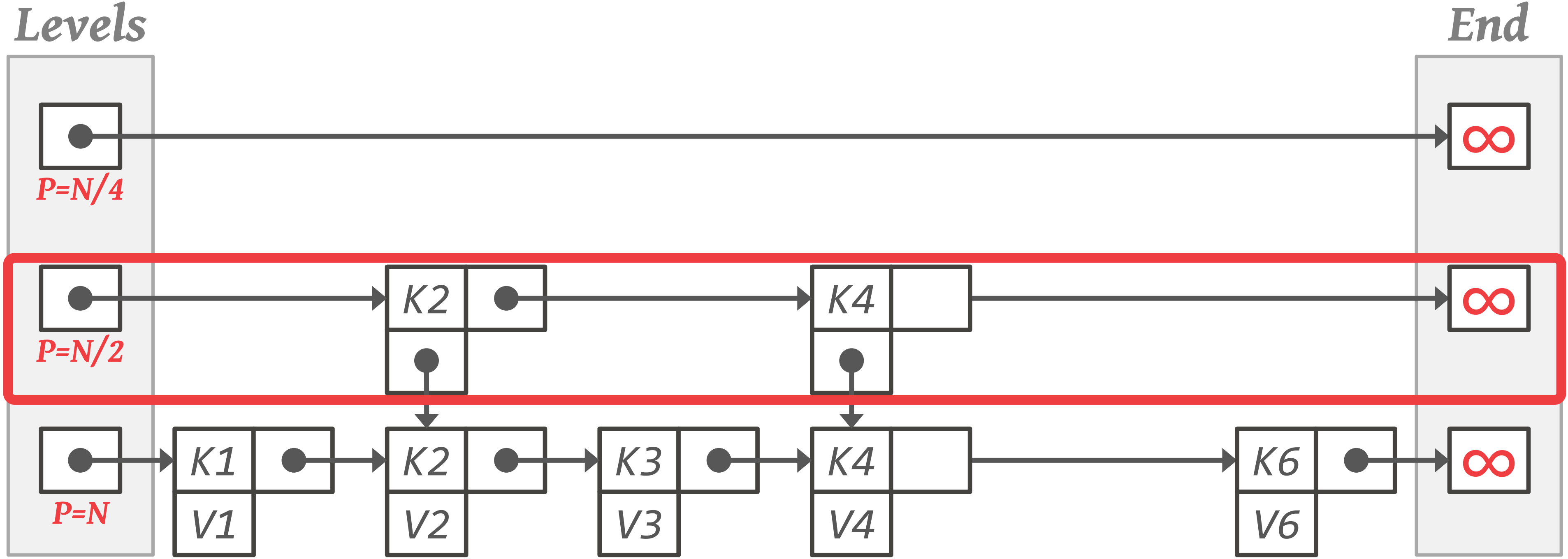
SKIP LISTS: INSERT



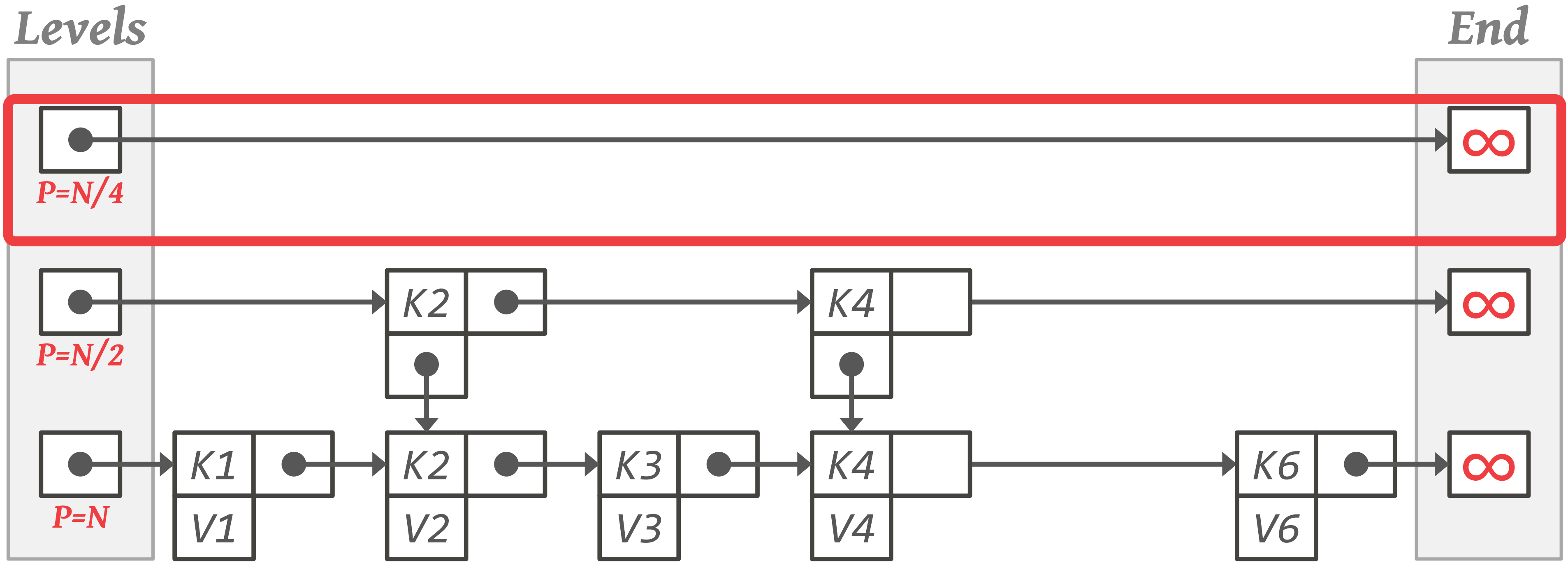
SKIP LISTS: INSERT



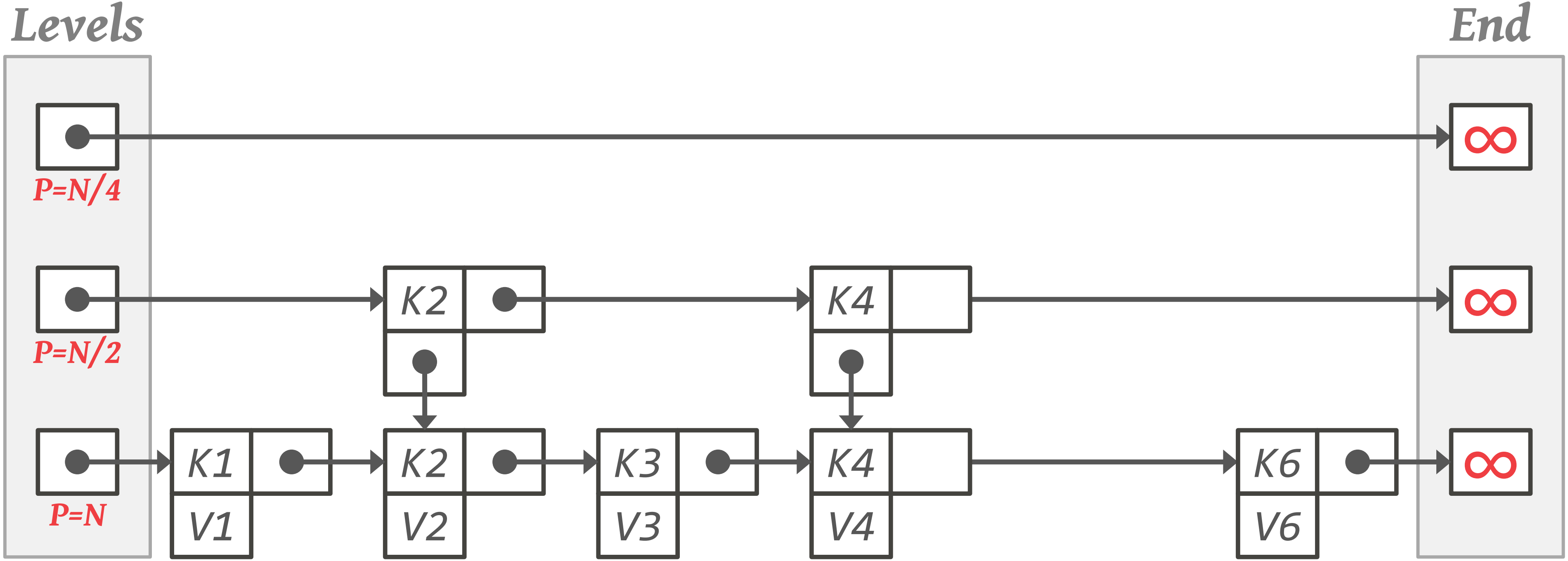
SKIP LISTS: INSERT



SKIP LISTS: INSERT

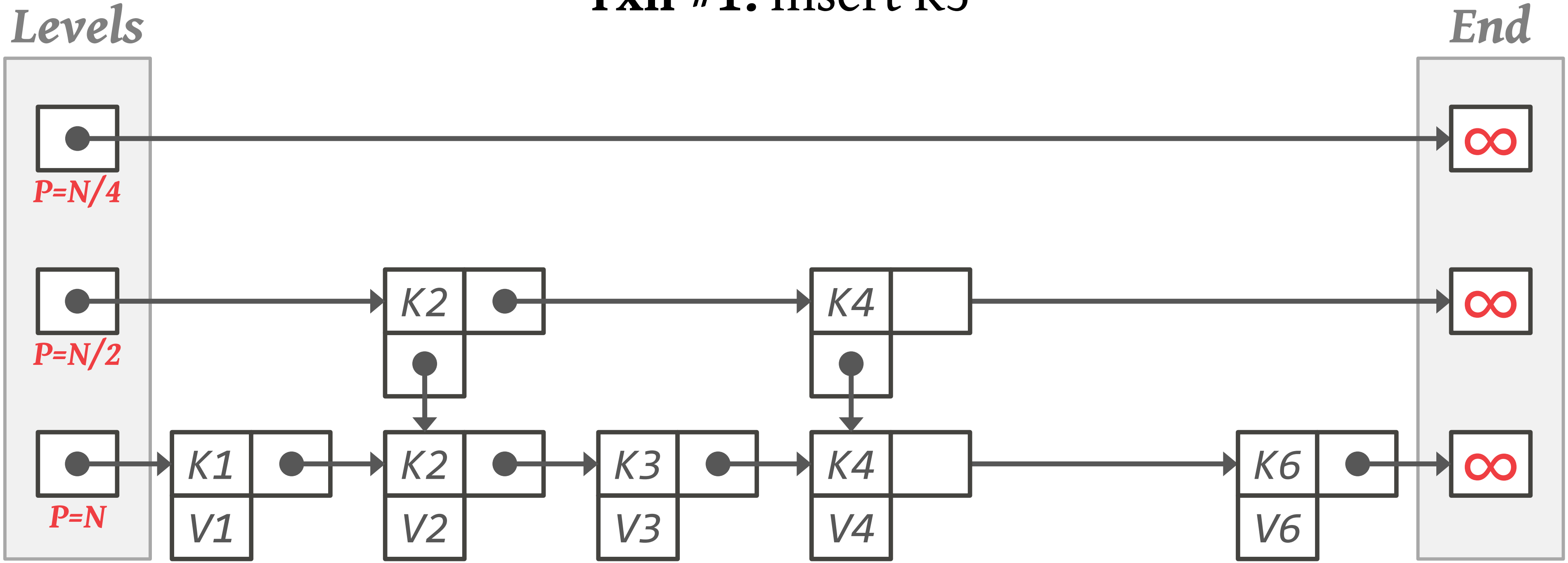


SKIP LISTS: INSERT



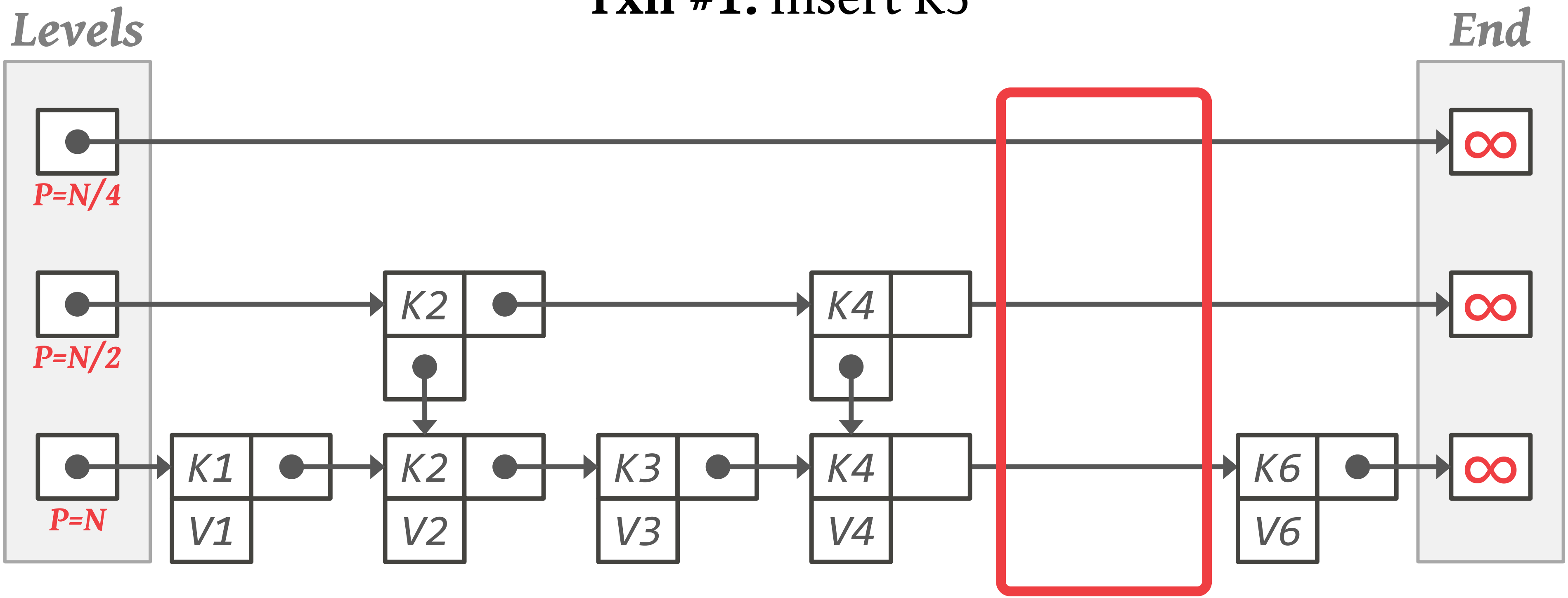
SKIP LISTS: INSERT

Txn #1: Insert K5



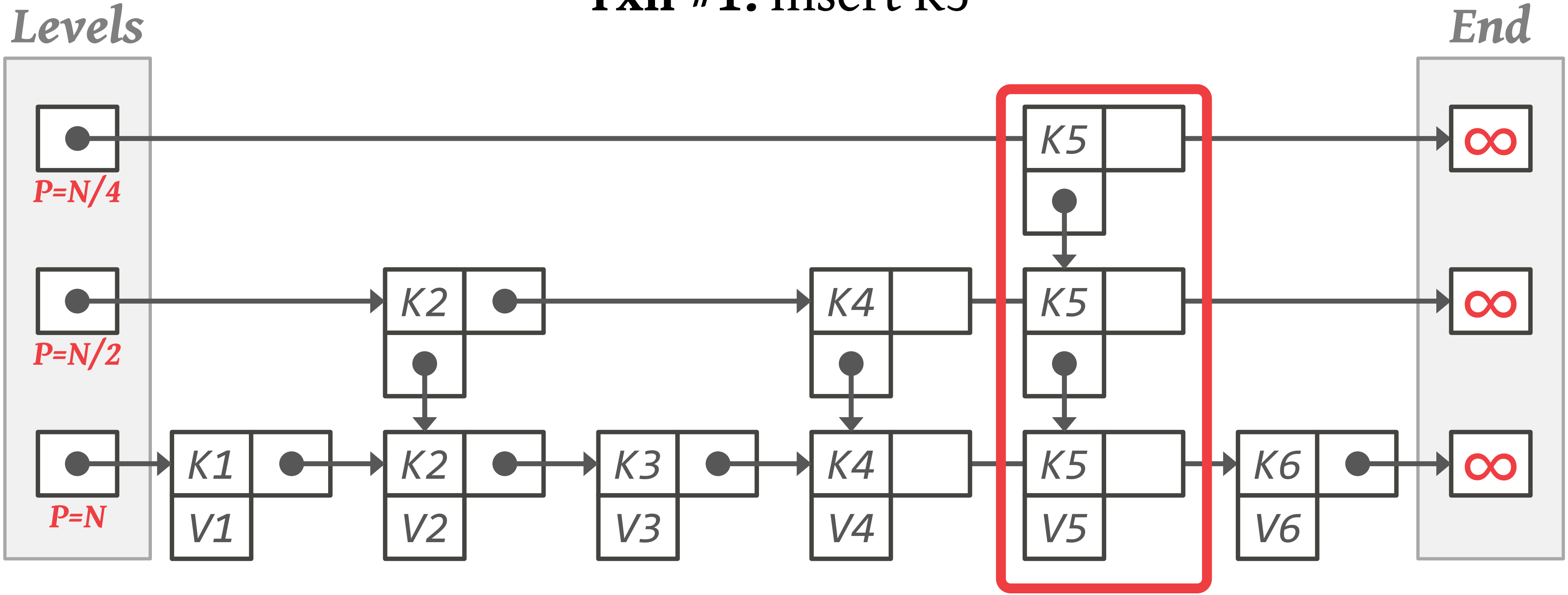
SKIP LISTS: INSERT

Txn #1: Insert K5



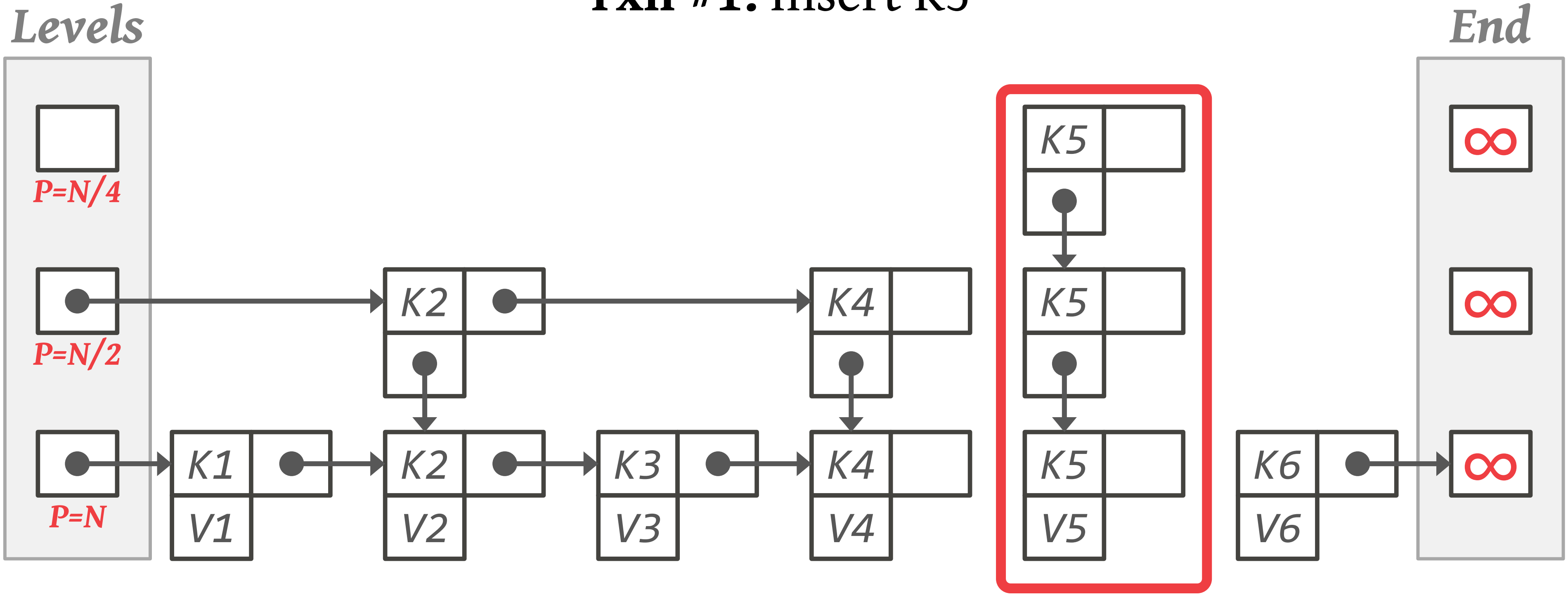
SKIP LISTS: INSERT

Txn #1: Insert K5



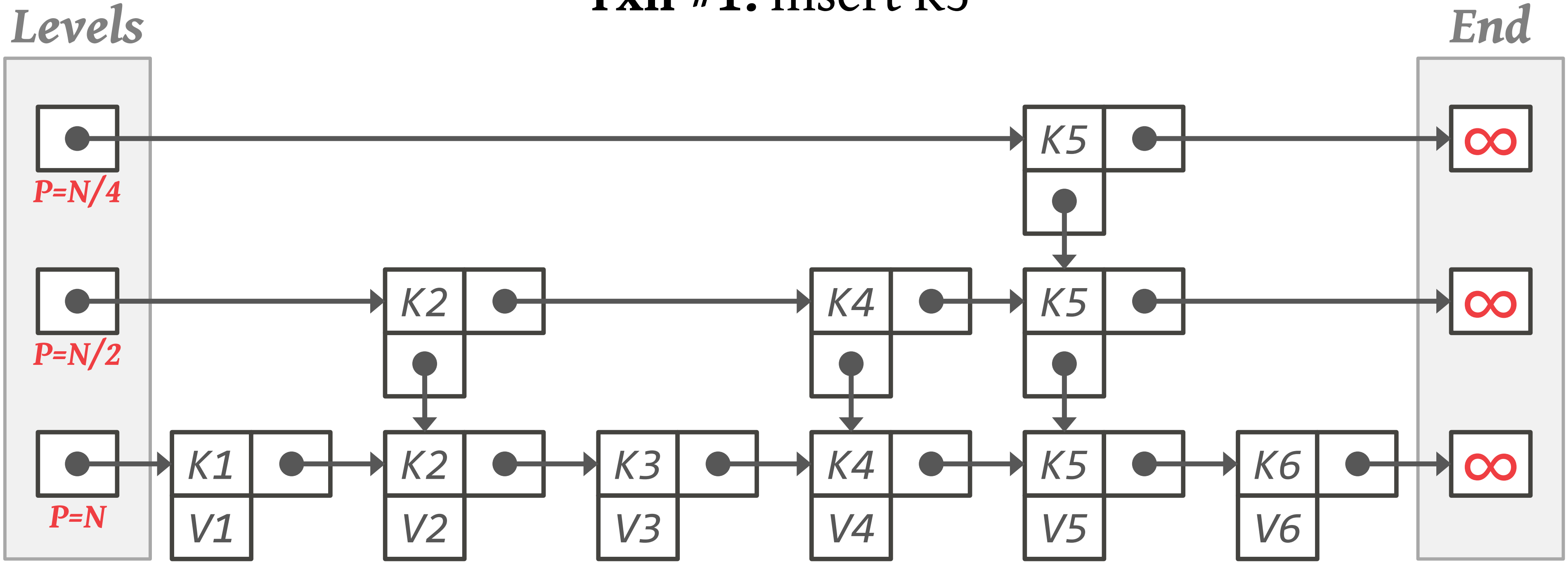
SKIP LISTS: INSERT

Txn #1: Insert K5

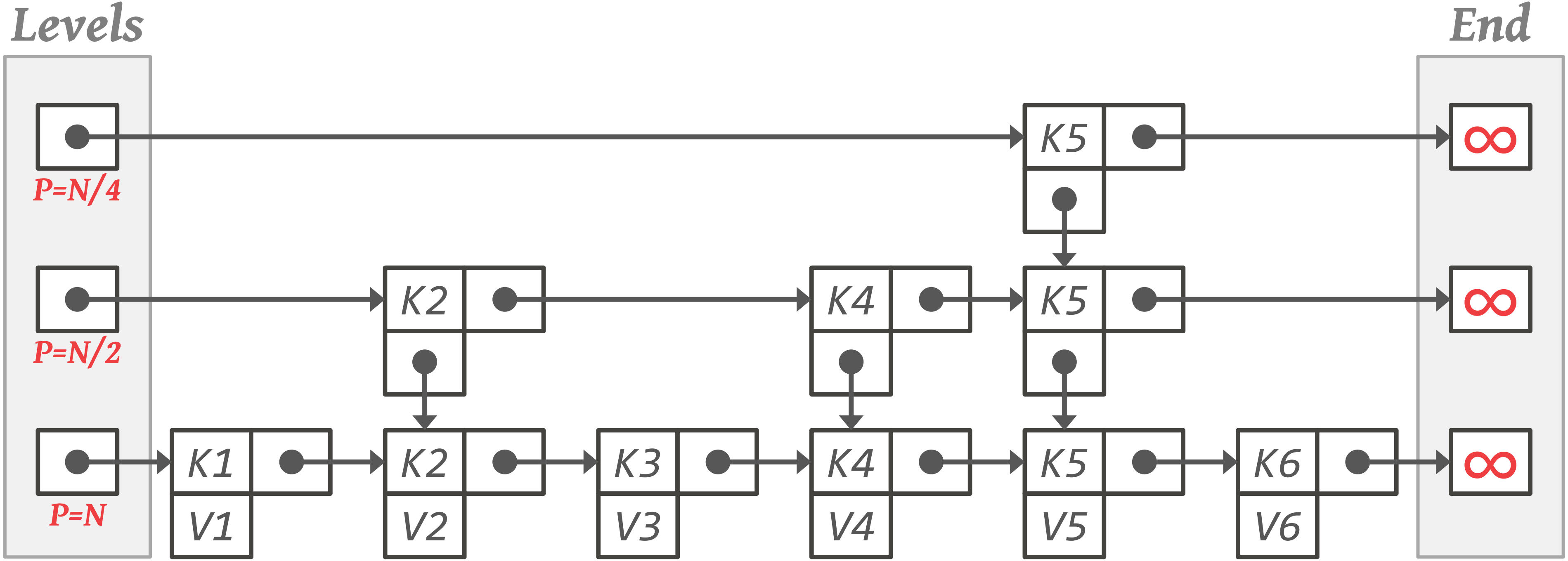


SKIP LISTS: INSERT

Txn #1: Insert K5

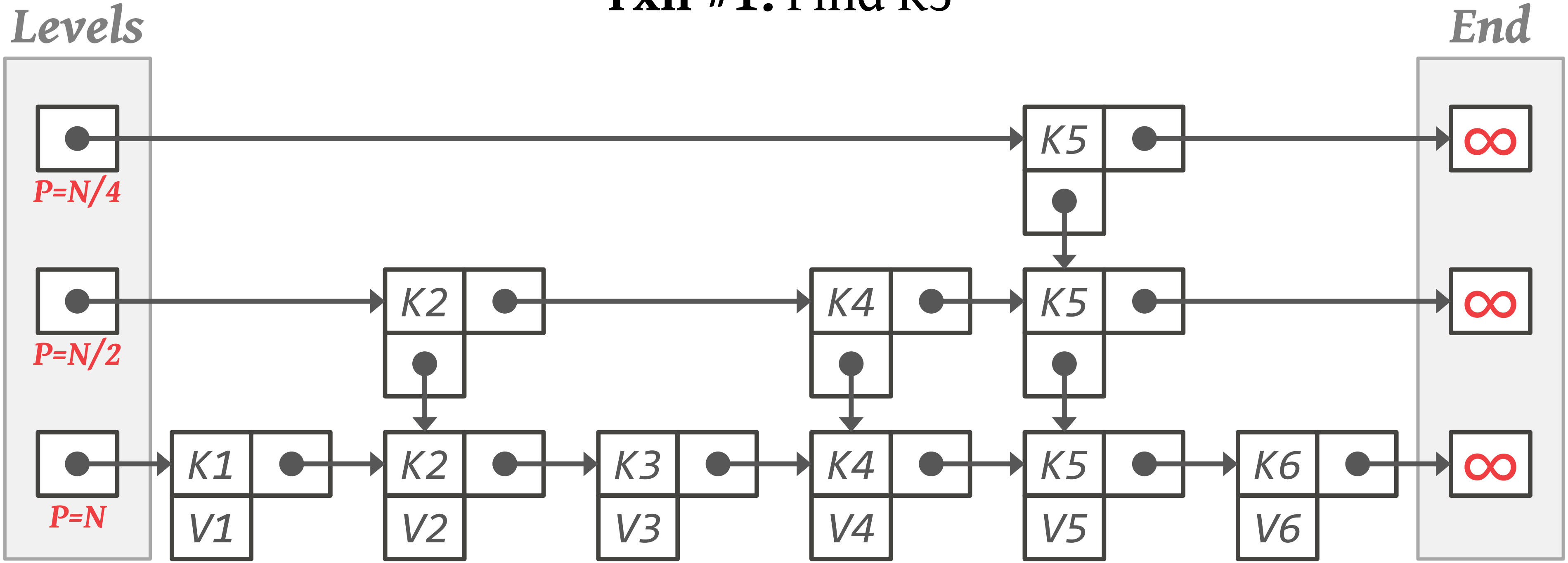


SKIP LISTS: SEARCH



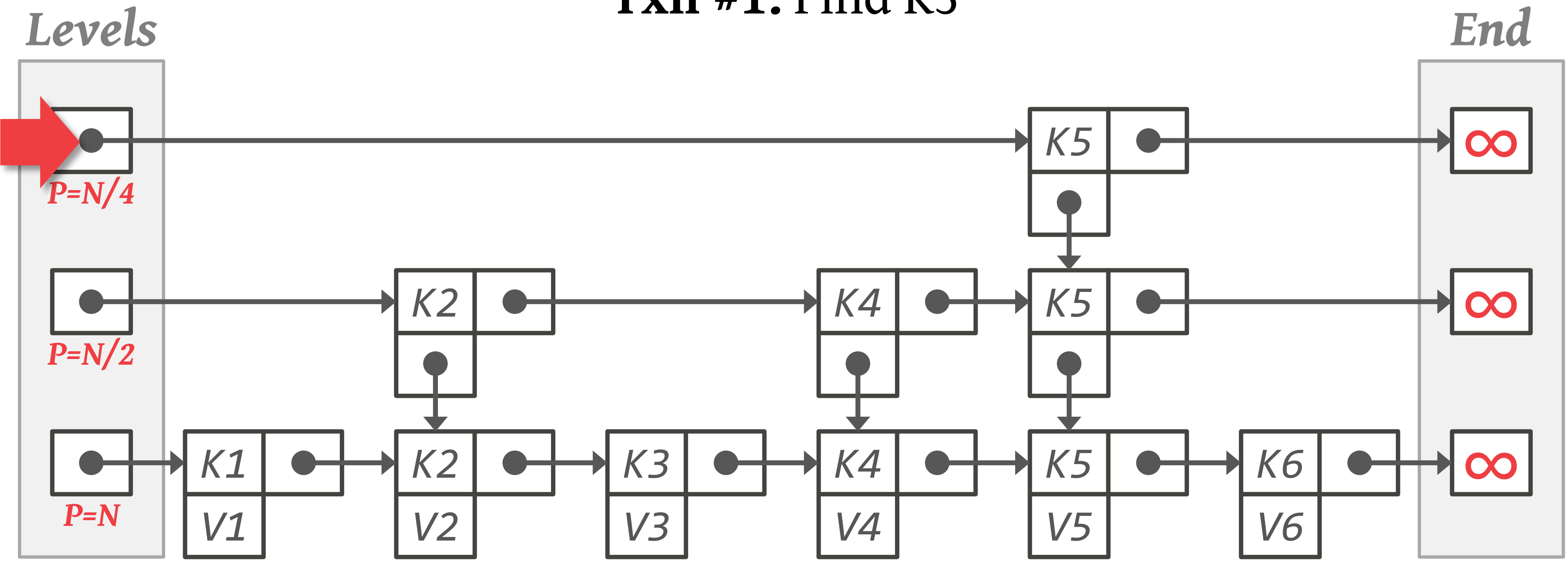
SKIP LISTS: SEARCH

Txn #1: Find K3



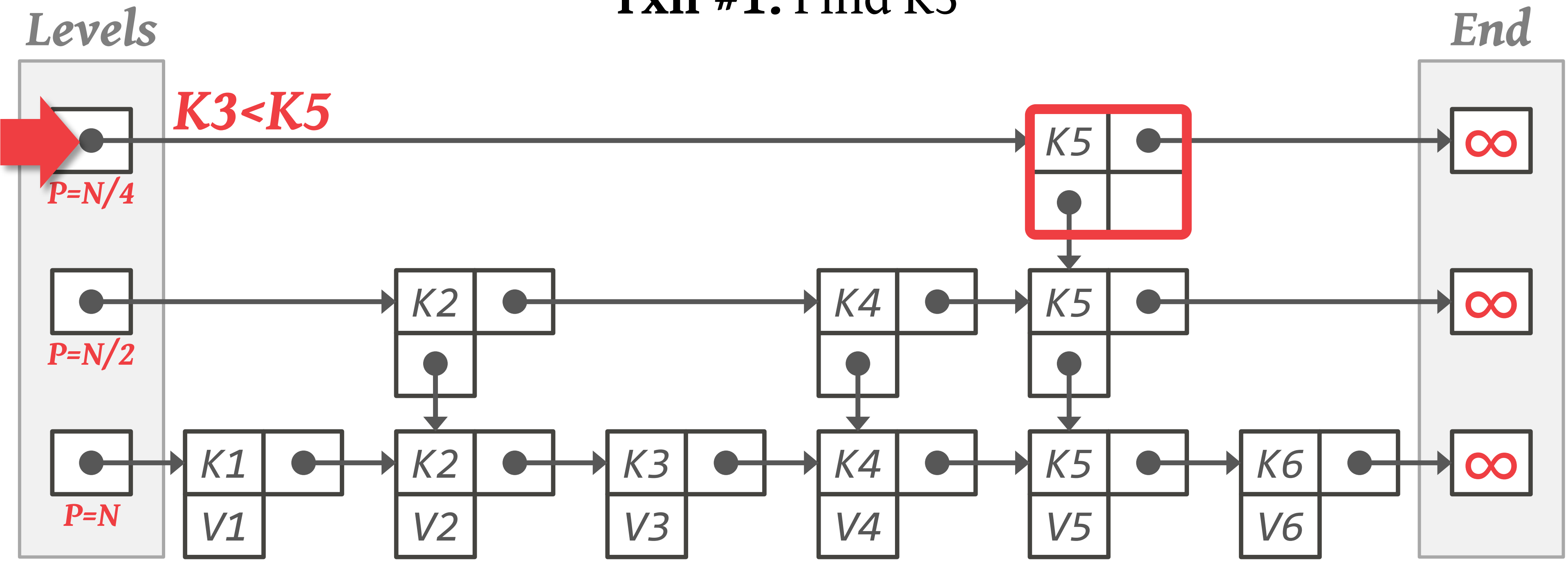
SKIP LISTS: SEARCH

Txn #1: Find K3



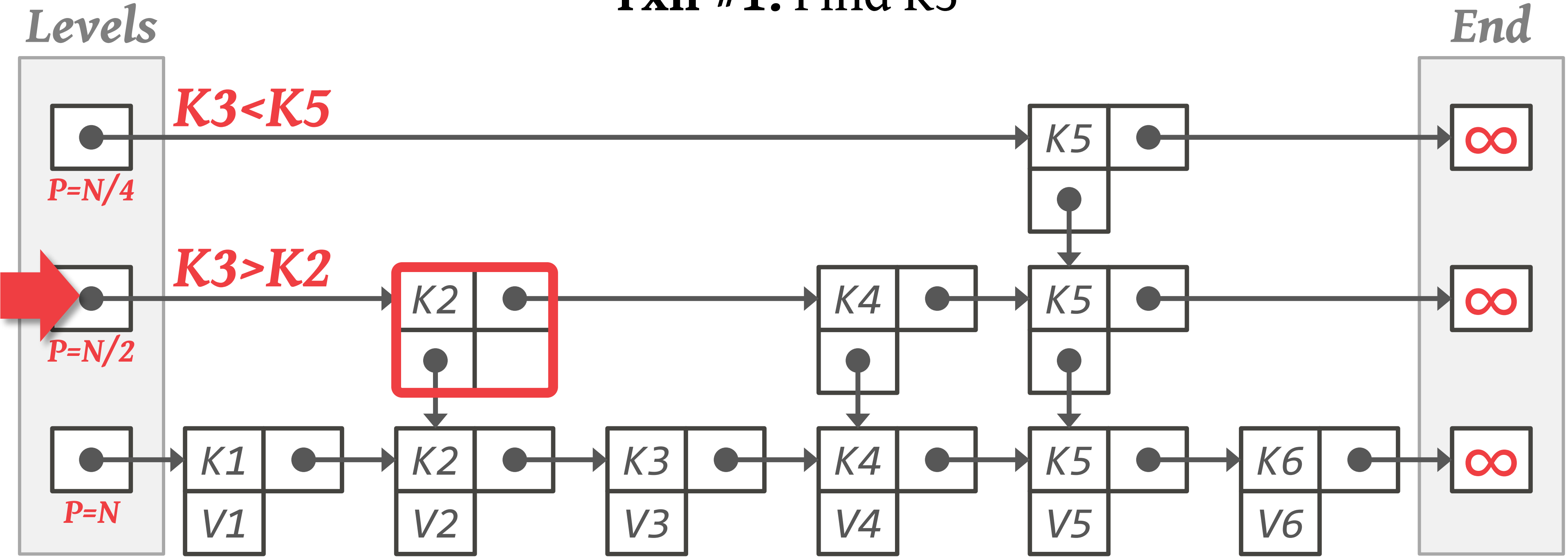
SKIP LISTS: SEARCH

Txn #1: Find K3



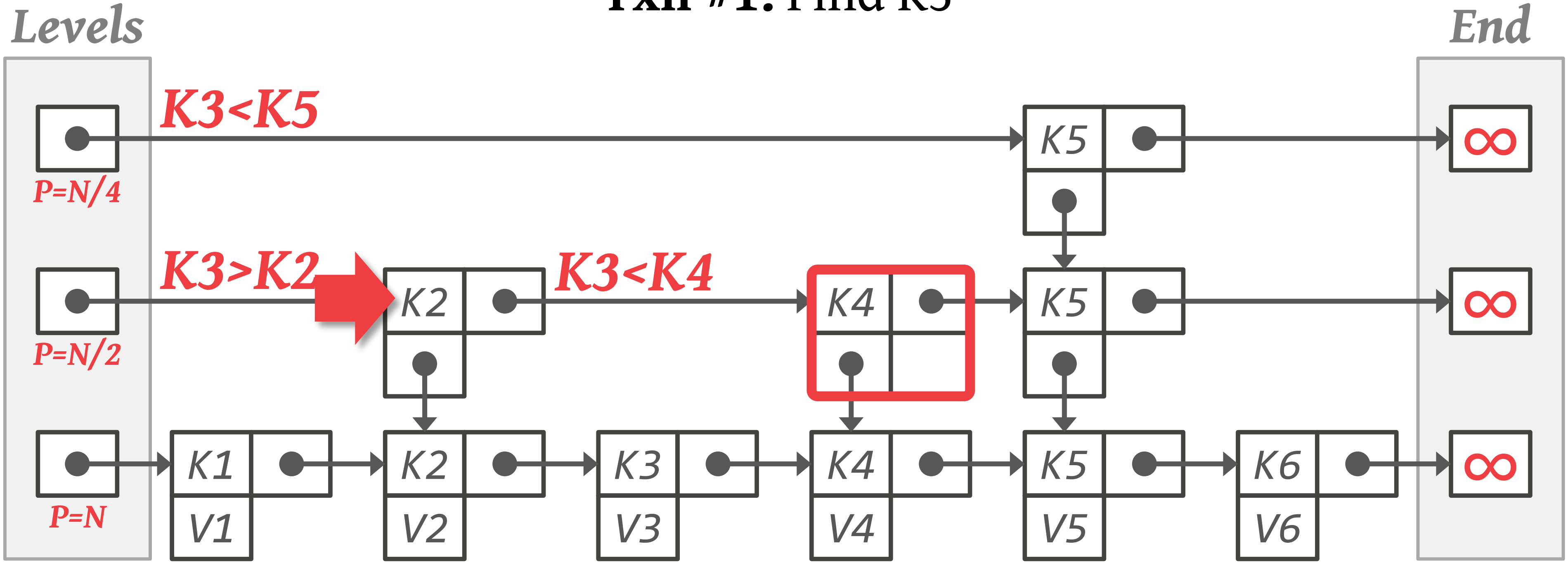
SKIP LISTS: SEARCH

Txn #1: Find K3



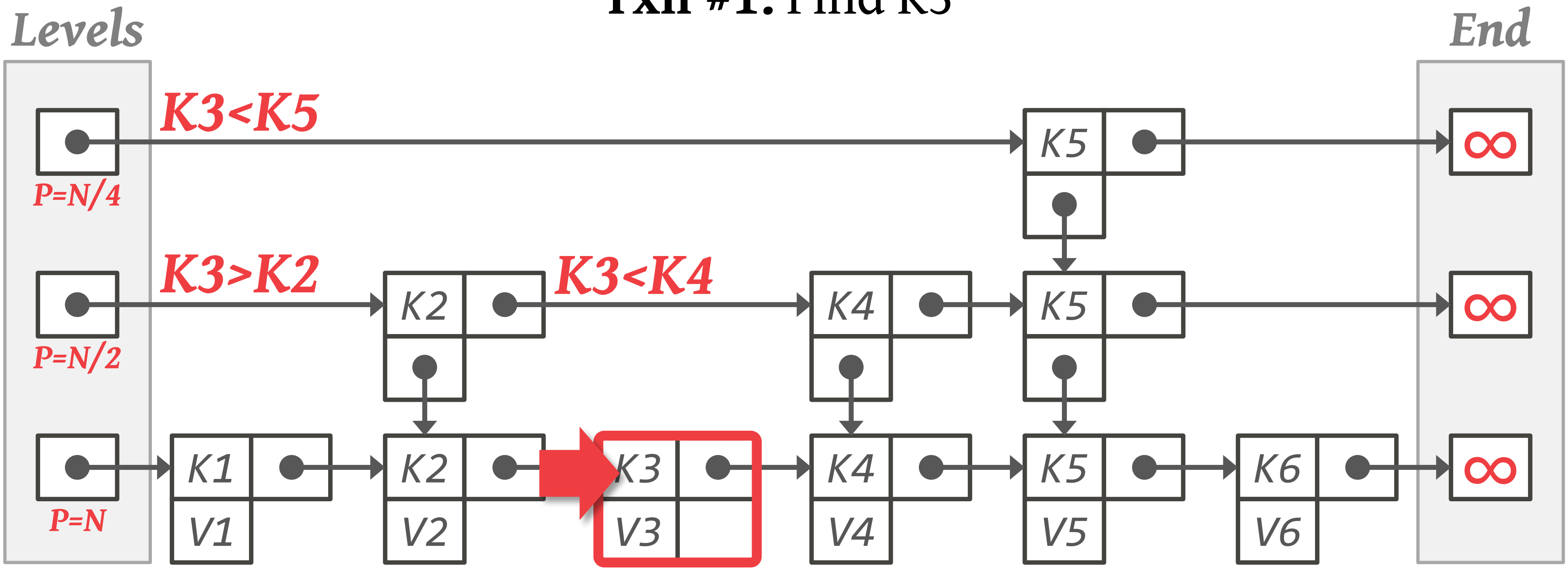
SKIP LISTS: SEARCH

Txn #1: Find K3



SKIP LISTS: SEARCH

Txn #1: Find K3



SKIP LISTS

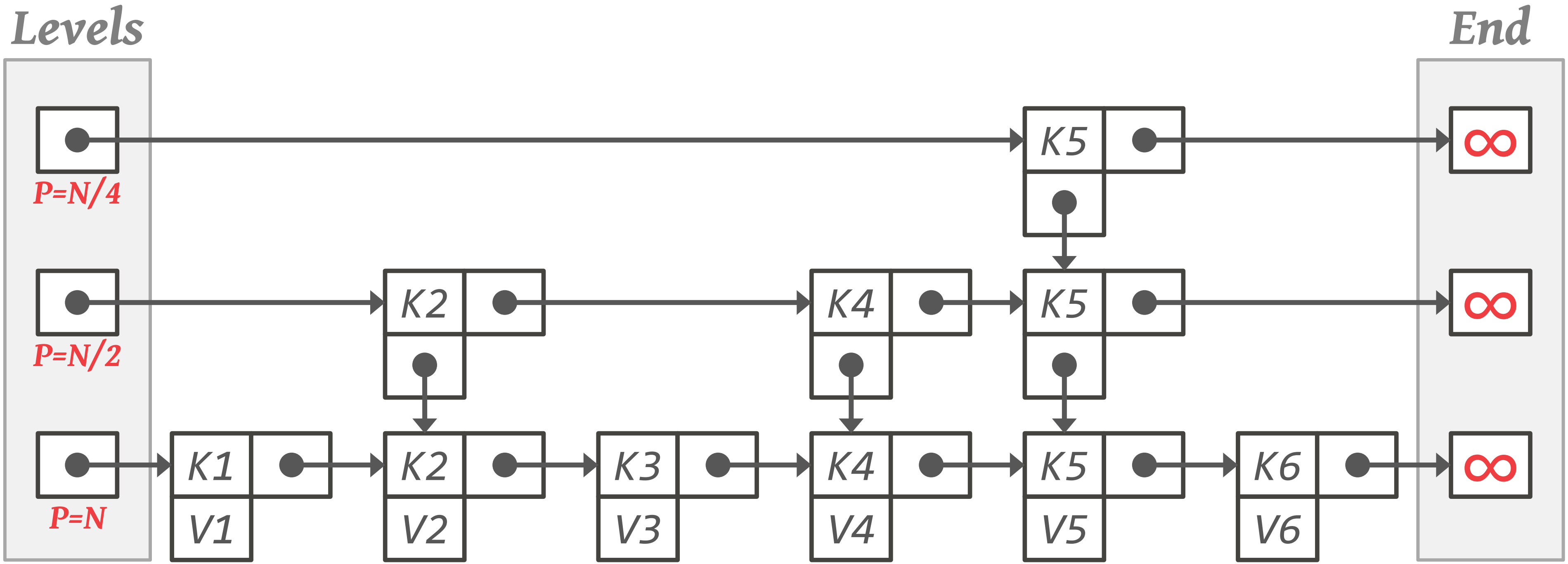
Advantages

- → Uses less memory than a typical B+tree (only if you don’t include reverse pointers).
- → Insertions and deletions do not require rebalancing.
- → It is possible to implement a concurrent skip list using only CAS instructions.

Disadvantages

- → Not cache friendly because they do not optimize locality of references.
- → Reverse search is non-trivial.

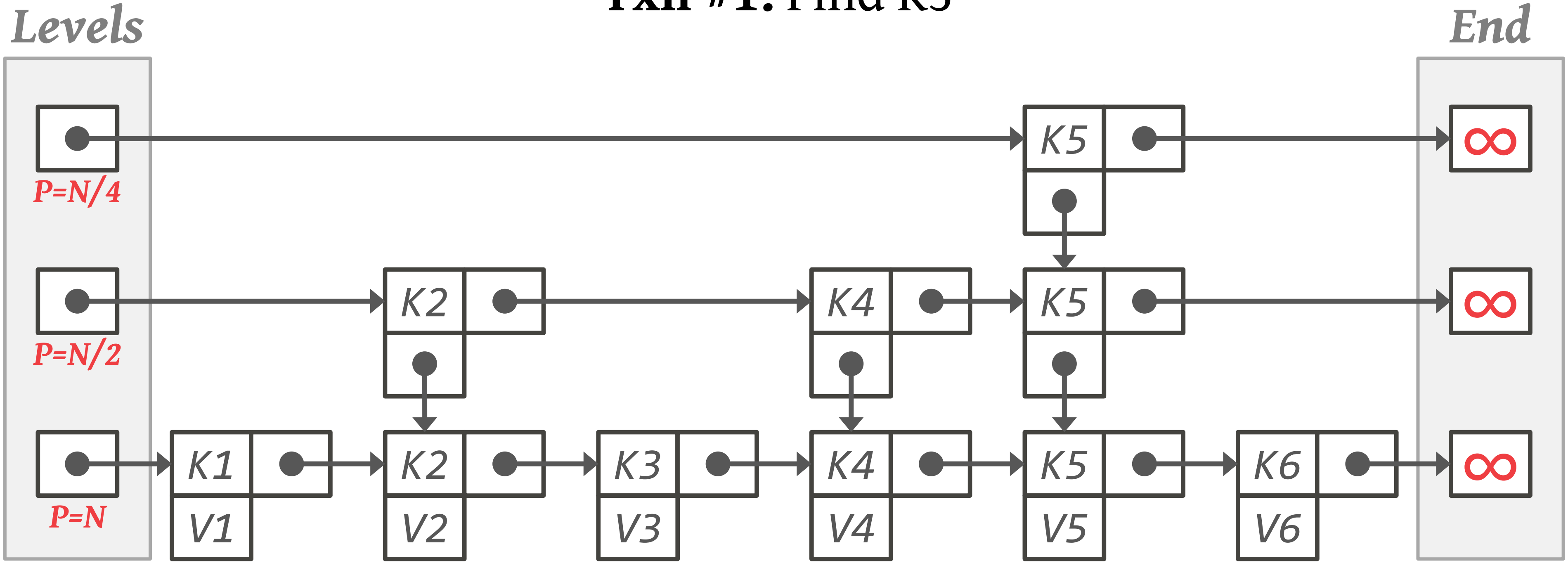
SKIP LISTS: REVERSE SEARCH



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

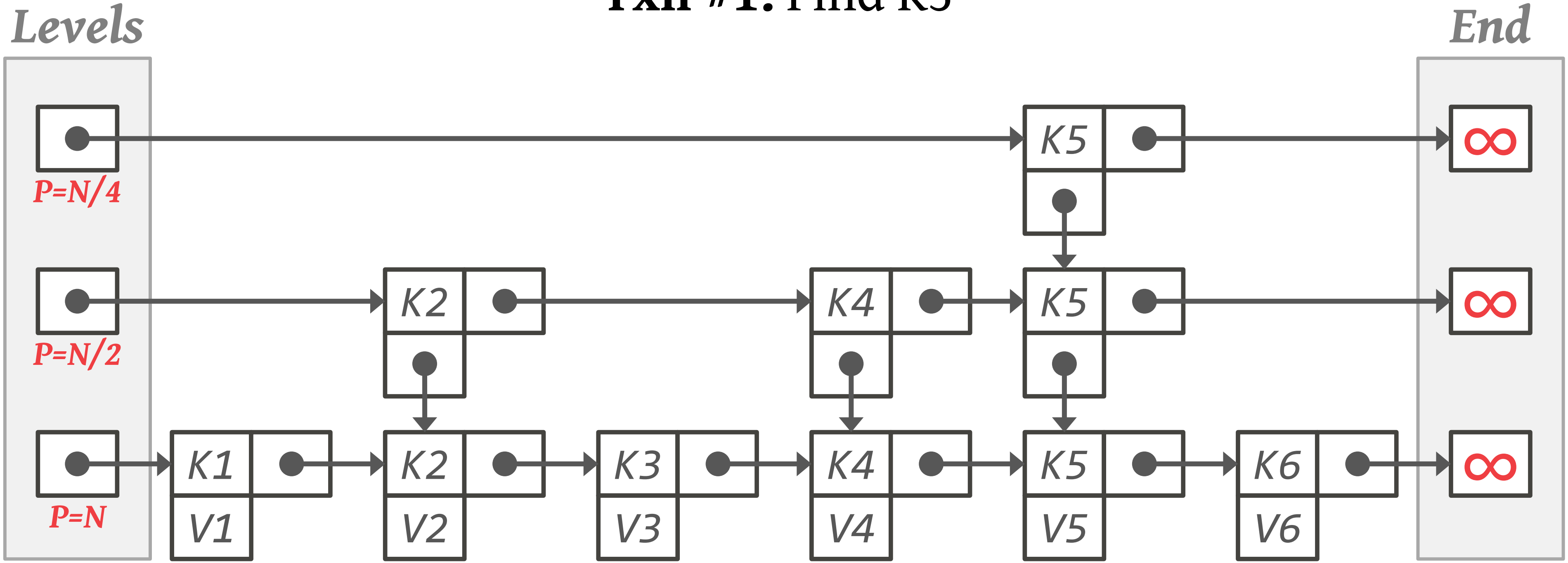
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

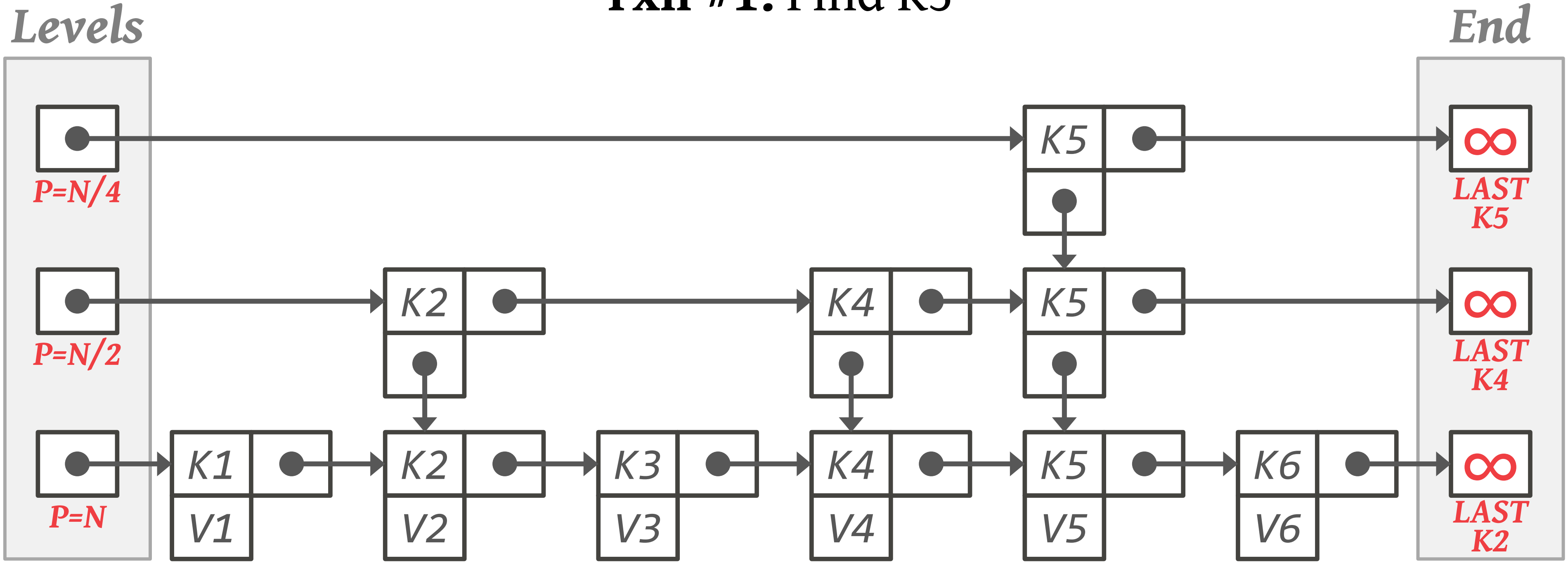
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

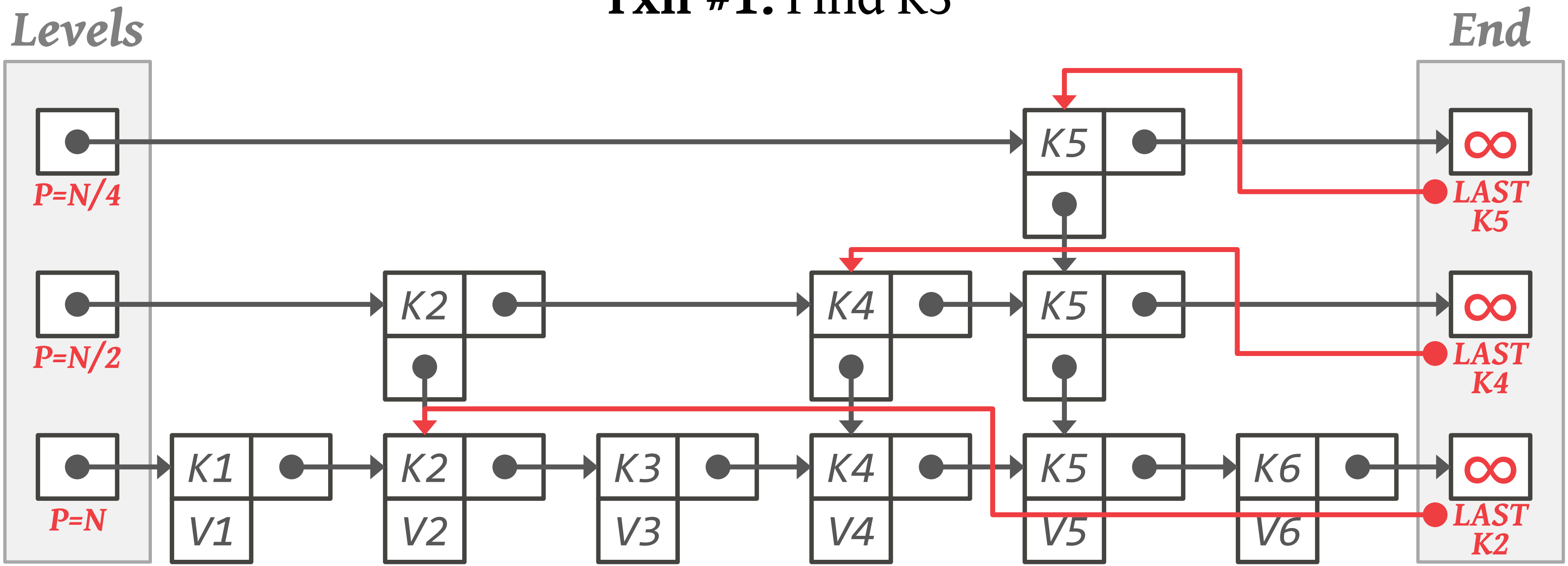
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

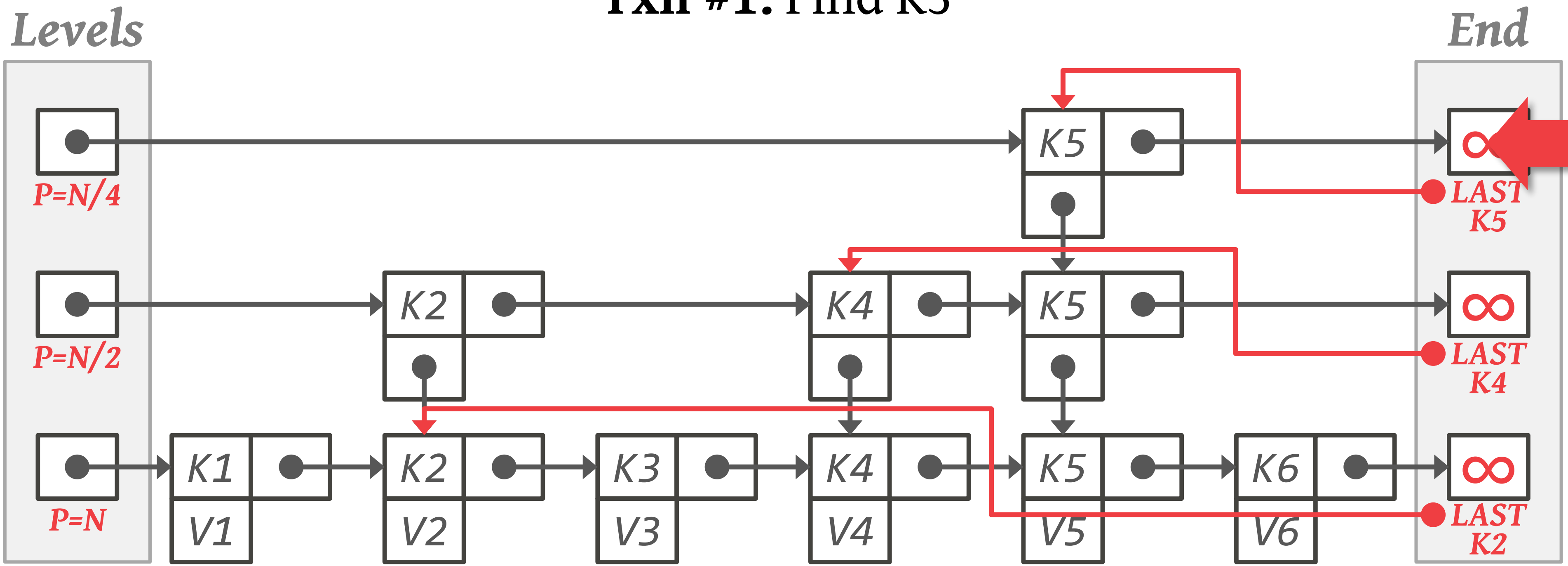
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

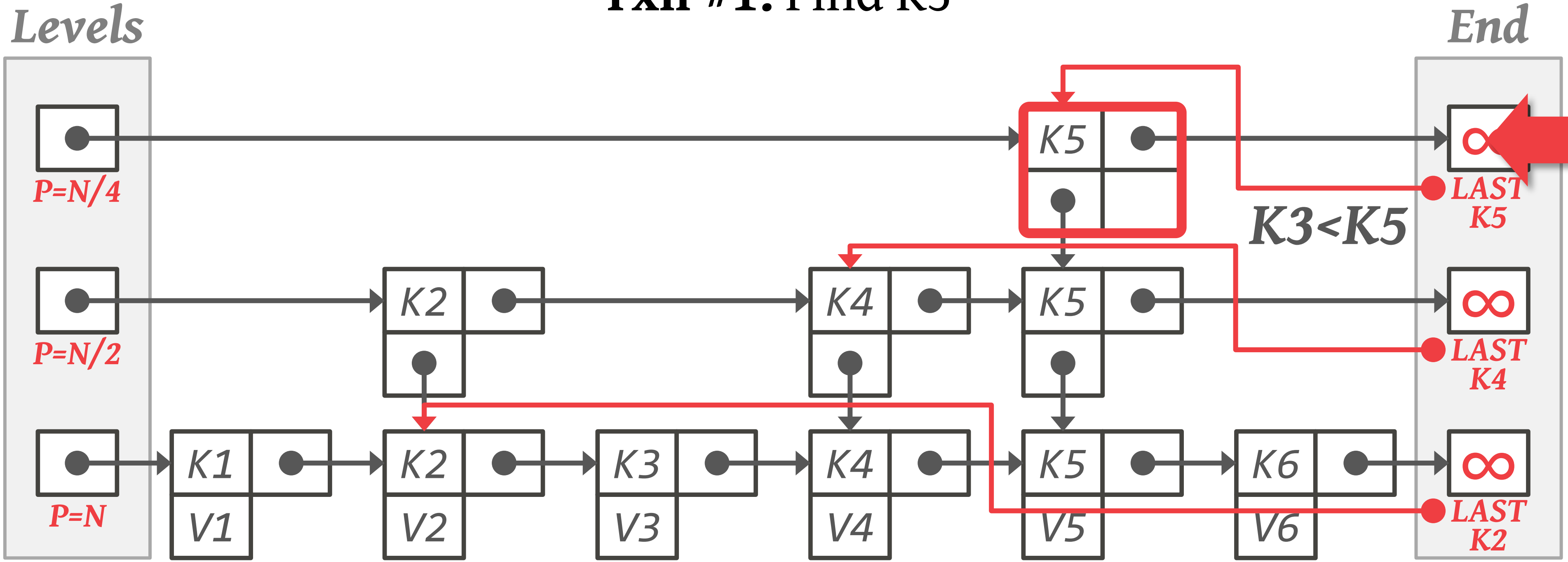
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

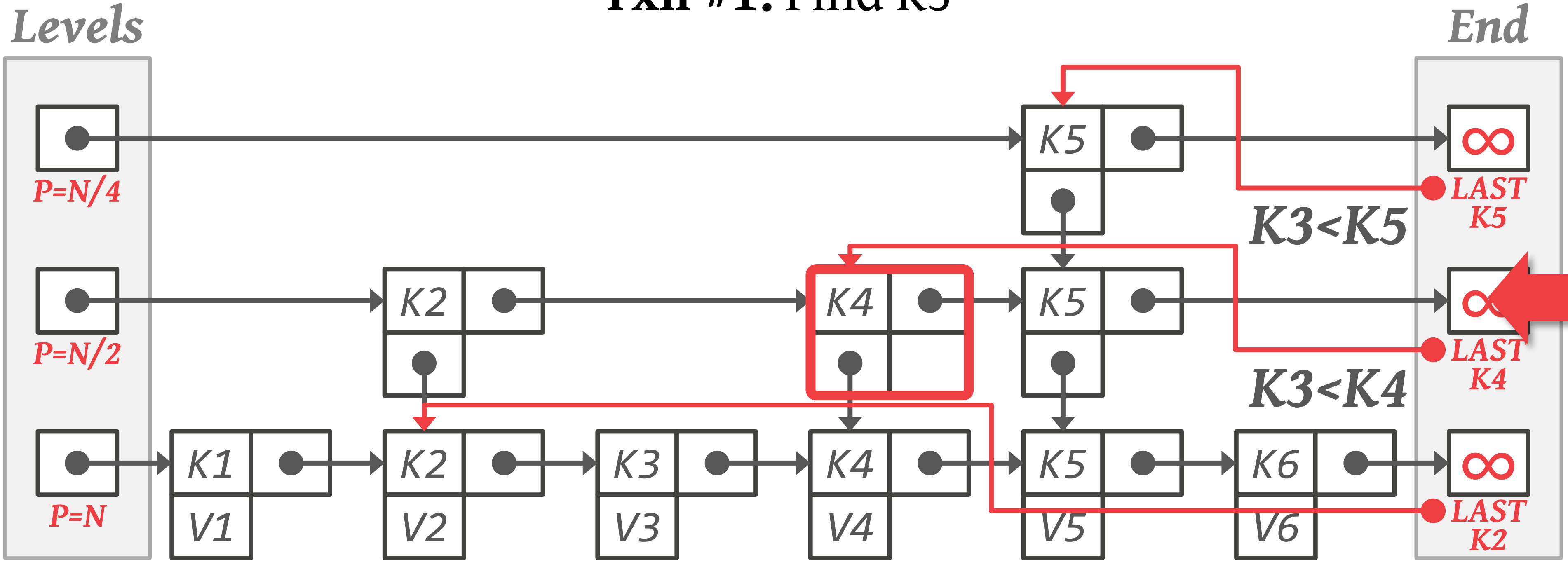
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

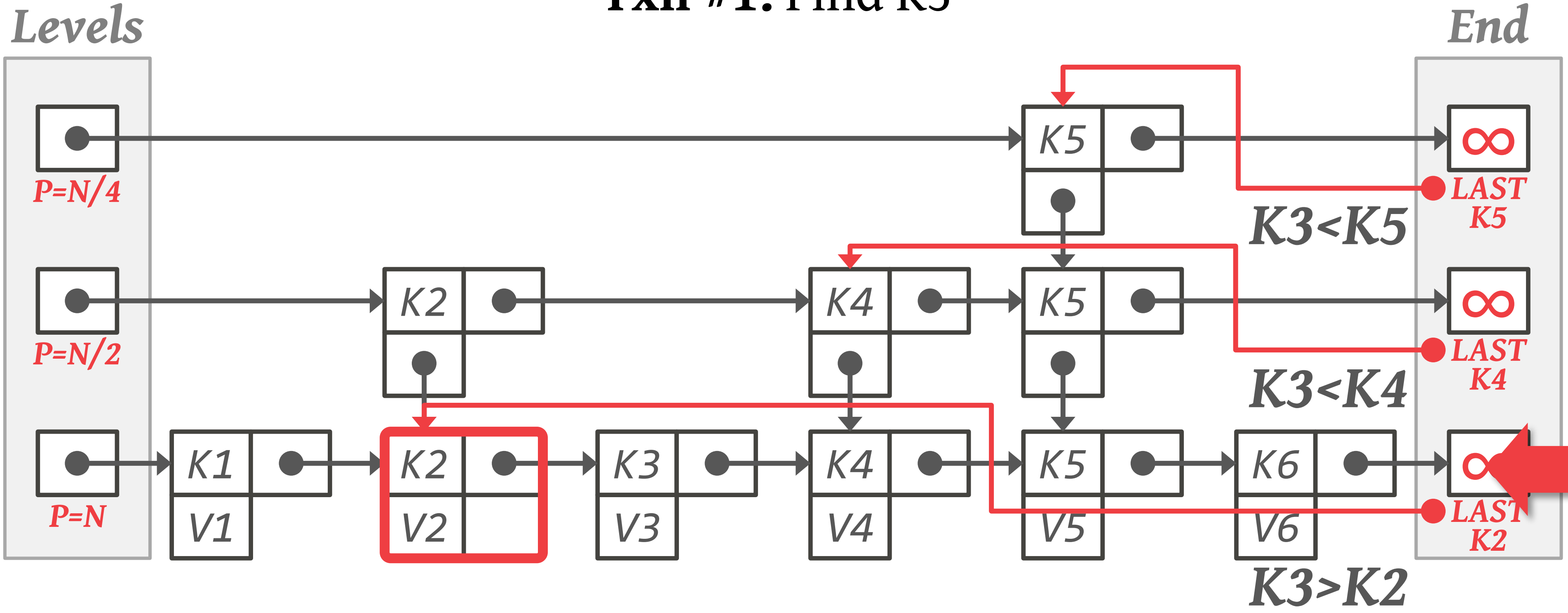
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

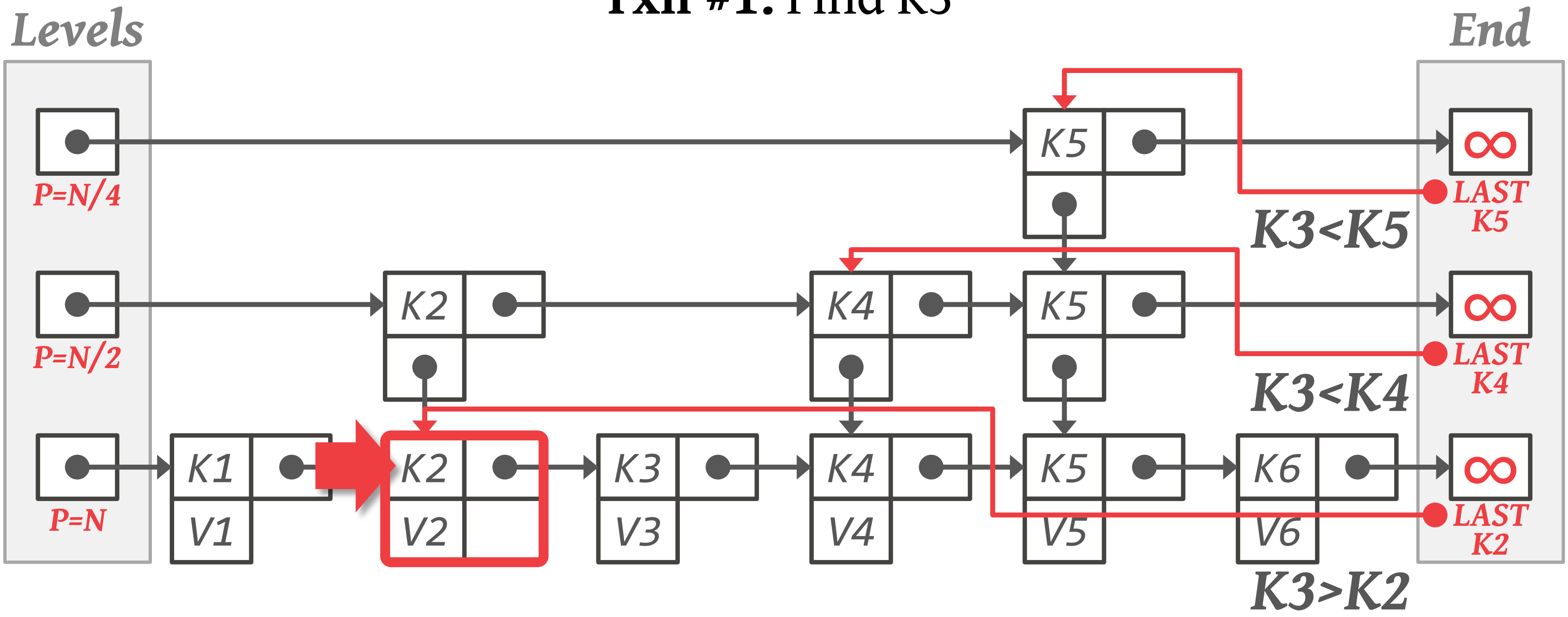
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

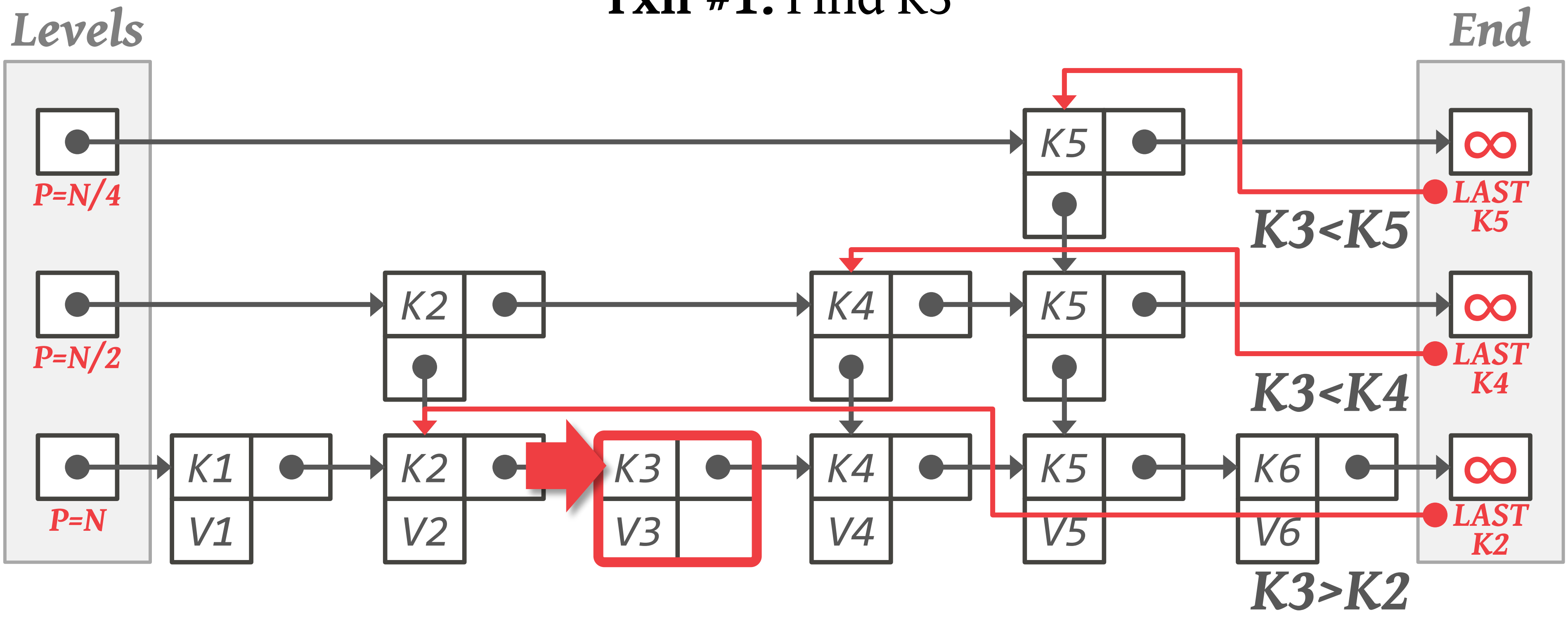
Txn #1: Find K3



Source: MemSQL

SKIP LISTS: REVERSE SEARCH

Txn #1: Find K3



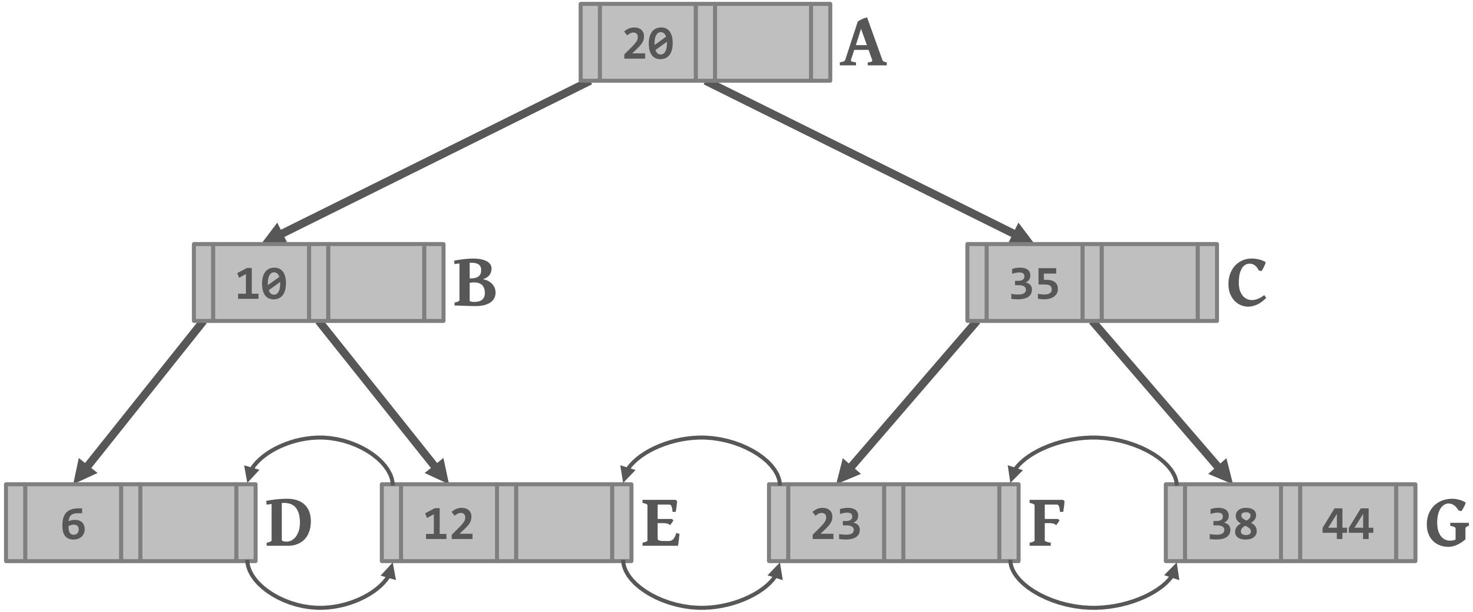
Source: MemSQL

WHY ARE INDEXES DIFFERENT?

The DBMS has to treat locking in indexes differently than how its concurrency control scheme manages database objects.

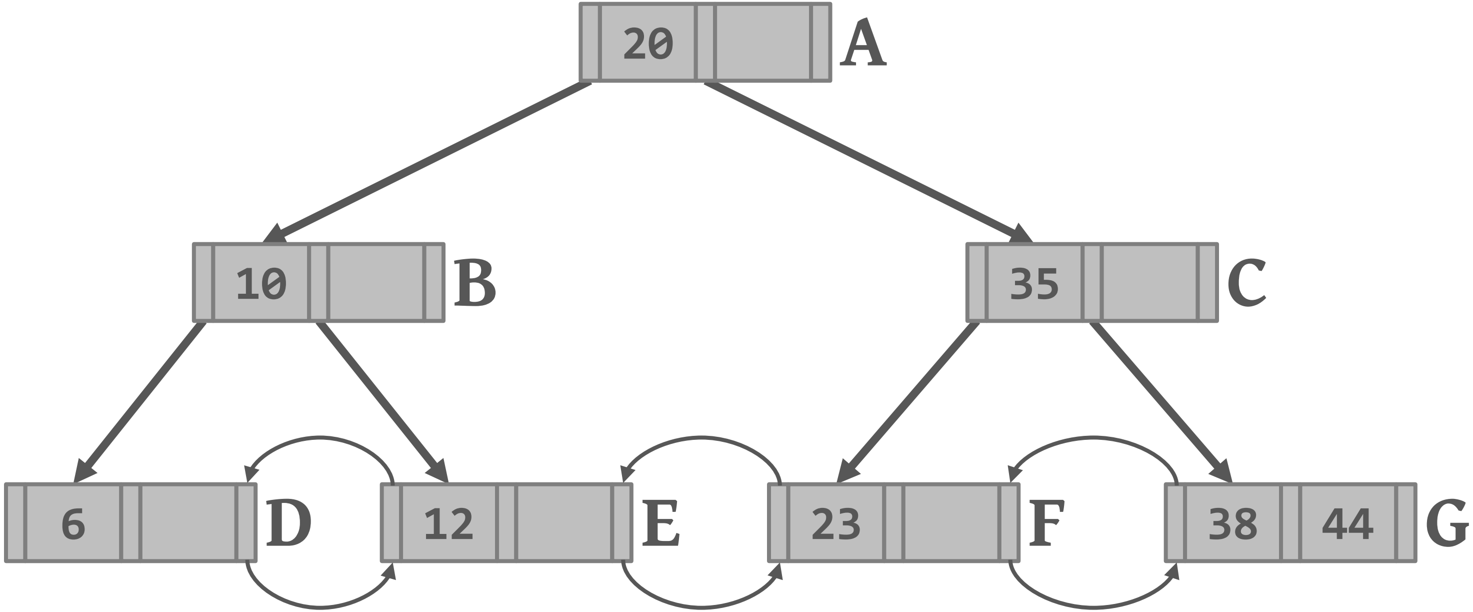
The physical structure can change as long as the logical contents are consistent.

PROBLEM SCENARIO #1



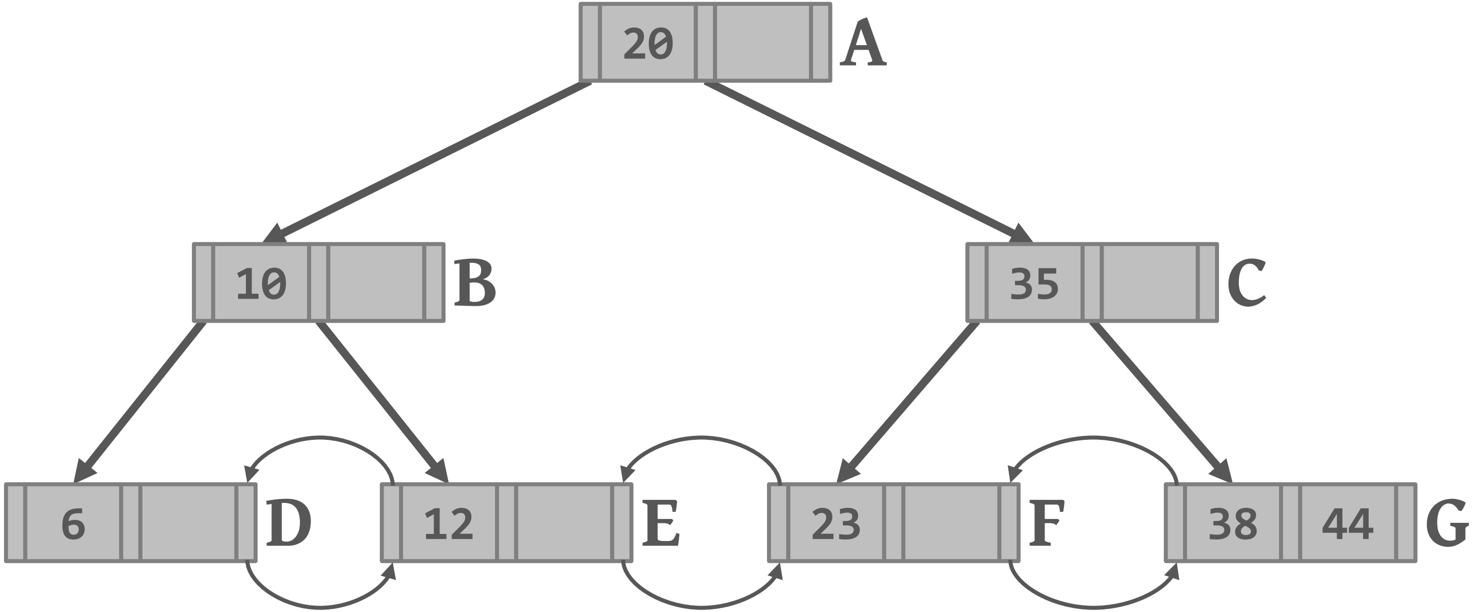
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



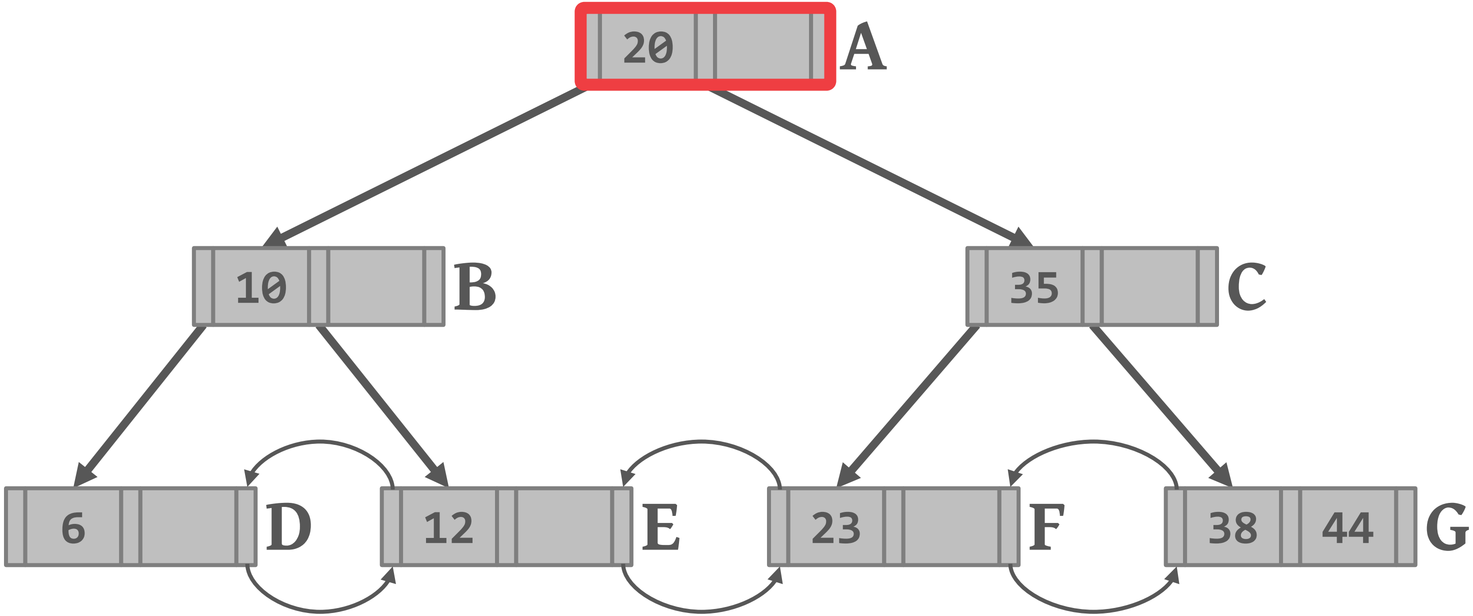
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



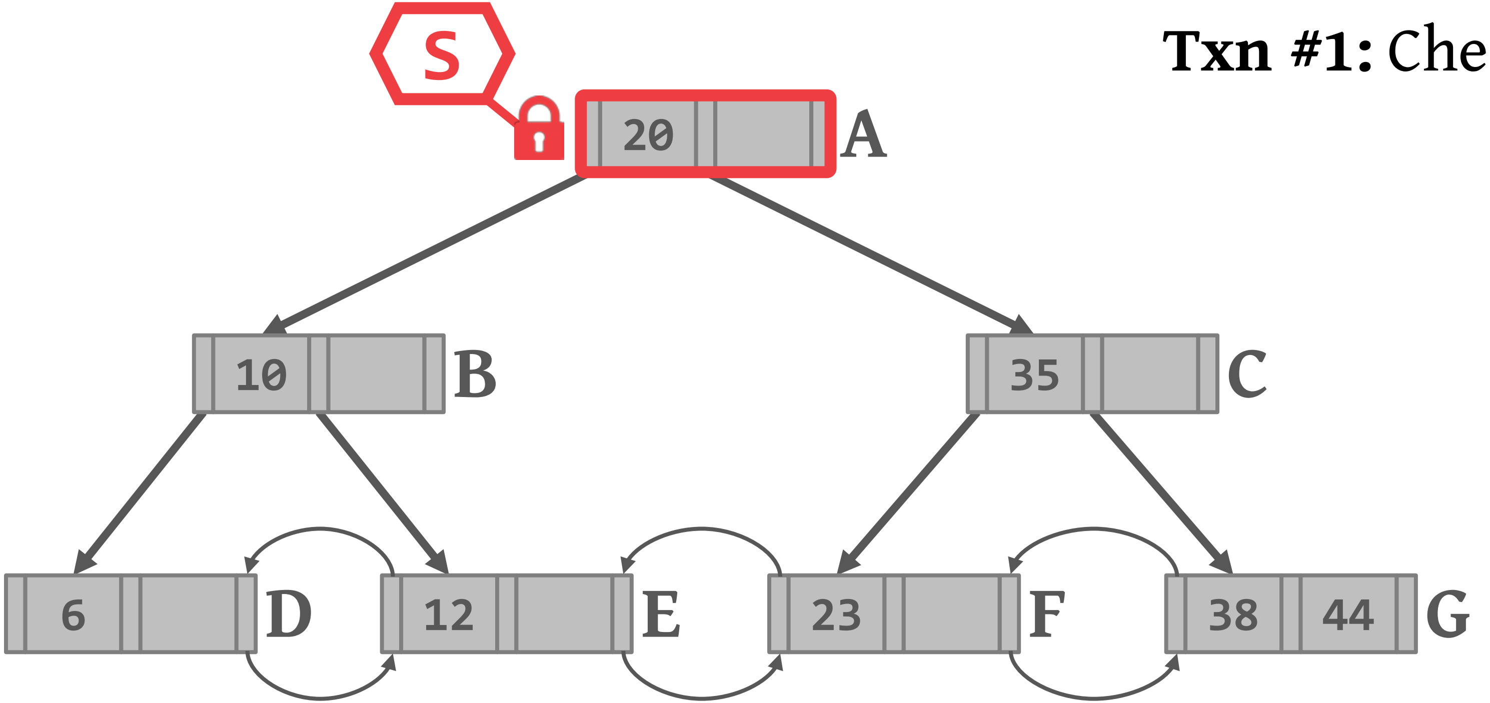
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



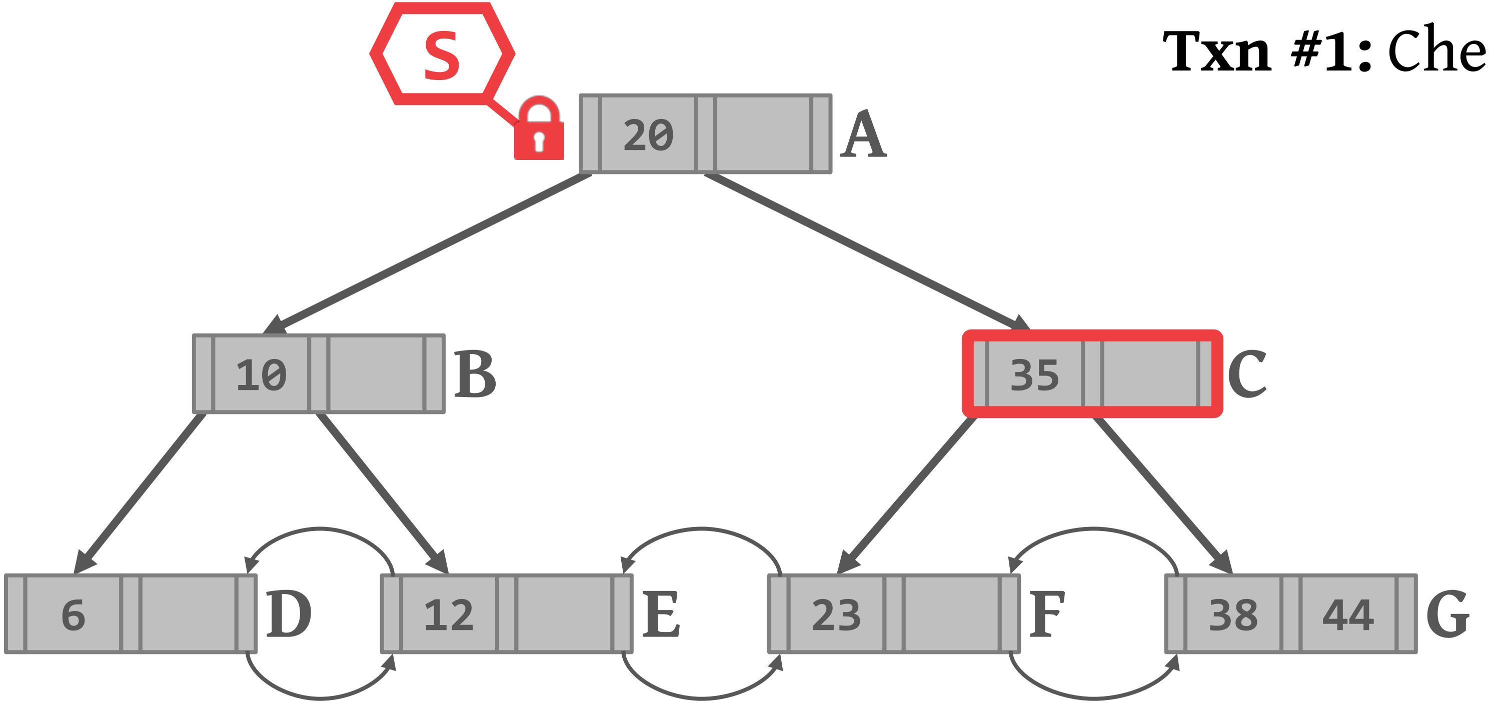
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



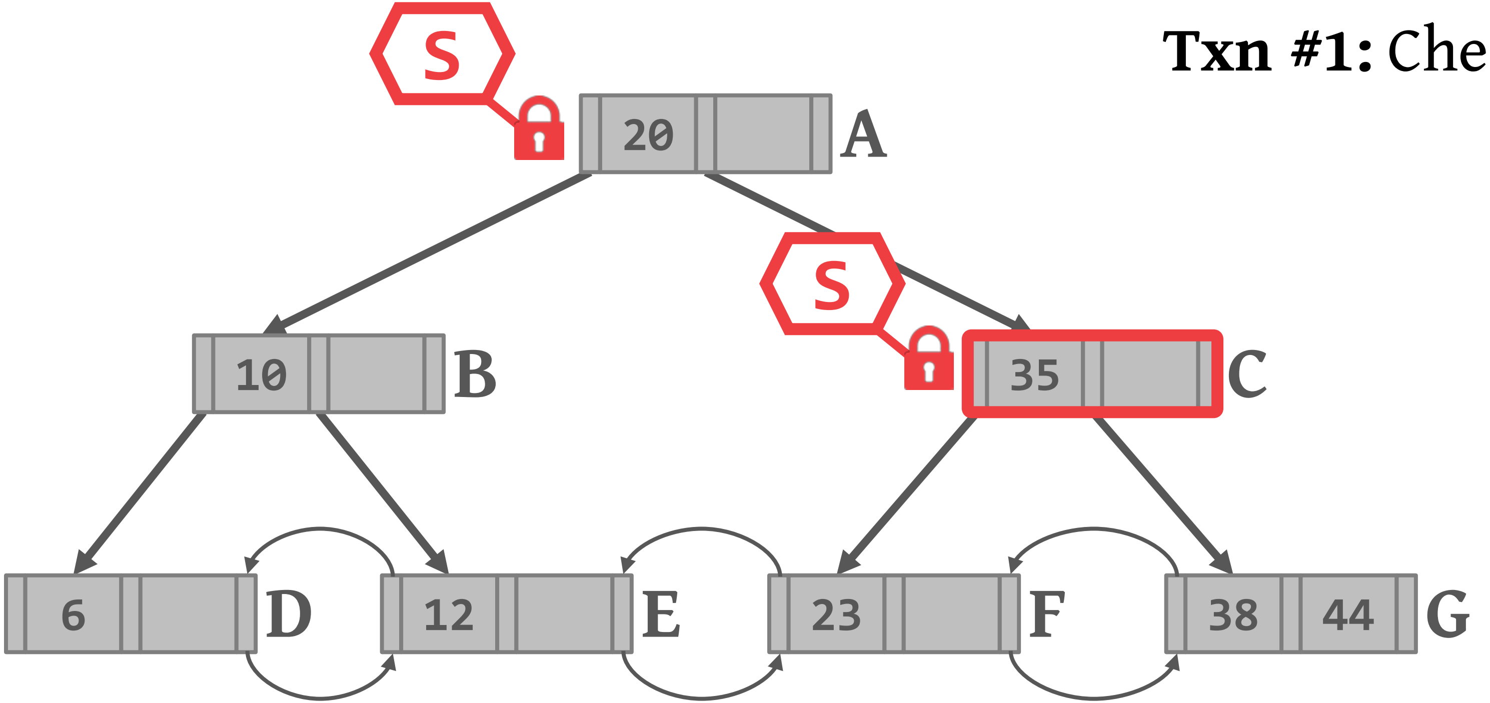
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



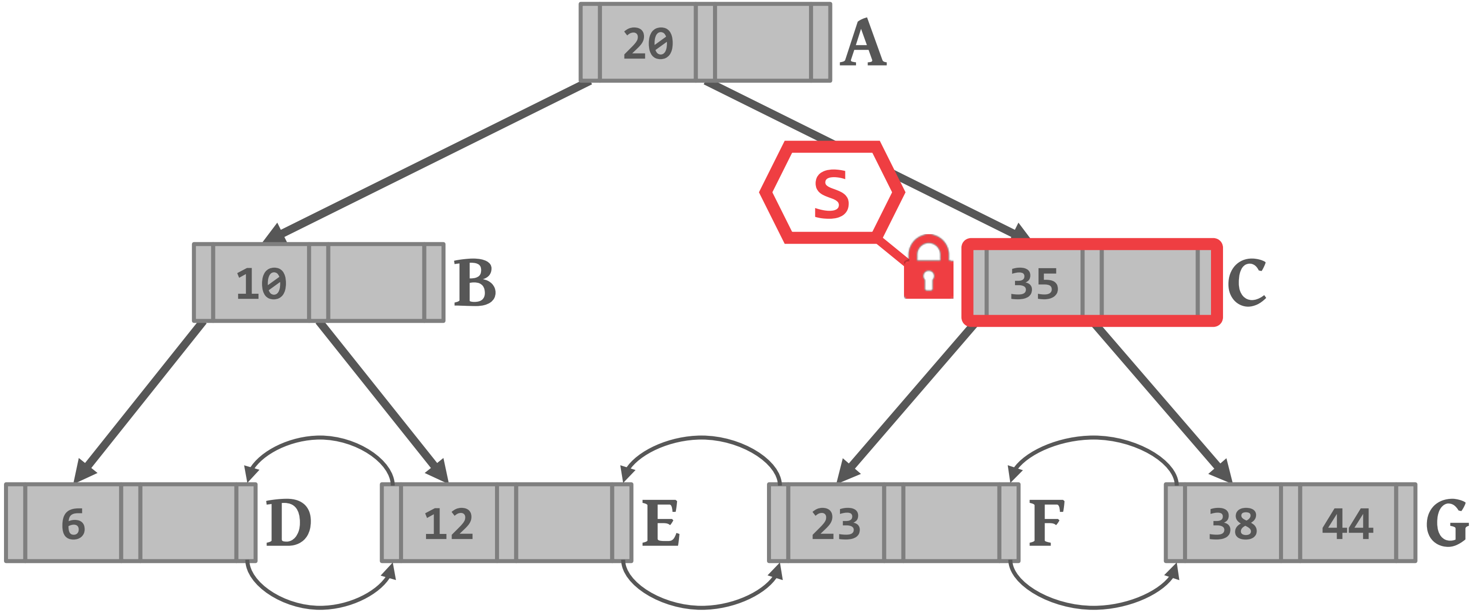
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



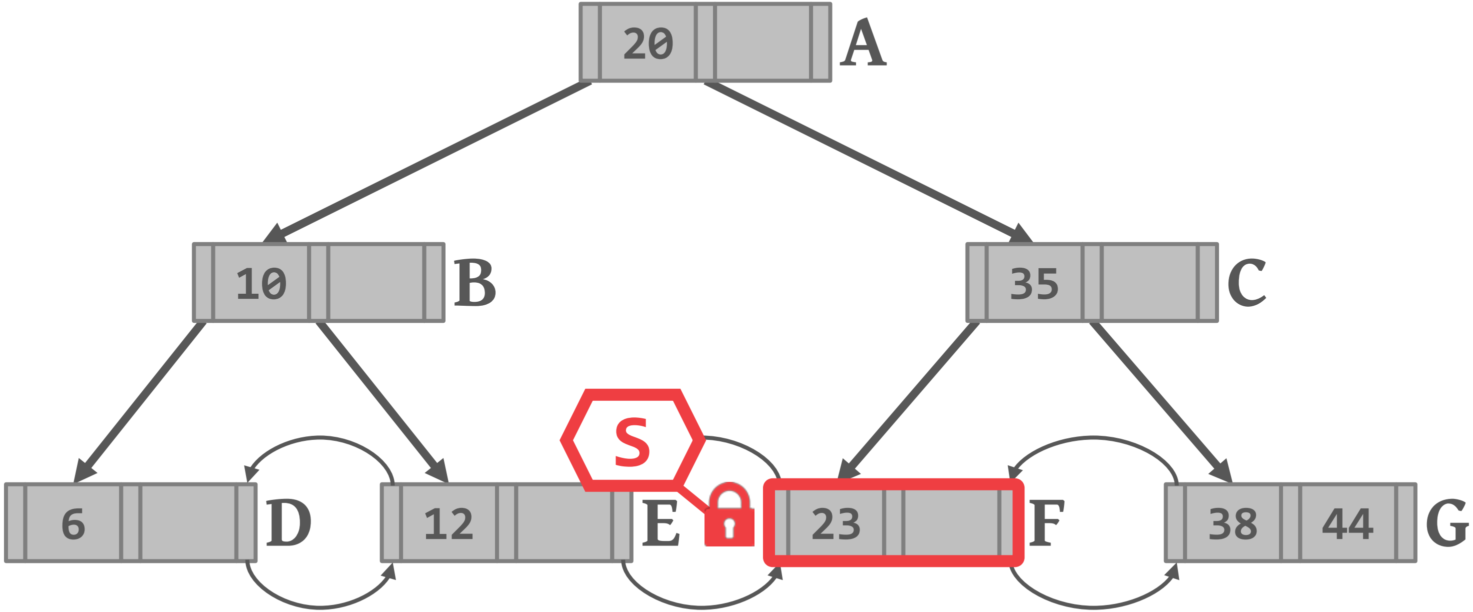
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



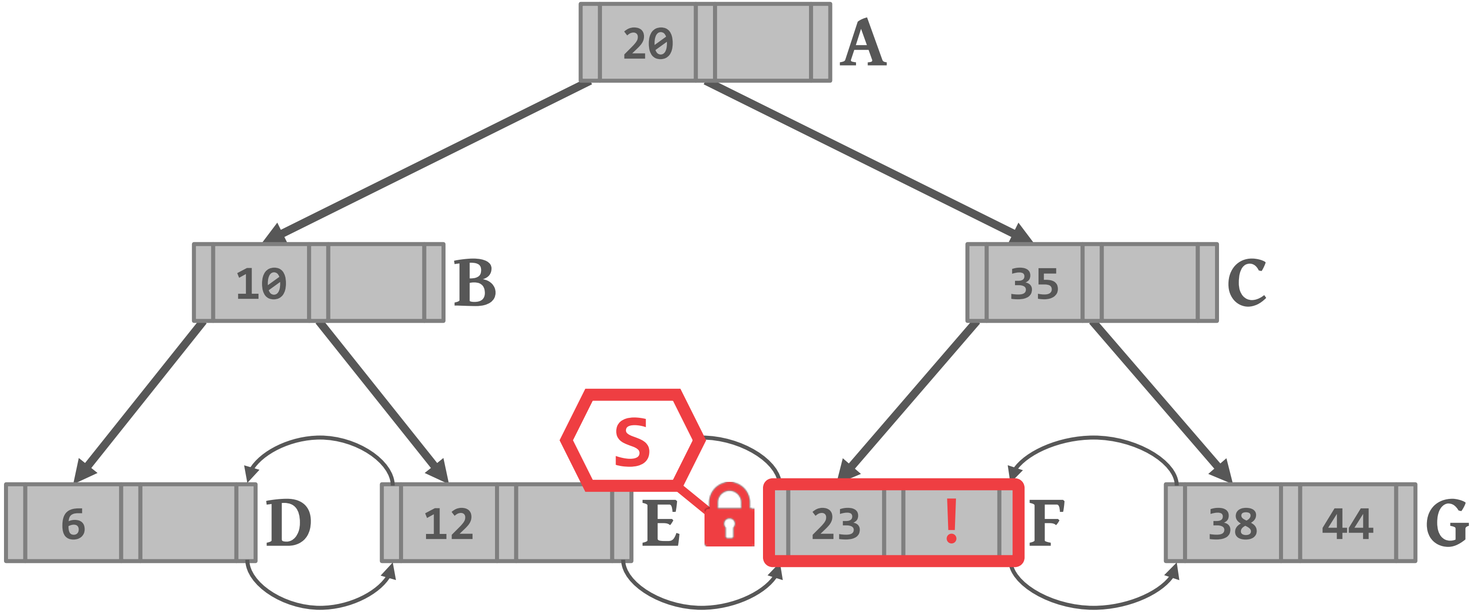
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



PROBLEM SCENARIO #1

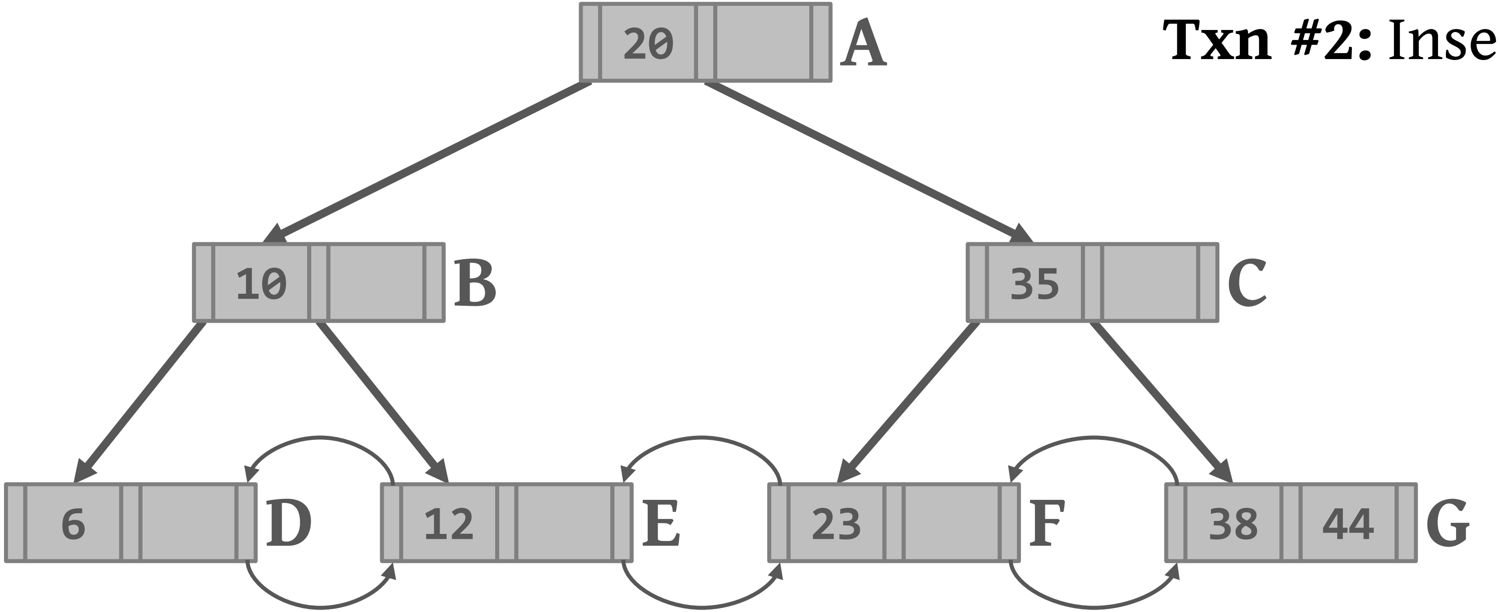
Txn #1: Check if 25 exists



PROBLEM SCENARIO #1

Txn #1: Check if 25 exists

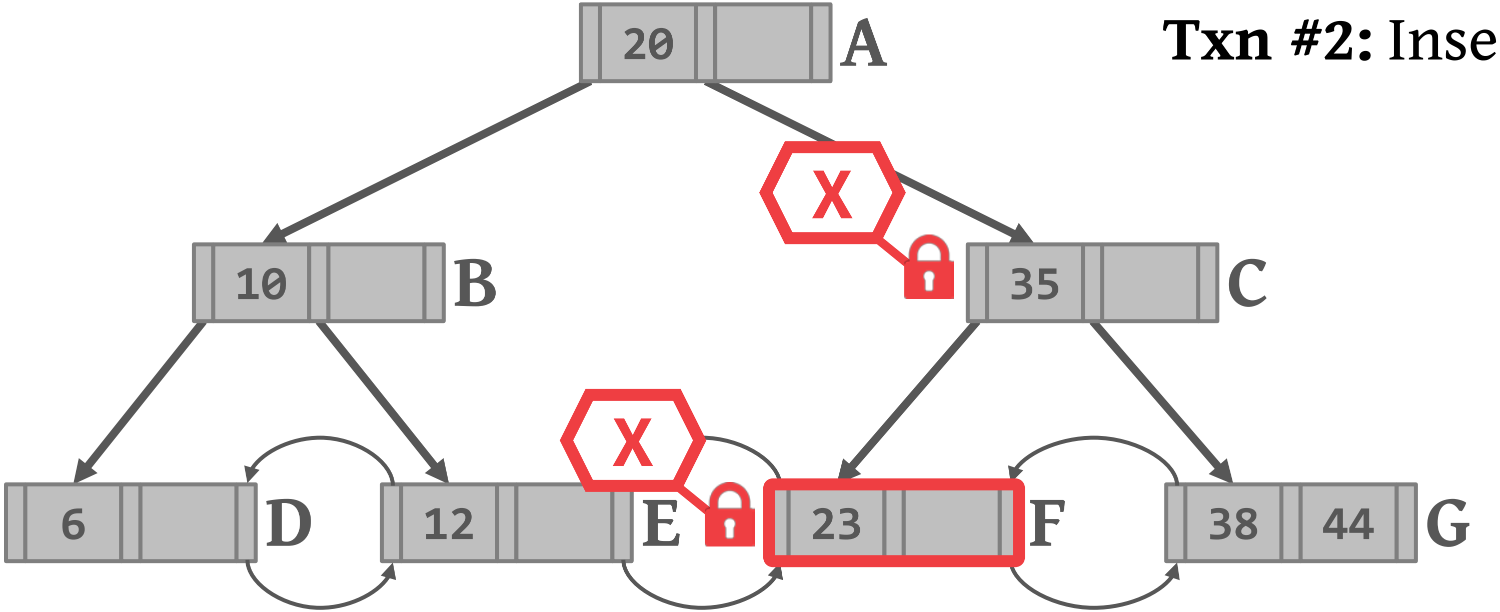
Txn #2: Insert 25



PROBLEM SCENARIO #1

Txn #1: Check if 25 exists

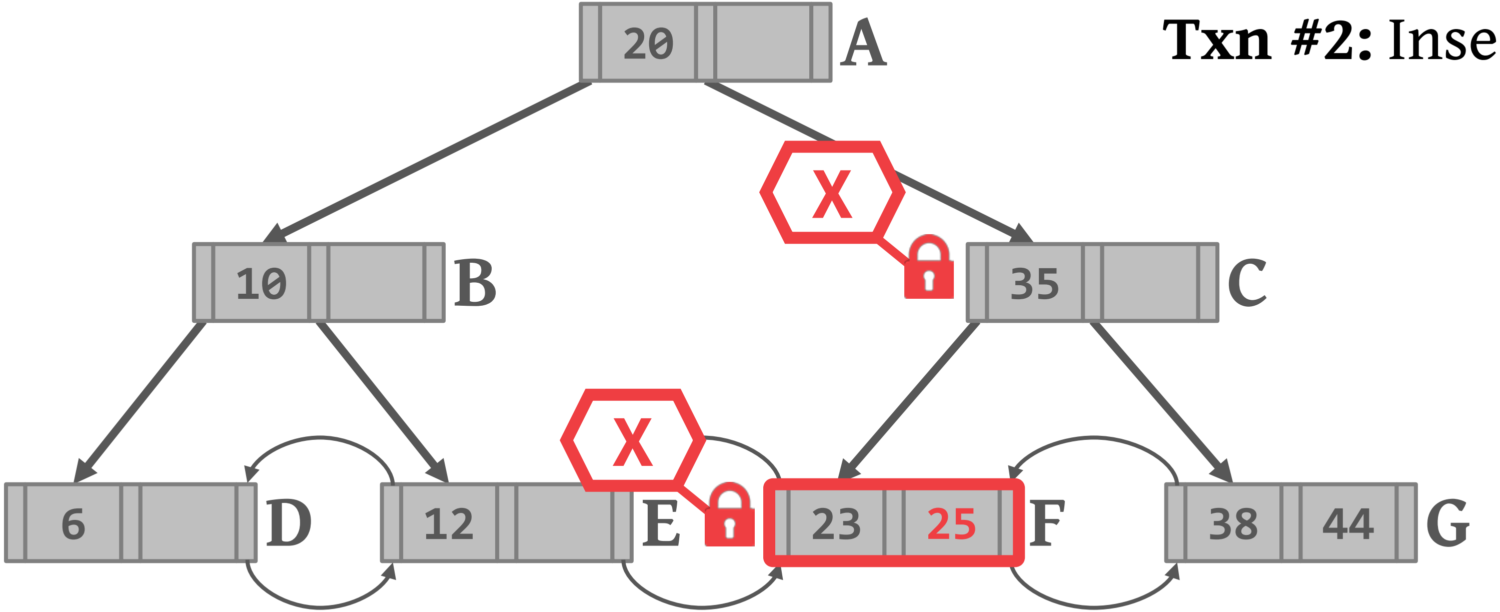
Txn #2: Insert 25



PROBLEM SCENARIO #1

Txn #1: Check if 25 exists

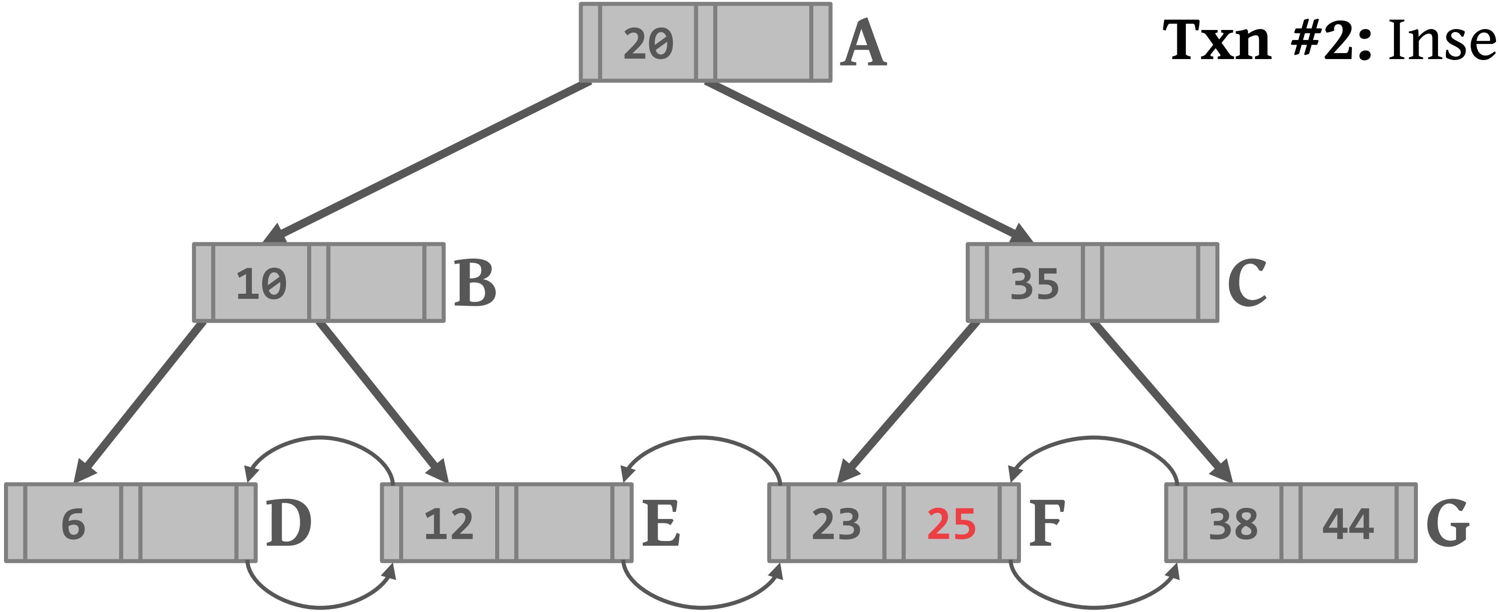
Txn #2: Insert 25



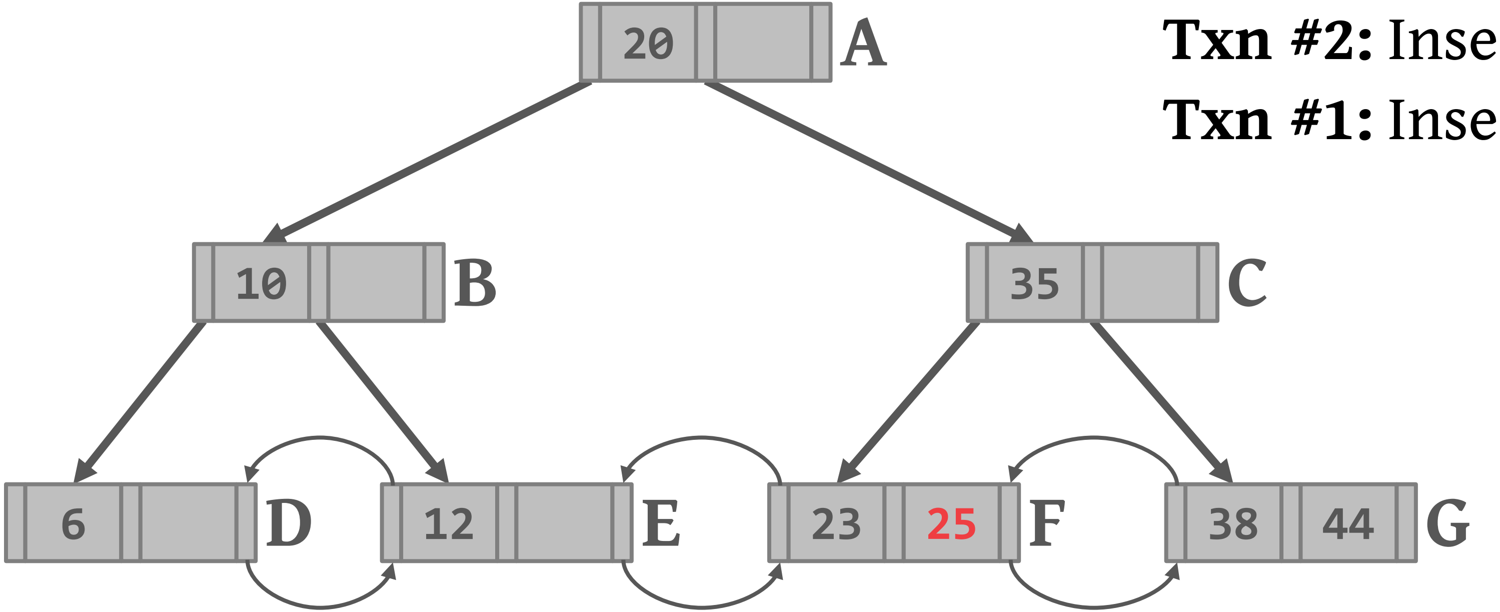
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists

Txn #2: Insert 25



PROBLEM SCENARIO #1



Txn #1: Check if 25 exists

Txn #2: Insert 25

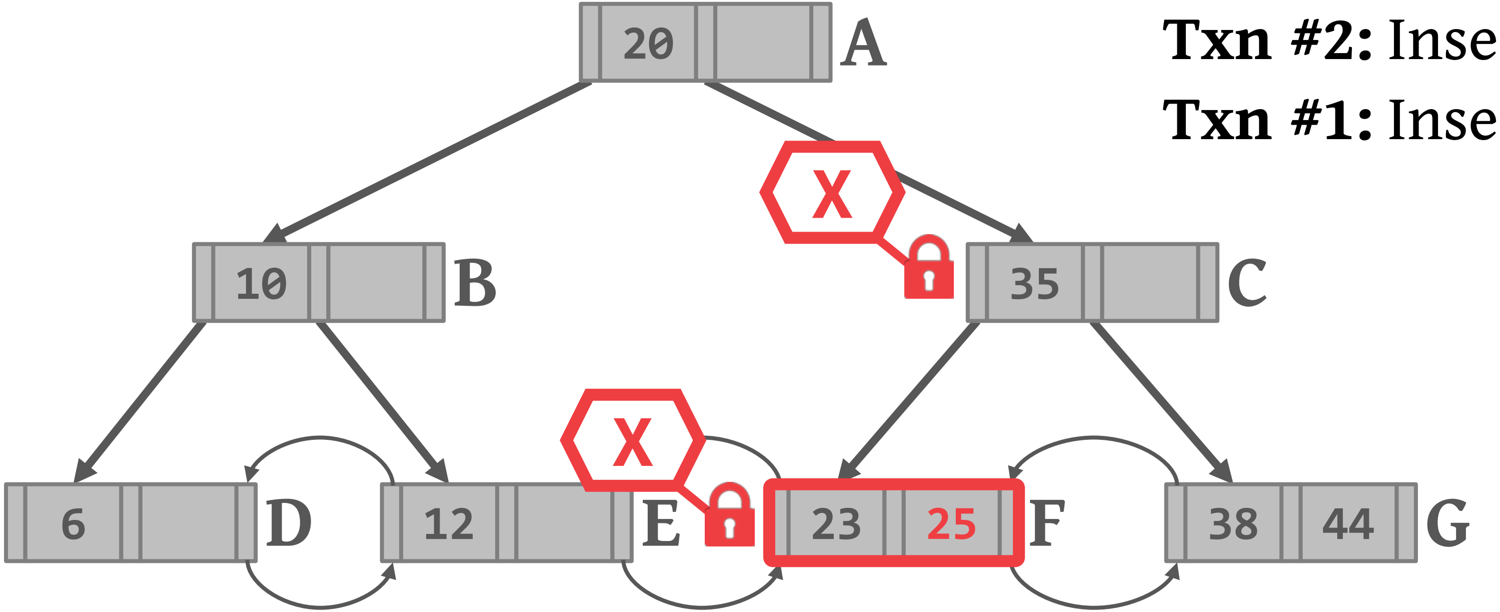
Txn #1: Insert 25

PROBLEM SCENARIO #1

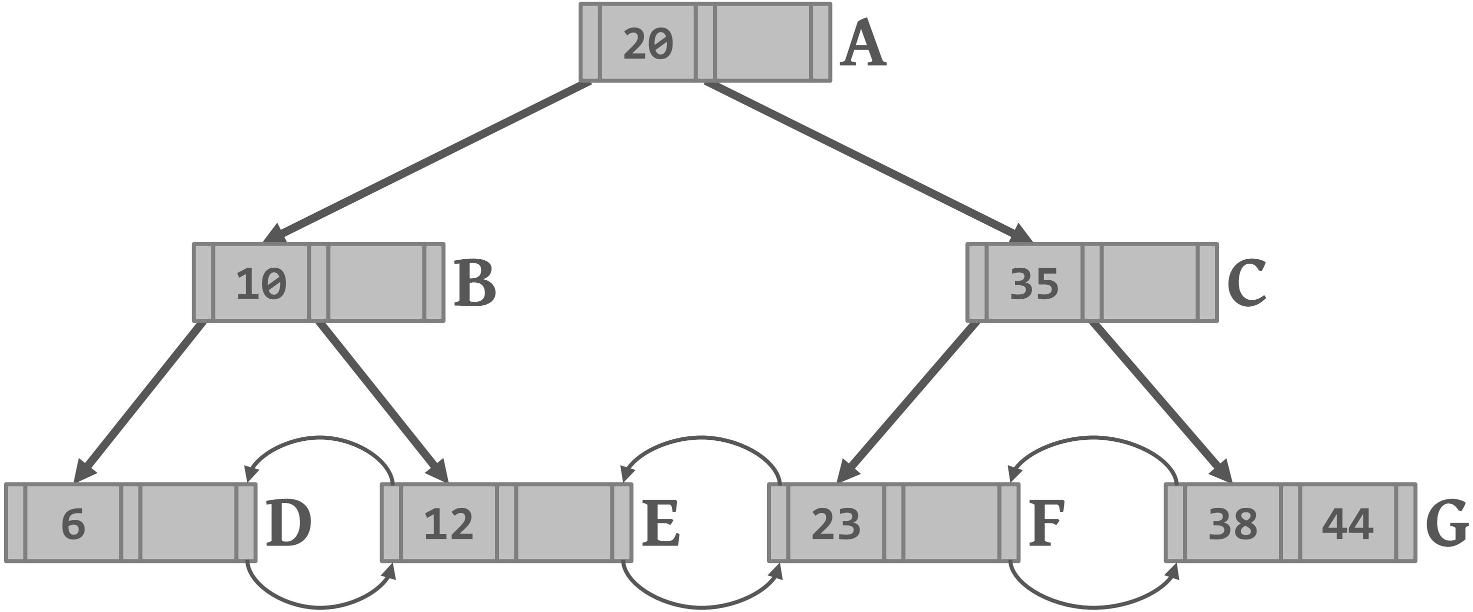
Txn #1: Check if 25 exists

Txn #2: Insert 25

Txn #1: Insert 25

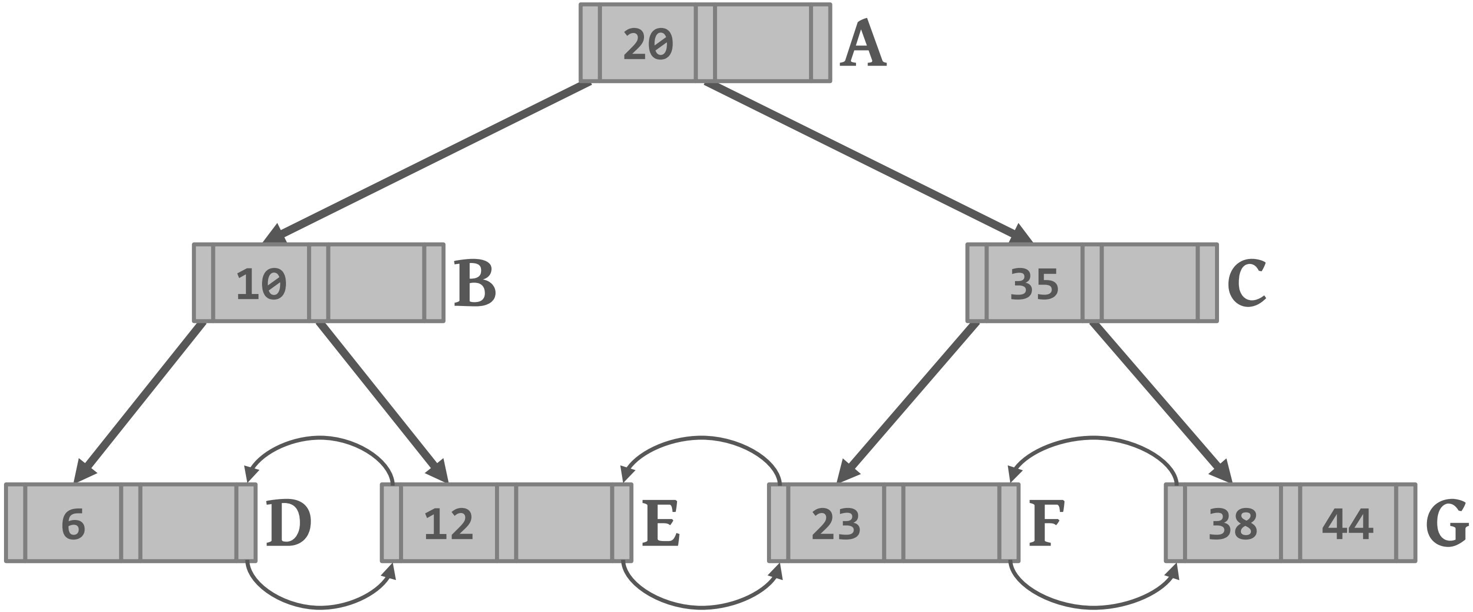


PROBLEM SCENARIO #2



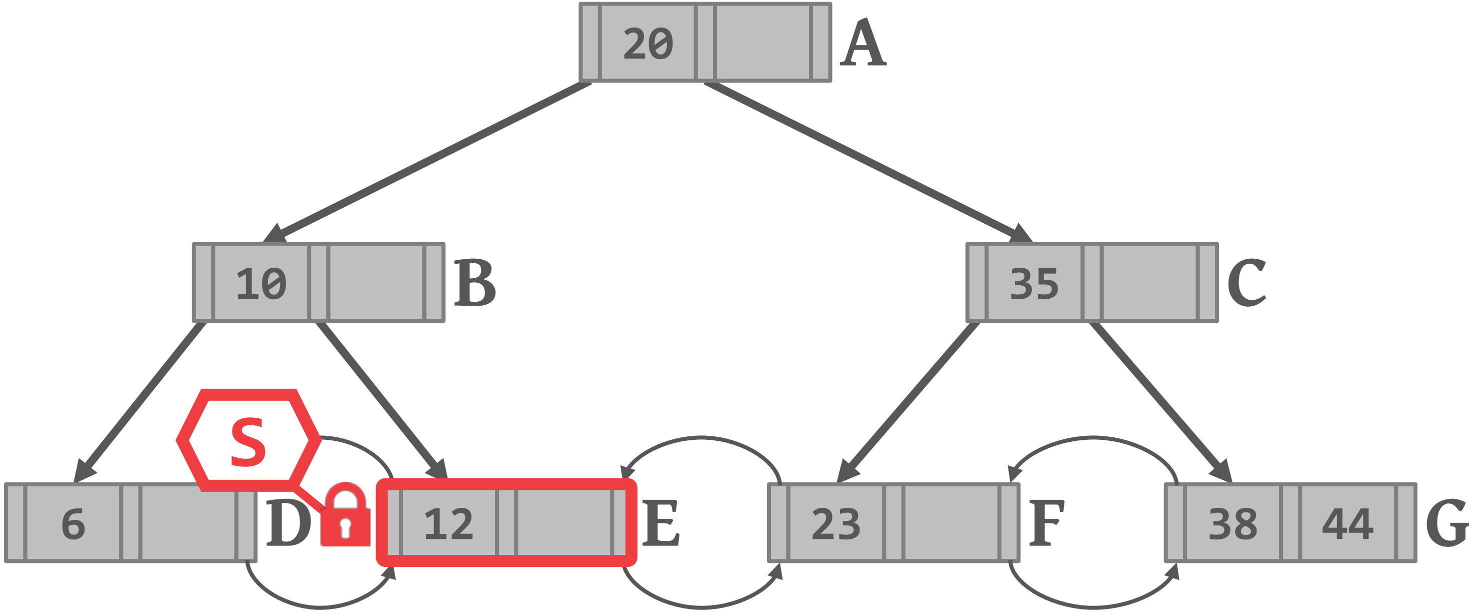
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]



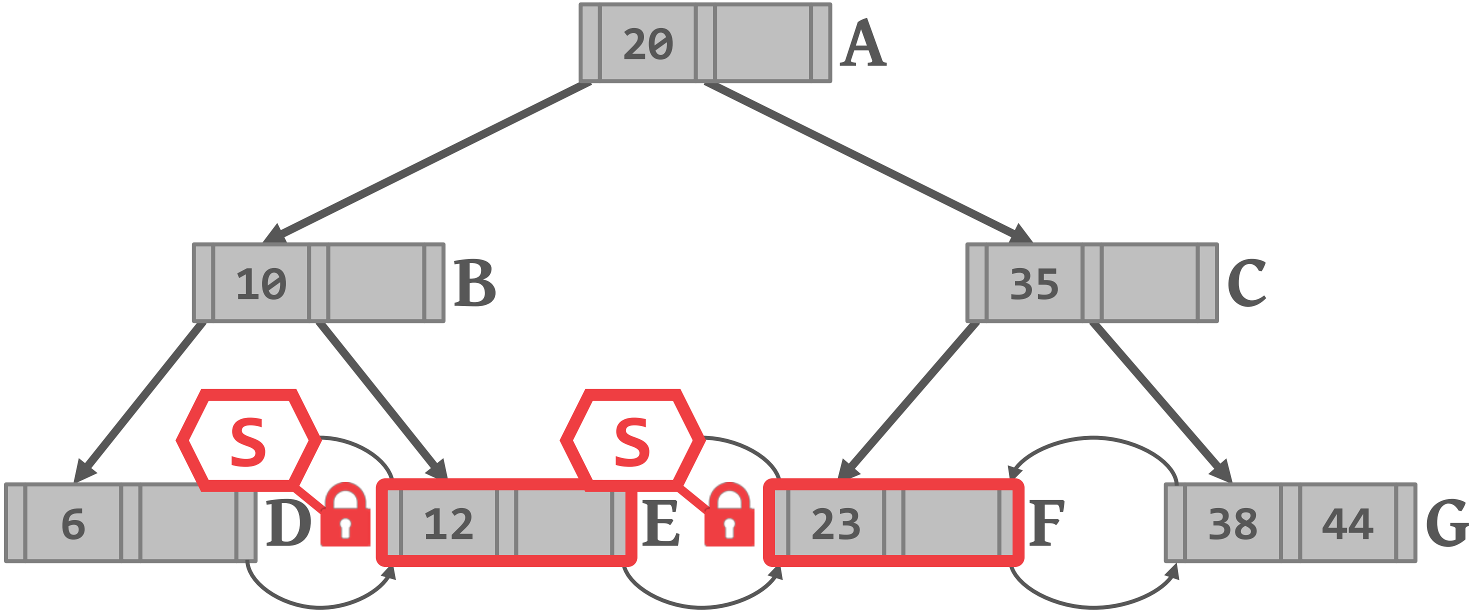
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]



PROBLEM SCENARIO #2

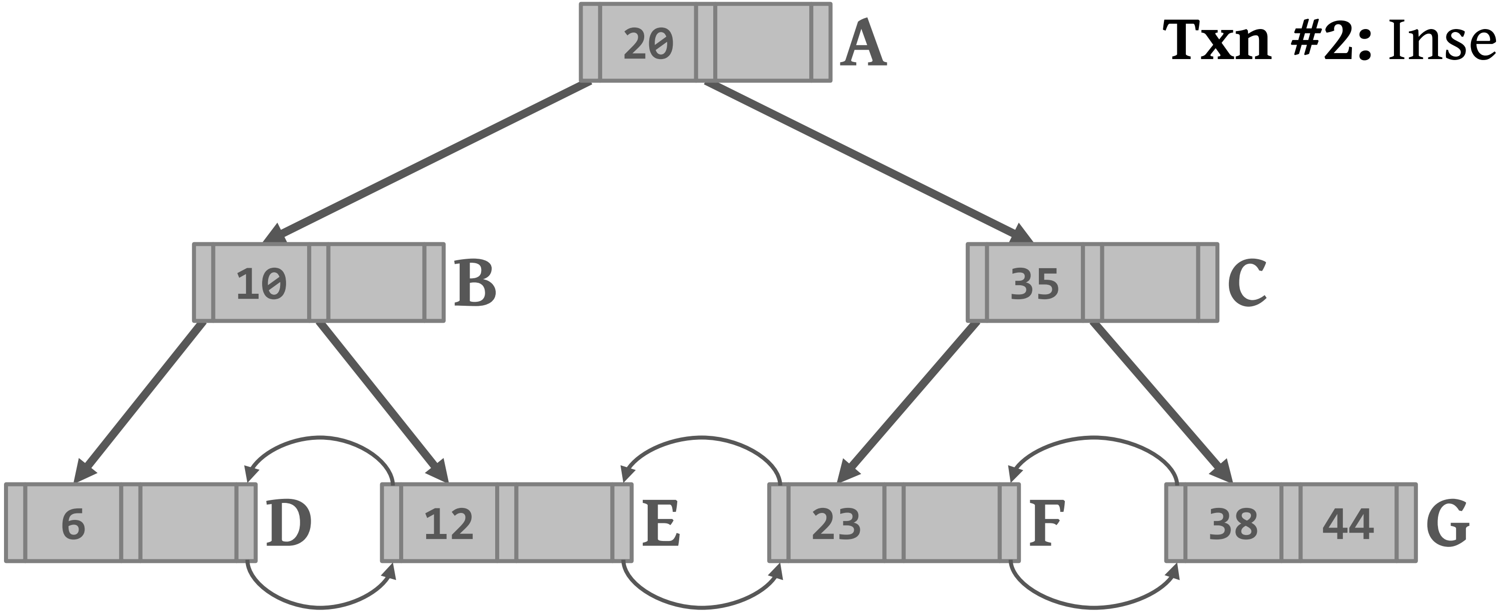
Txn #1: Scan [12, 23]



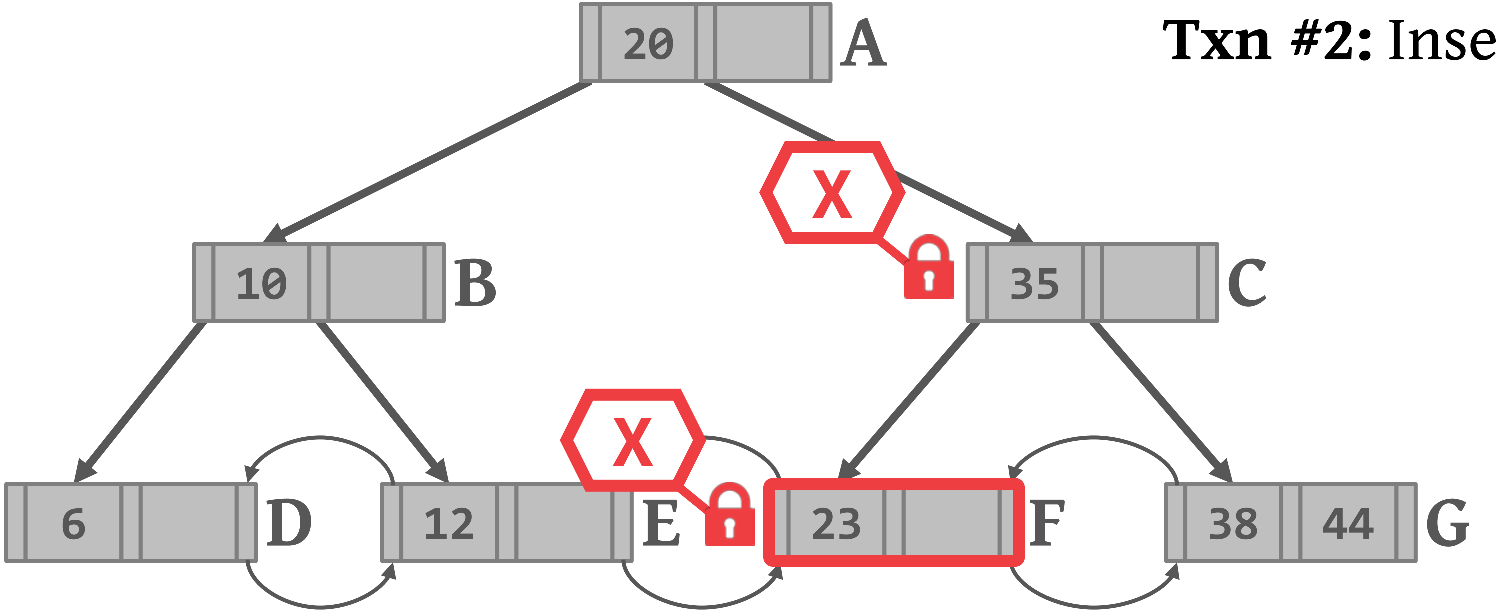
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]

Txn #2: Insert 21



PROBLEM SCENARIO #2



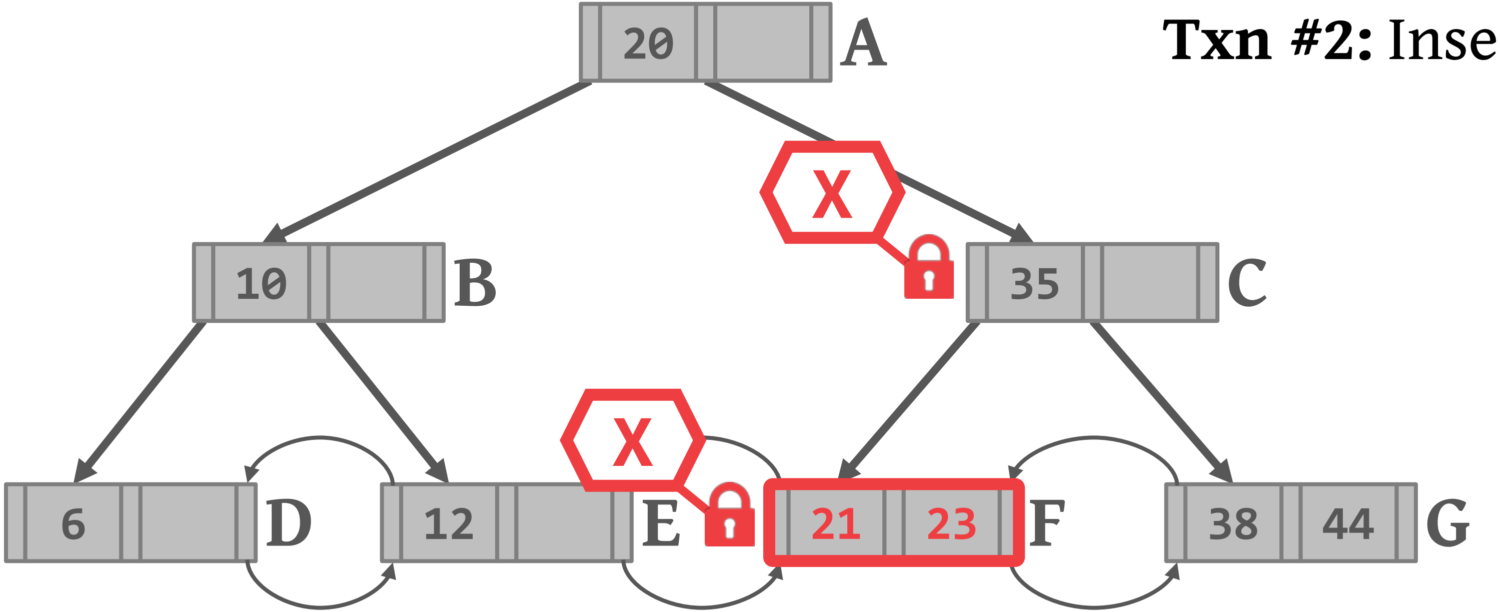
Txn #1: Scan [12, 23]

Txn #2: Insert 21

PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]

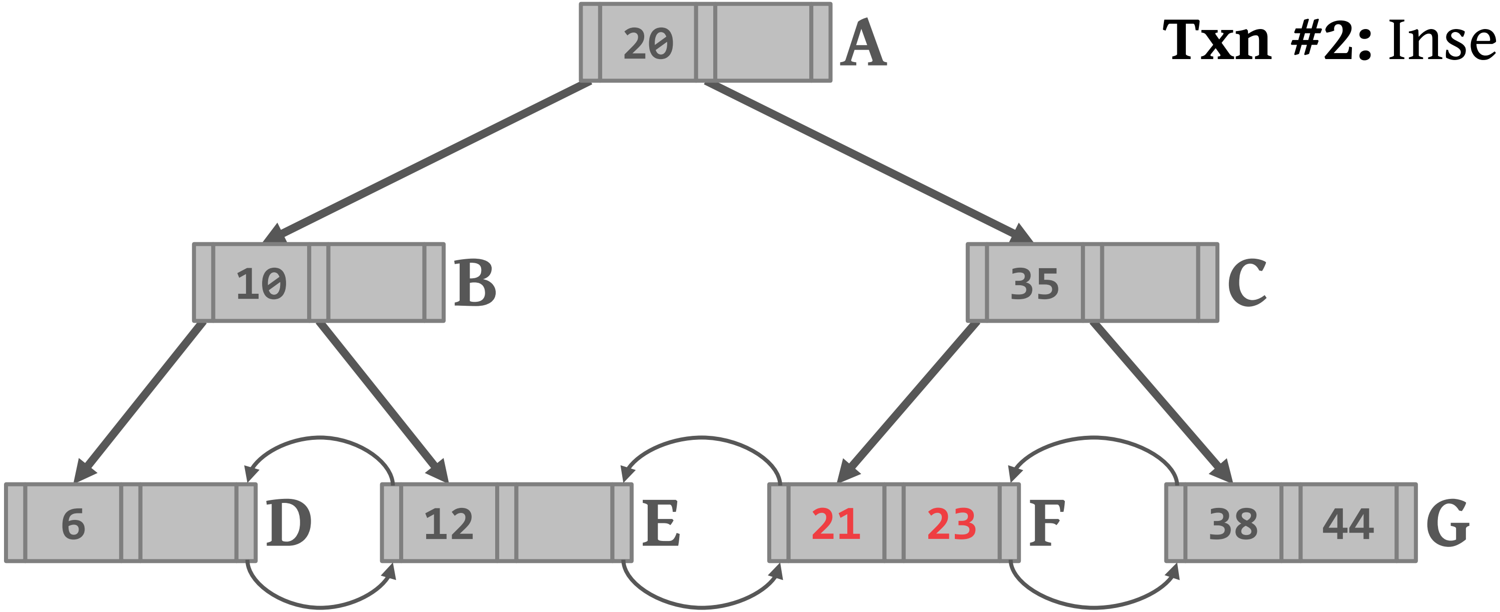
Txn #2: Insert 21



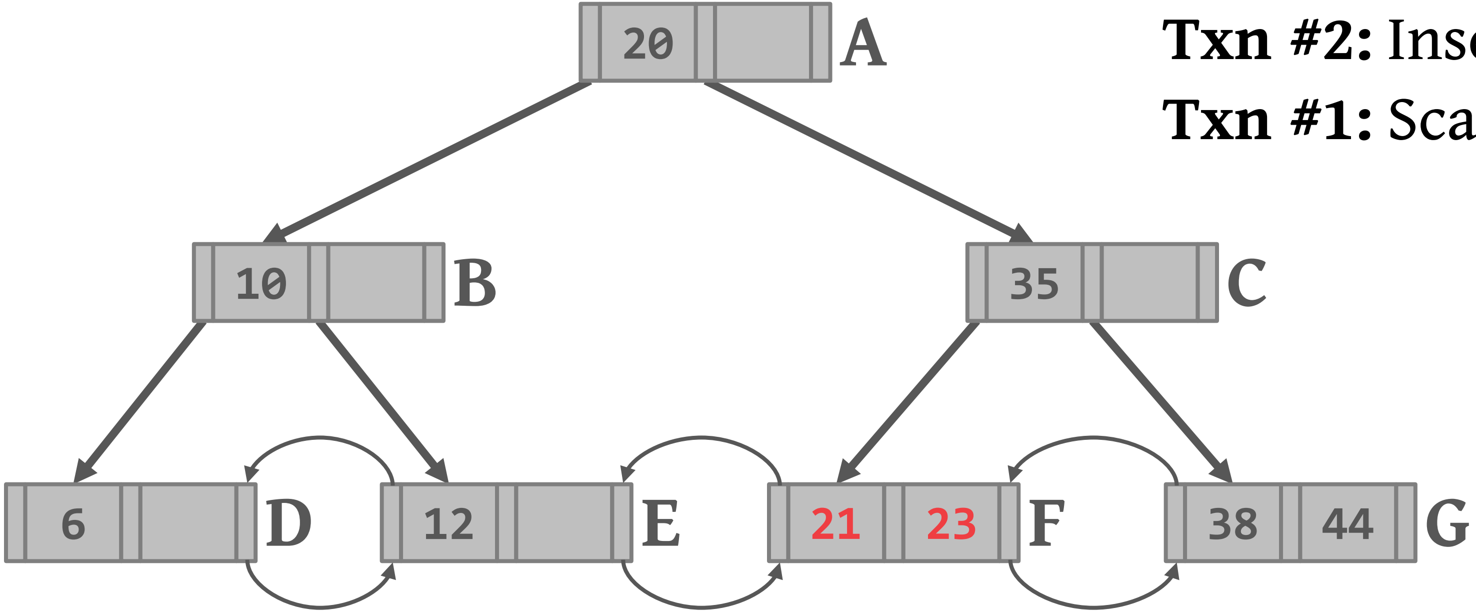
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]

Txn #2: Insert 21



PROBLEM SCENARIO #2

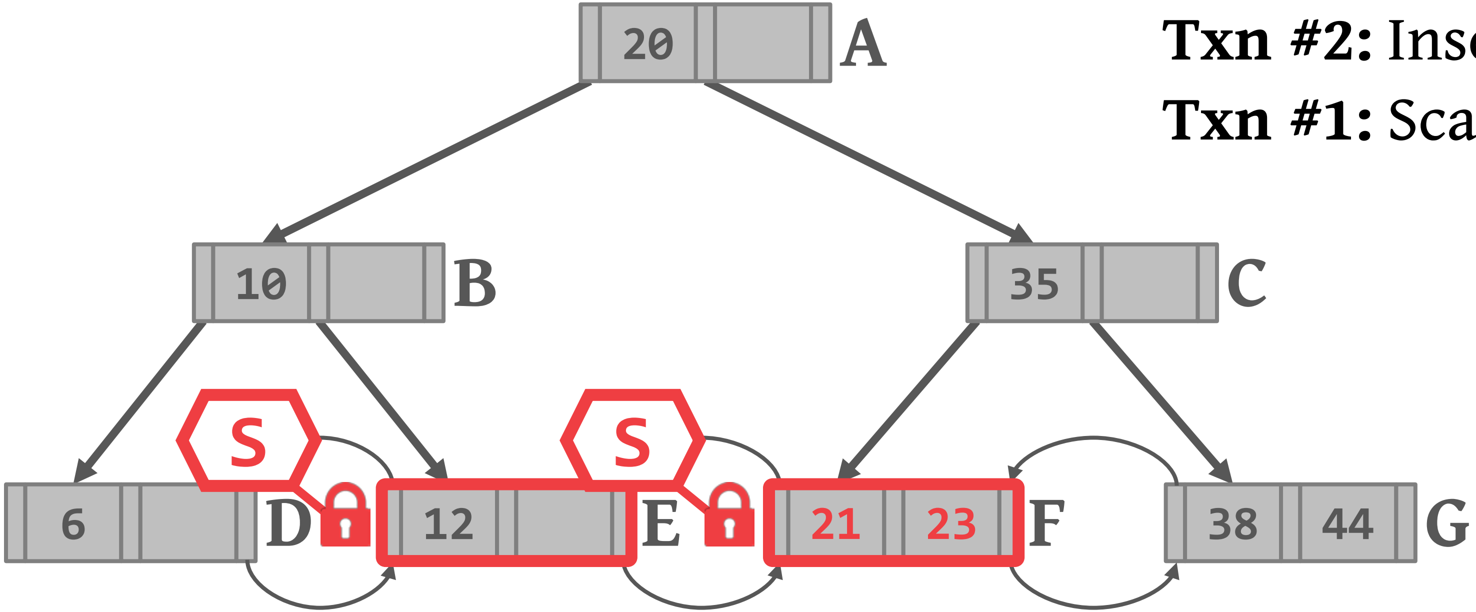


Txn #1: Scan [12, 23]

Txn #2: Insert 21

Txn #1: Scan [12, 23]

PROBLEM SCENARIO #2



Txn #1: Scan [12, 23]

Txn #2: Insert 21

Txn #1: Scan [12, 23]

LOCKS VS. LATCHES

Locks

→ Protects the index's logical contents from other txns.

→ Held for txn duration.

→ Need to be able to rollback changes.

Latches

→ Protects the critical sections of the index's internal data structure from other threads.

→ Held for operation duration.

→ Do not need to be able to rollback changes.

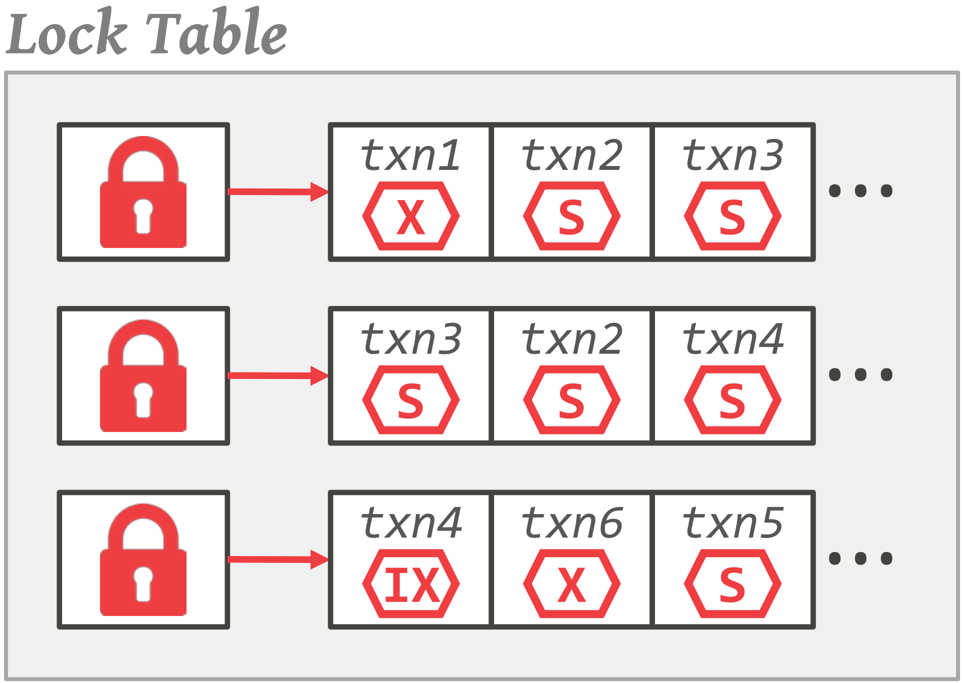
A SURVEY OF B-TREE LOCKING TECHNIQUES
TODS 2010

LOCKS VS. LATCHES

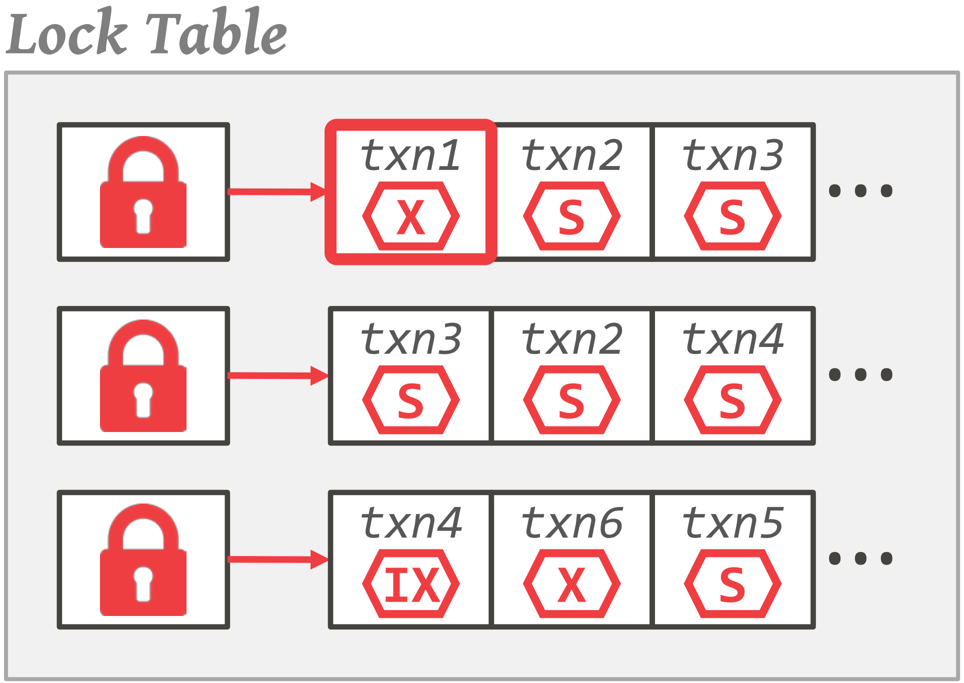
	Locks	*Latches*
Separate…	User transactions	Threads
Protect…	Database Contents	In-Memory Data Structures
During…	Entire Transactions	Critical Sections
Modes…	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
…by…	Waits-for, Timeout, Aborts	Coding Discipline
Kept in…	Lock Manager	Protected Data Structure

Source: Goetz Graefe

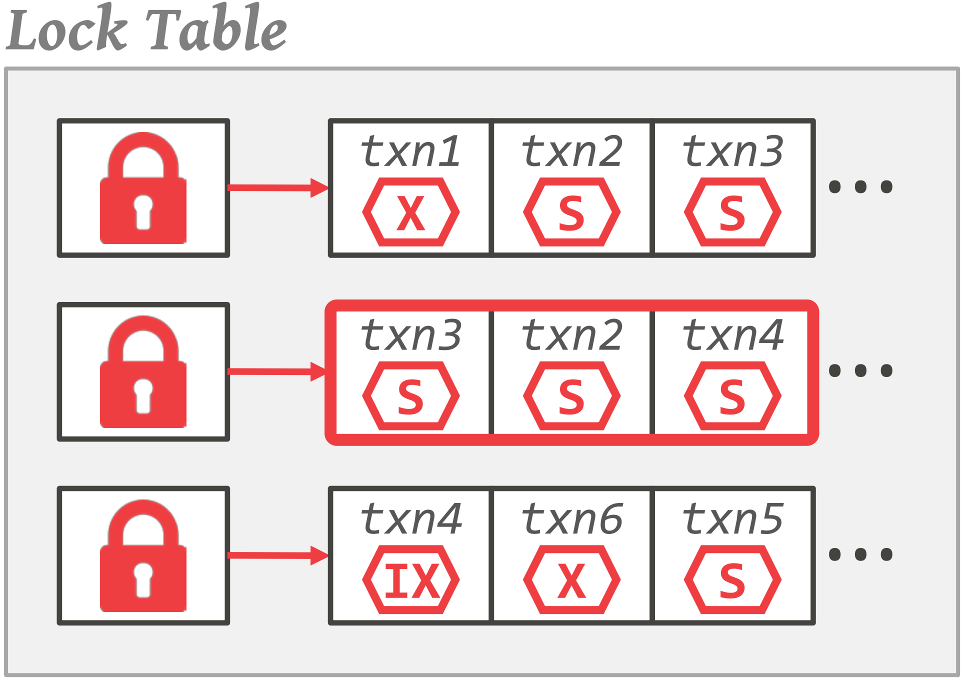
INDEX LOCKS


Lock Table
txn1
X
txn2
S
txn3
S
...
txn3
S
txn2
S
txn4
S
...
txn4
IX
txn6
X
txn5
S
...

INDEX LOCKS


Lock Table
txn1
X
txn2
S
txn3
S
...
txn3
S
txn2
S
txn4
S
...
txn4
IX
txn6
X
txn5
S
...

INDEX LOCKS


Lock Table
txn1
X
txn2
S
txn3
S
...
txn3
S
txn2
S
txn4
S
...
txn4
IX
txn6
X
txn5
S
...

LOCK-FREE INDEXES

Possibility #1: No Locks

→ Txns don't acquire locks to access/modify database.
→ Still have to use latches to install updates.

Possibility #2: No Latches

→ Use multi-versioning inside of the index. Swap pointers using atomic updates to install updates.
→ Still have to use locks to validate txns.

INDEX LOCKING

Predicate Locks
Key-Value Locks
Gap Locks
Key-Range Locks
Hierarchical Locking

PREDICATE LOCKS

Proposed locking scheme from System R.

→ Shared lock on the predicate in a **WHERE** clause of a **SELECT** query.

→ Exclusive lock on the predicate in a **WHERE** clause of any **UPDATE**, **INSERT** or **DELETE** query.

Never implemented in any system.

THE NOTIONS OF CONSISTENCY AND PREDICATE LOCKS IN A DATABASE SYSTEM
CACM 1976

PREDICATE LOCKS

```
SELECT SUM(balance)
  FROM account
 WHERE name = ‘Tupac’
```

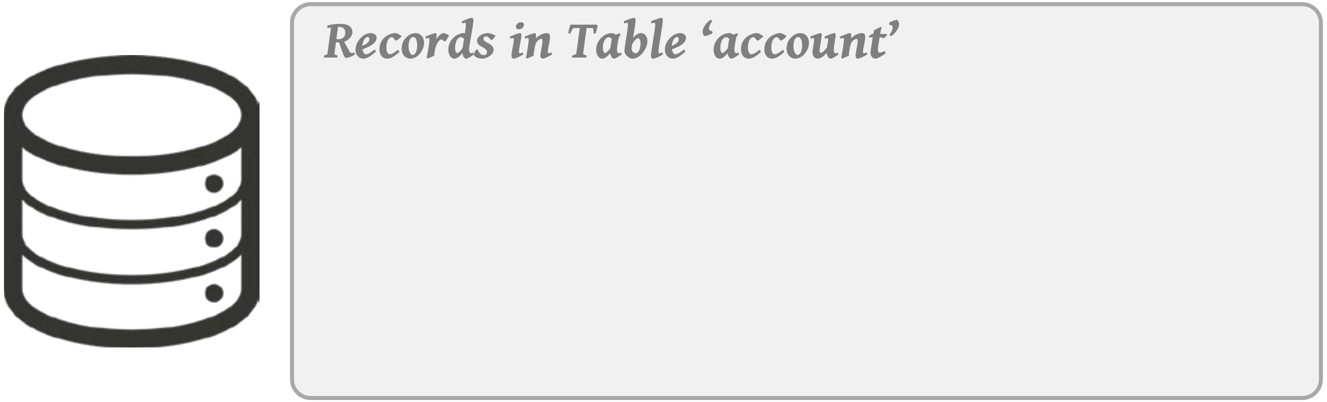
```
INSERT INTO account
 (name, balance)
 VALUES (‘Tupac’, 100);
```

Records in Table ‘account’

PREDICATE LOCKS

```
SELECT SUM(balance)
  FROM account
 WHERE name = ‘Tupac’
```

```
INSERT INTO account
 (name, balance)
 VALUES (‘Tupac’, 100);
```

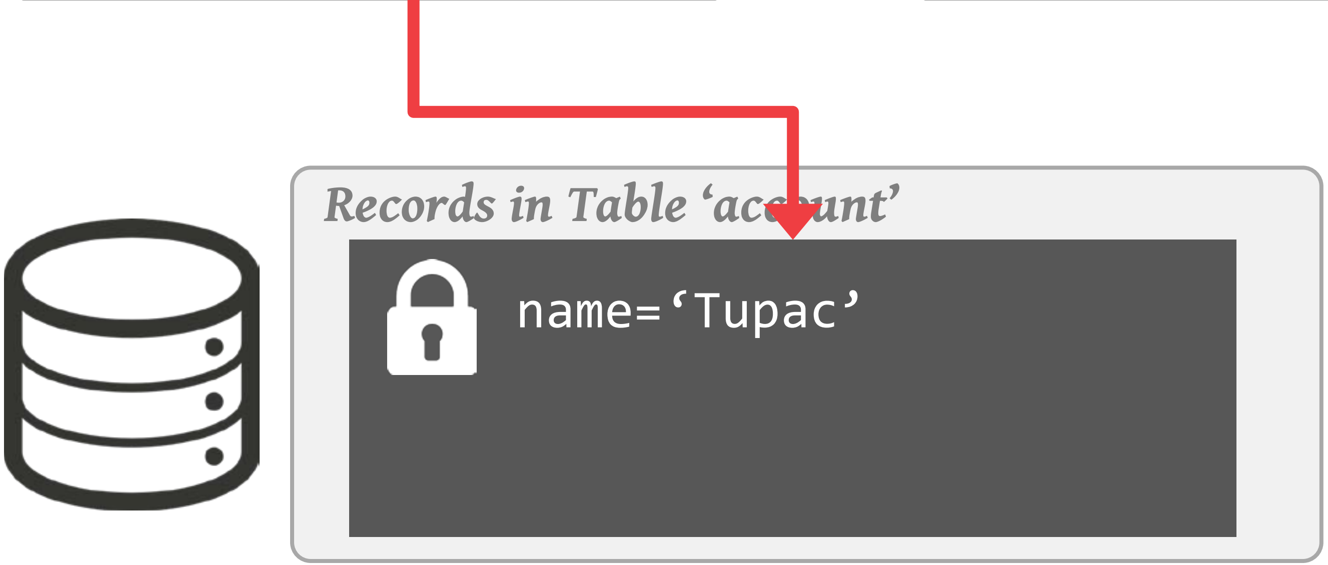


Records in Table ‘account’

PREDICATE LOCKS

```
SELECT SUM(balance)
  FROM account
 WHERE name = ‘Tupac’
```

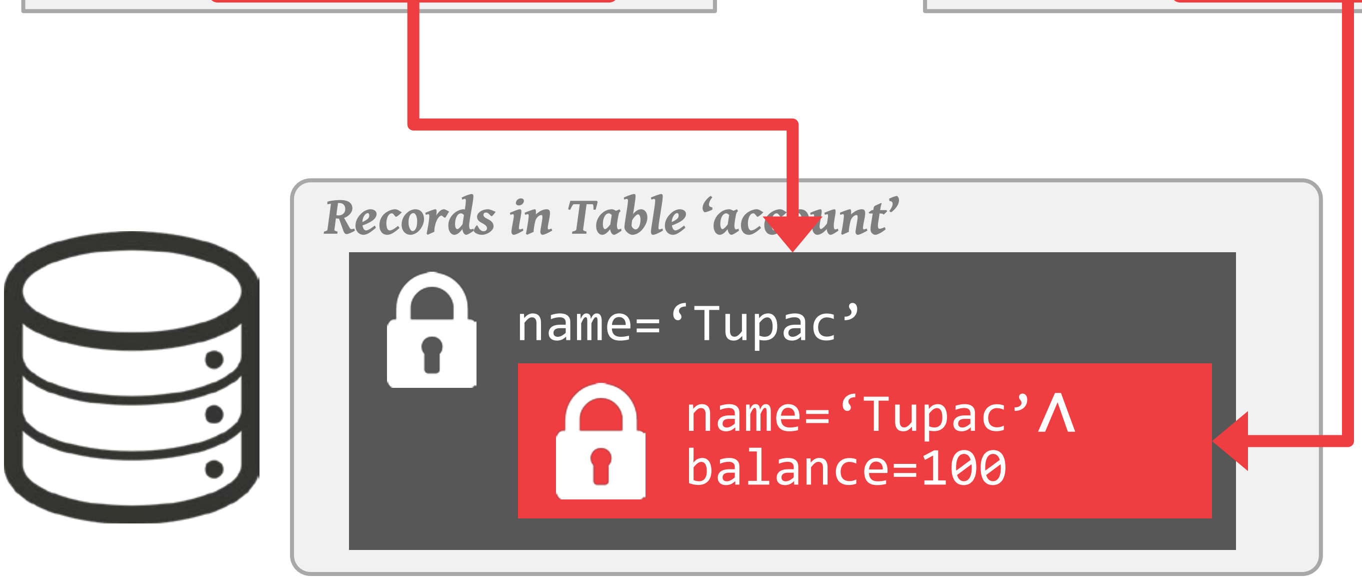
```
INSERT INTO account
 (name, balance)
 VALUES (‘Tupac’, 100);
```



PREDICATE LOCKS

```
SELECT SUM(balance)
  FROM account
 WHERE name = ‘Tupac’
```

```
INSERT INTO account
 (name, balance)
 VALUES (‘Tupac’, 100);
```

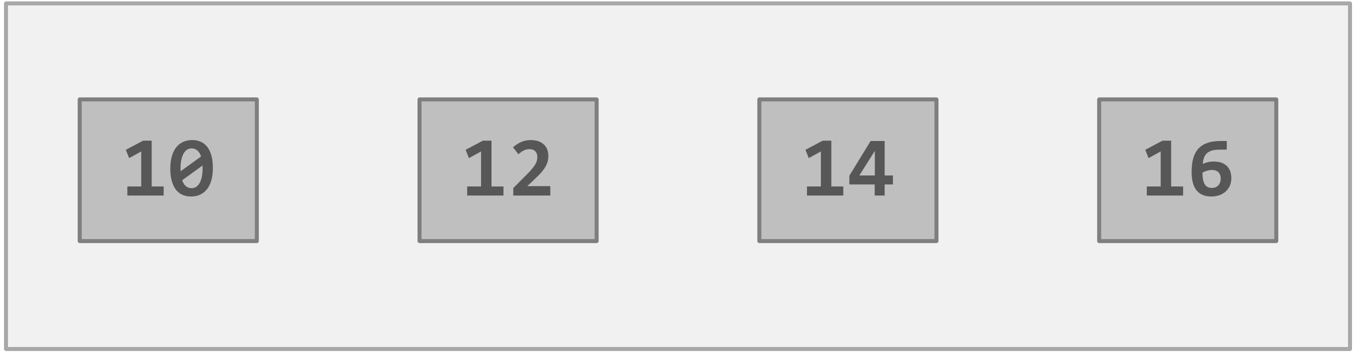


KEY-VALUE LOCKS

Locks that cover a single key value.

Need “virtual keys” for non-existent values.

B+Tree Leaf Node

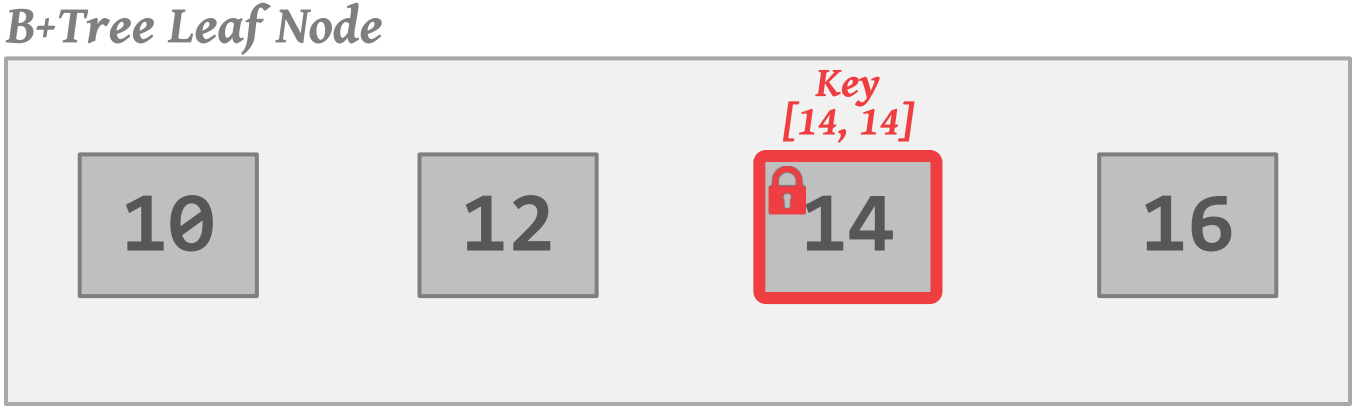


10 12 14 16

KEY-VALUE LOCKS

Locks that cover a single key value.

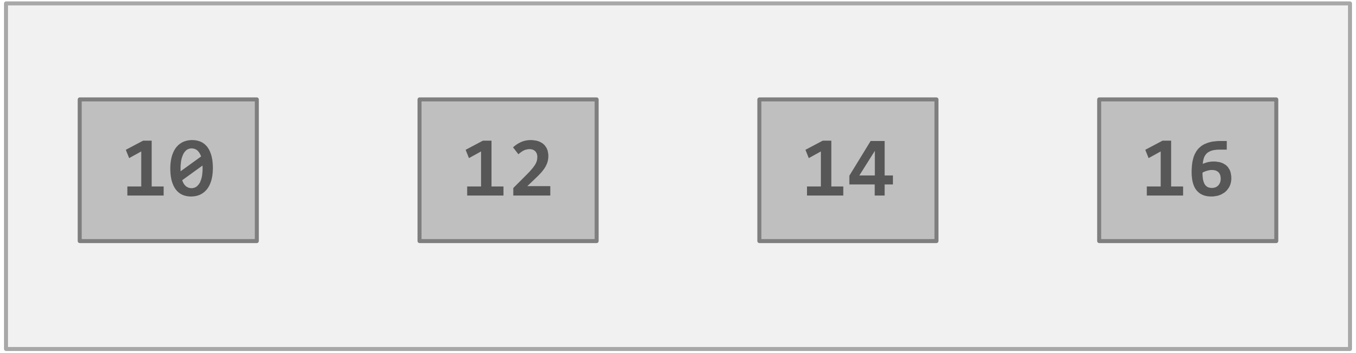
Need "virtual keys" for non-existent values.



GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

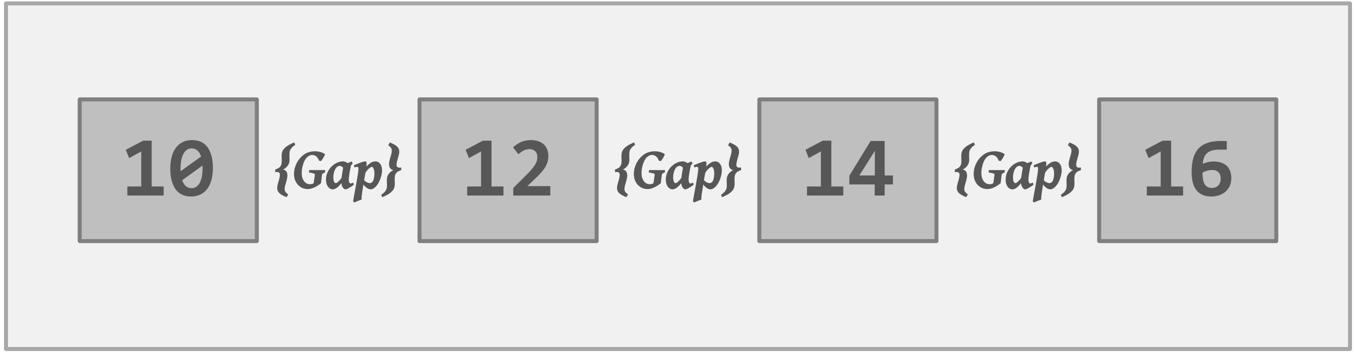
B+Tree Leaf Node



GAP LOCKS

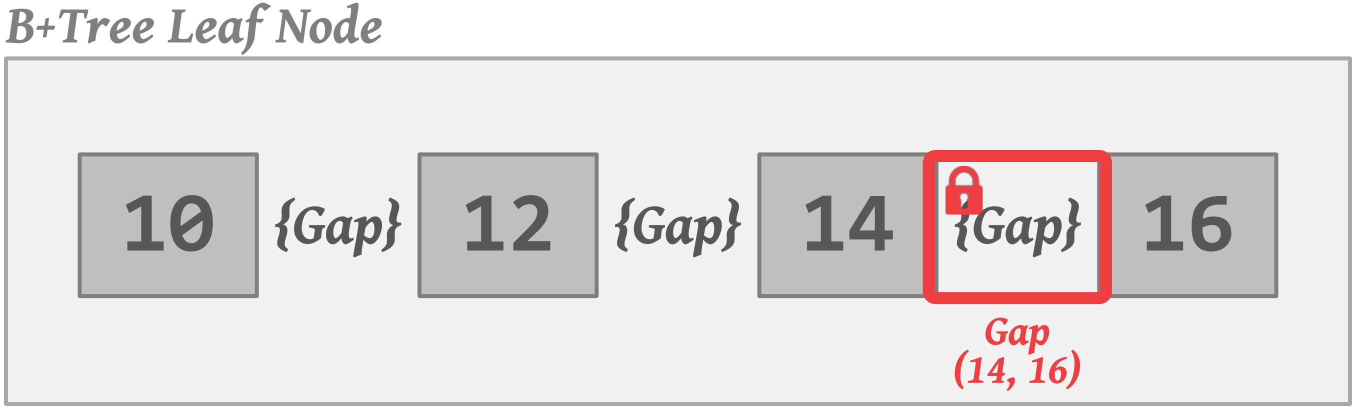
Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

B+Tree Leaf Node



GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.



KEY-RANGE LOCKS

A txn takes locks on ranges in the key space.

→ Each range is from one key that appears in the relation, to the next that appears.

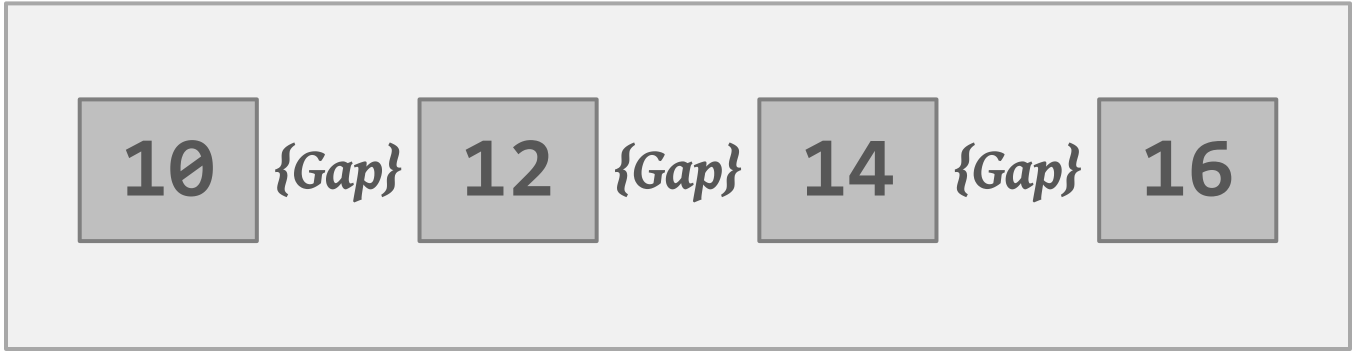
→ Define lock modes so conflict table will capture commutativity of the operations available.

KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.

→ Need "virtual keys" for artificial values (infinity)

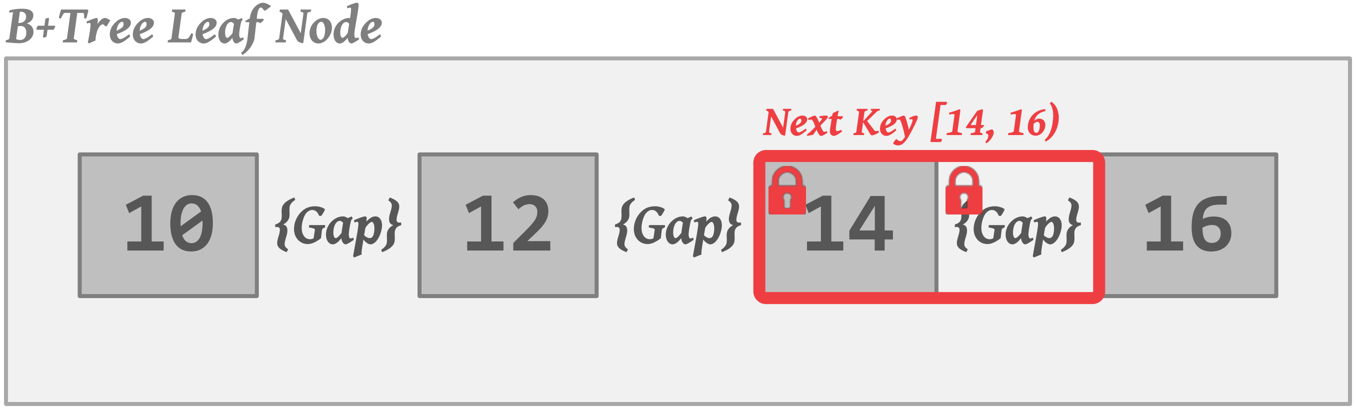
B+Tree Leaf Node



KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.

→ Need "virtual keys" for artificial values (infinity)

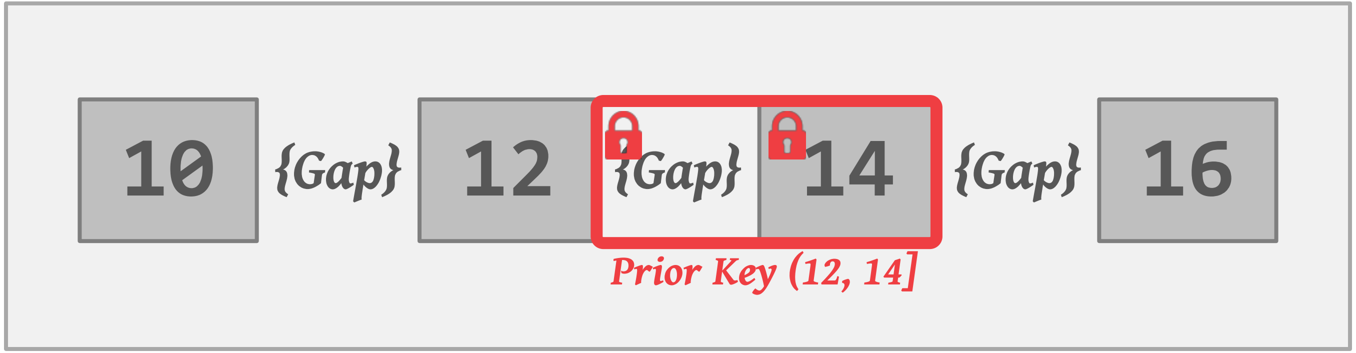


KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.

→ Need "virtual keys" for artificial values (infinity)

B+Tree Leaf Node

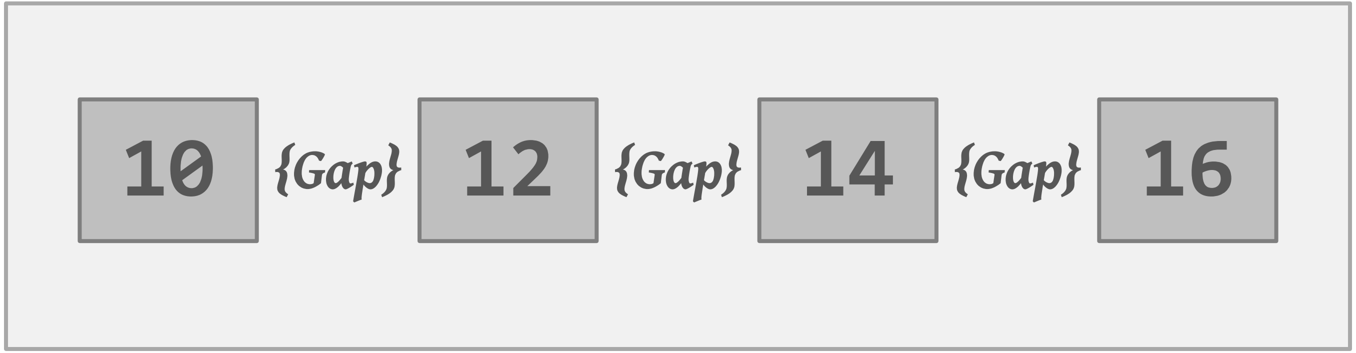


HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

→ Reduces the number of visits to lock manager.

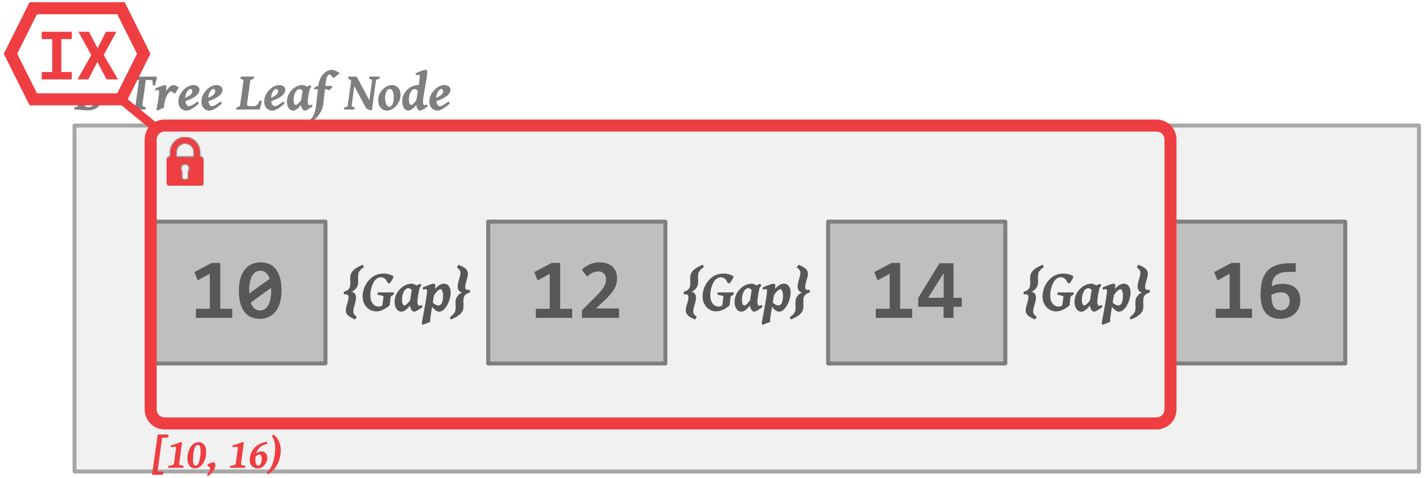
B+Tree Leaf Node



HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

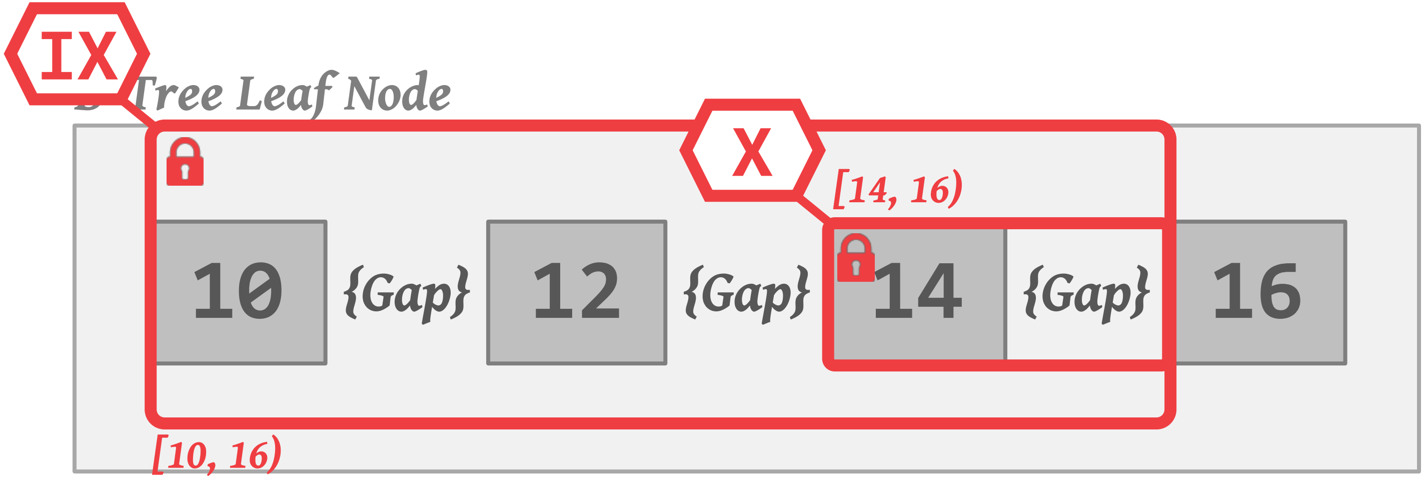
→ Reduces the number of visits to lock manager.



HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

→ Reduces the number of visits to lock manager.



PARTING THOUGHTS

Hierarchical locking essentially provides predicate locking without complications.
→ Index locking occurs only in the leaf nodes.
→ Latching is to ensure consistent data structure.

Just like concurrency control schemes, research on fast indexes is hot again.

PRISON TATTOOS

Some of you are going to end up in prison.

→ This is just the nature of the database game.

Part of surviving prison is being able to navigate and avoid the various factions.

TEAR DROP

THREE DOTS

FIVE DOTS

MARA SALVATRUCHA GANG (MS13)

ARYAN BROTHERHOOD

NEXT CLASS

Bw-Tree (Hekaton)

Concurrent Skip Lists (MemSQL)

ART Index (HyPer)