AB-tree: Index for Concurrent Random Sampling and Updates

Zhuoyue Zhao
University at Buffalo

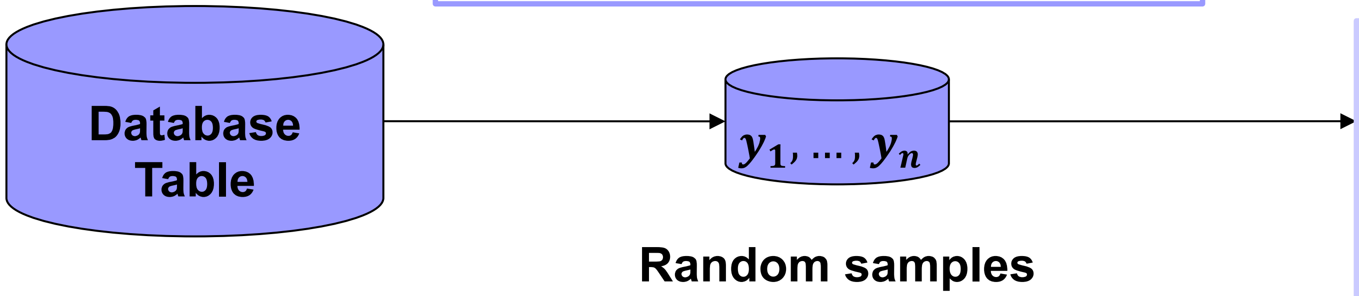
Dong Xie
Pennsylvania State University

Feifei Li
Alibaba

Motivation

- Approximate Query Processing (AQP) uses random samples
  - to provide fast and approximate answers with error guarantees
  - existing solutions often make trade-off between
    - efficient online updates and
    - low response time

```
SELECT SUM(y)
FROM A
WHERE x >= 5 AND x <= 10
```



Unbiased estimator

$$\tilde{Y} = \frac{\sum_{i \in [n]} y_i / p_i}{n}$$

Confidence interval ε, δ

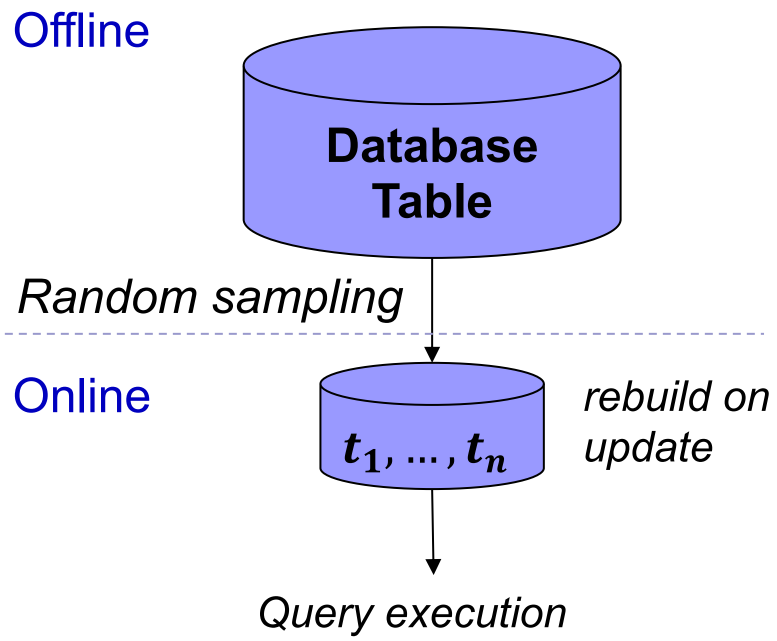
$$Pr\left(\left|Y - \tilde{Y}\right| \le \varepsilon\right) \ge 1 - \delta$$

Motivation

- How do existing AQP systems perform random sampling?

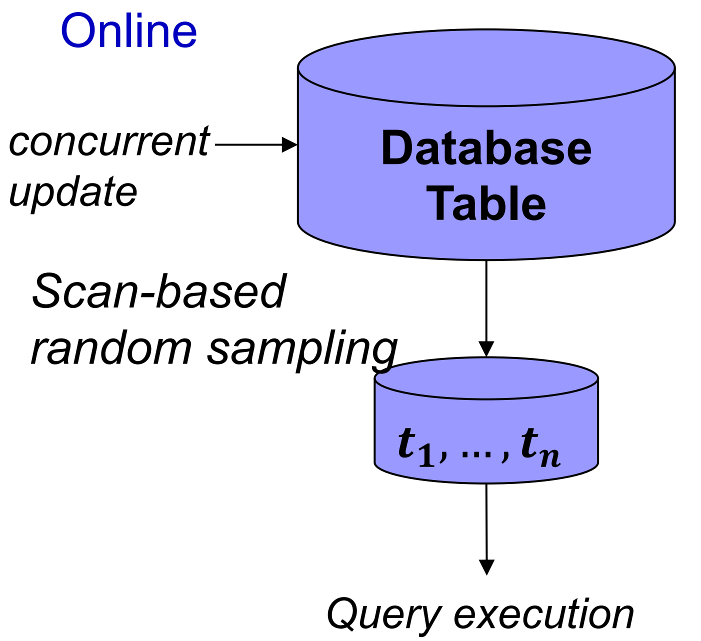
Offline sampling

- ✓ Fast query: linear to sample size
- × Stale data and needs rebuild
- × Slow and delayed batch update



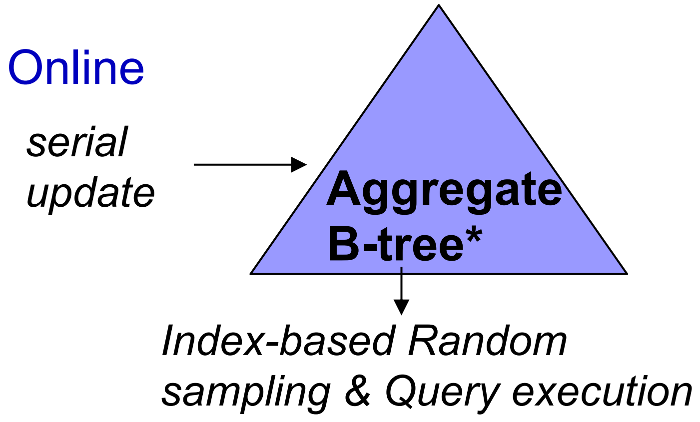
Online Scan-based Sampling

- × Slow query: linear to data size
- ✓ Query over latest updates
- ✓ Fast concurrent update



Online Index-based Sampling

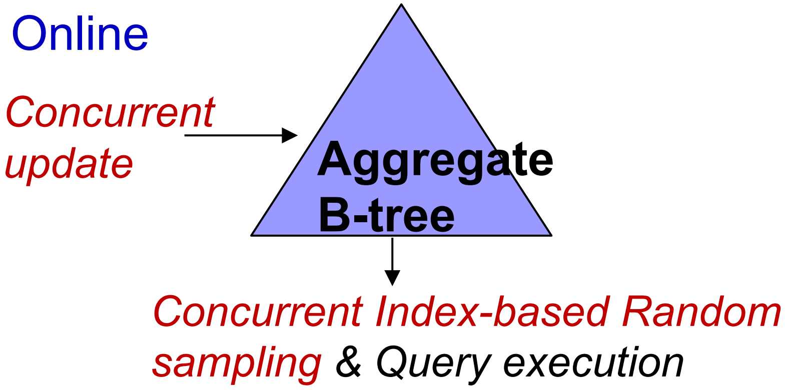
- ✓ Fast query: linear to sample size
- ✓ Query over latest updates
- × Slow serial update



*aka Ranked B-Tree, see [Frank Olken's PhD thesis, 1993]

Goals

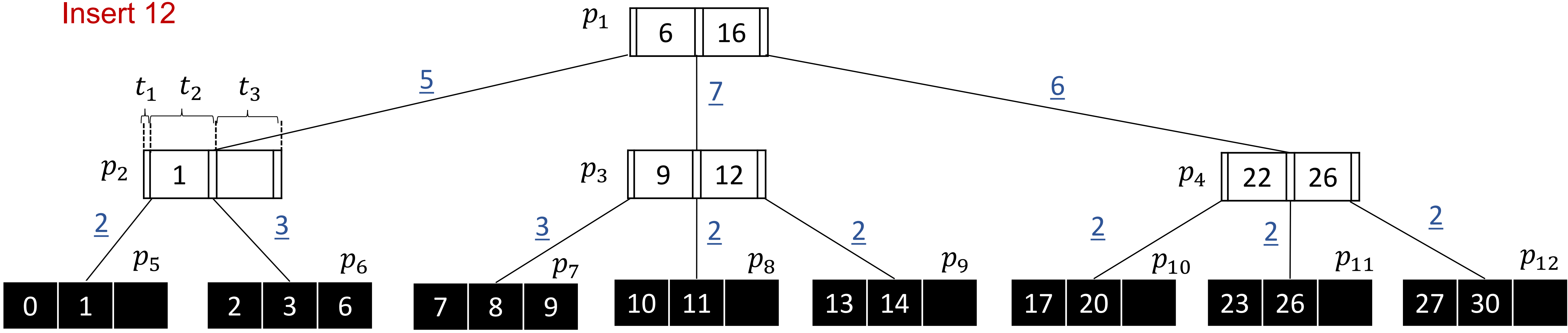
- Design an index structure that supports
  - ✓ Fast AQP query: sampling scales (almost) linear to sample size
  - ✓ Query over latest updates
  - ✓ Fast concurrent update



Example: aggregate B-tree with uniform weights

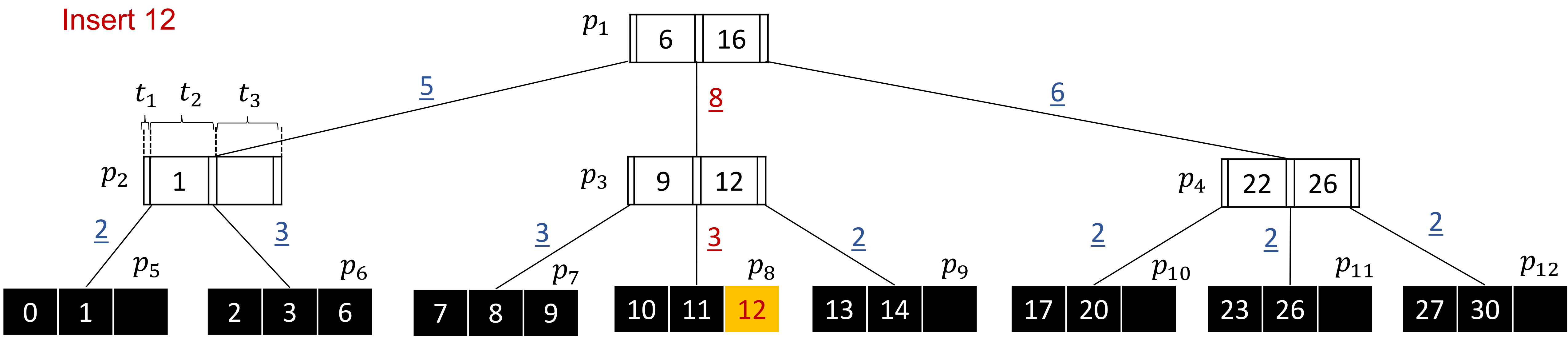
- Aggregate B-tree
  - Maintains sub-tree weights w_c along with page pointer c
    - w_c is the sum of weights in the sub-tree
  - Starting from root, randomly descend into sub-trees with probability $\propto w_c$
    - It can be shown the leaf tuple sampled has a probability proportional to its weight

Insert 12



Example: aggregate B-tree with uniform weights

- Aggregate B-tree
  - Maintains sub-tree weights w_c along with page pointer c
    - w_c is the sum of weights in the sub-tree
  - Starting from root, randomly descend into sub-trees with probability $\propto w_c$
    - It can be shown the leaf tuple sampled has a probability proportional to its weight
  - Weight updates must be applied atomically along a tree path from root to leaf where insertion happens



Baseline and our solution

- Baseline: X-latch tree path for each update
  - × Every update blocks every other thread
  - × Sampling and update throughput drops significantly under heavy update workload

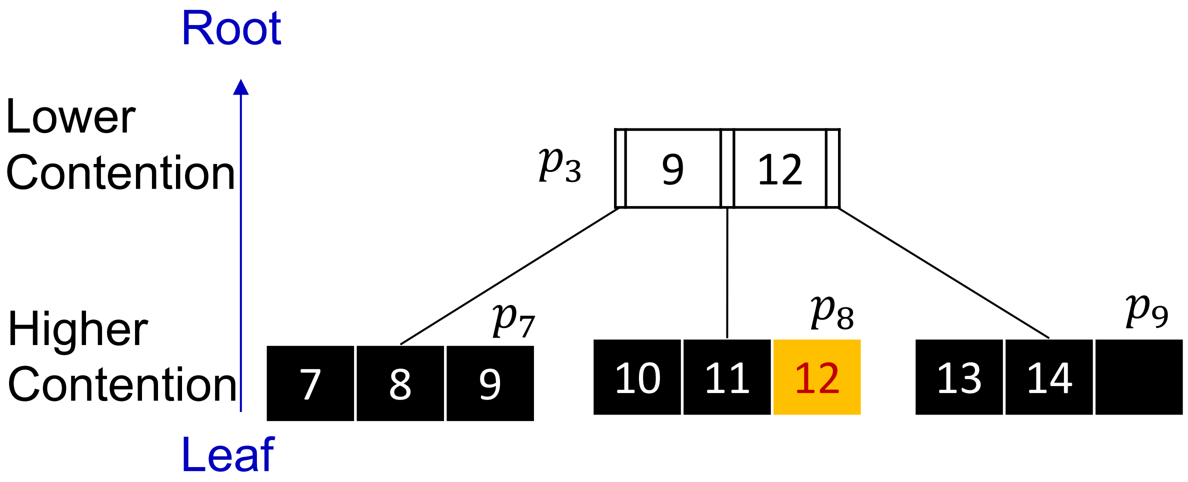
- Challenges: how to ensure highly concurrent sampling and update without impacting the correctness of random sampling

- Our solution: AB-tree
  - – based on B-link tree implementation in PostgreSQL 13
  - – available here: https://github.com/zzy7896321/abtree_public

Challenge 1: Non-blocking Weight Updates

- Different contention pattern than conventional concurrent B-trees

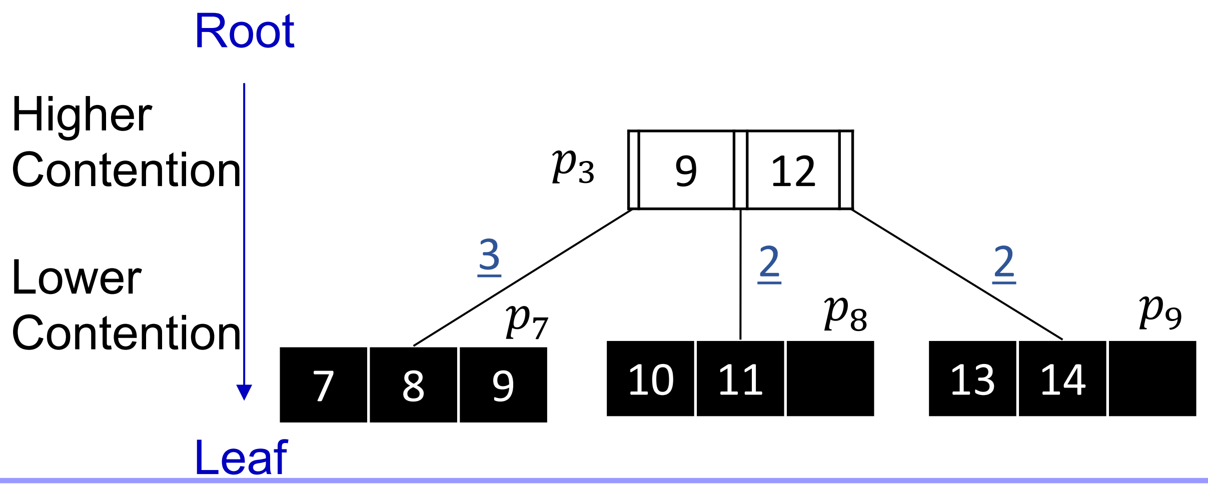
Regular B-tree



- SMO often happens around leaf
- Internal pages rarely updated

Conventional wisdom:
Localize contention to one or two pages at a time using atomic Compare-And-Swap (CAS) or X-latches.

Aggregate B-tree



- Internal pages have higher contention for weight updates
- Root page is always contended in any update

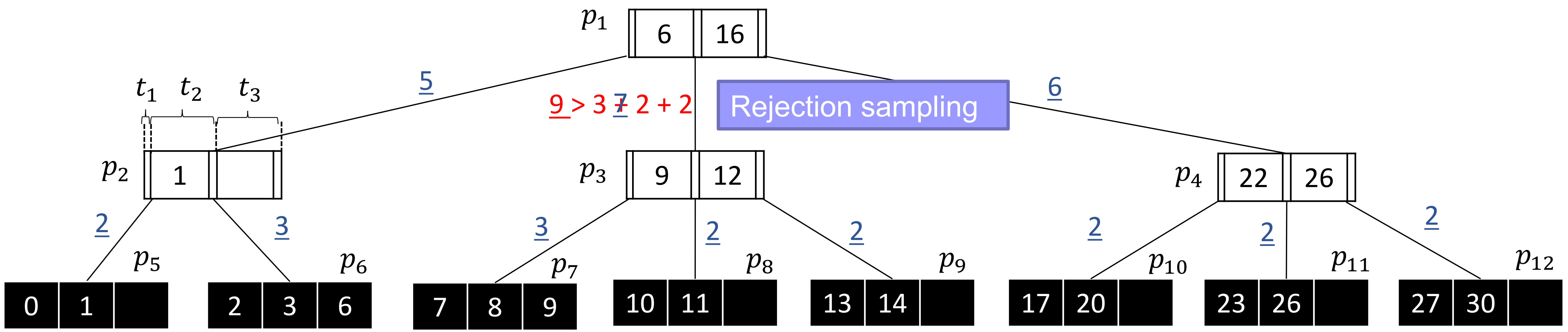
Can we update weights without X-latching the entire tree path?

- Yes, use CAS with S-latch one page at a time!
  - S-latch guarantees no concurrent SMO while CAS is applied
  - Weight updater do not block others
  - Correctness of sampling?

Challenge 2: Ensuring Consistent Weights for Sampling

- Consistent weights needed for sampling purpose
  - perform rejection sampling as in [Olken'93]

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

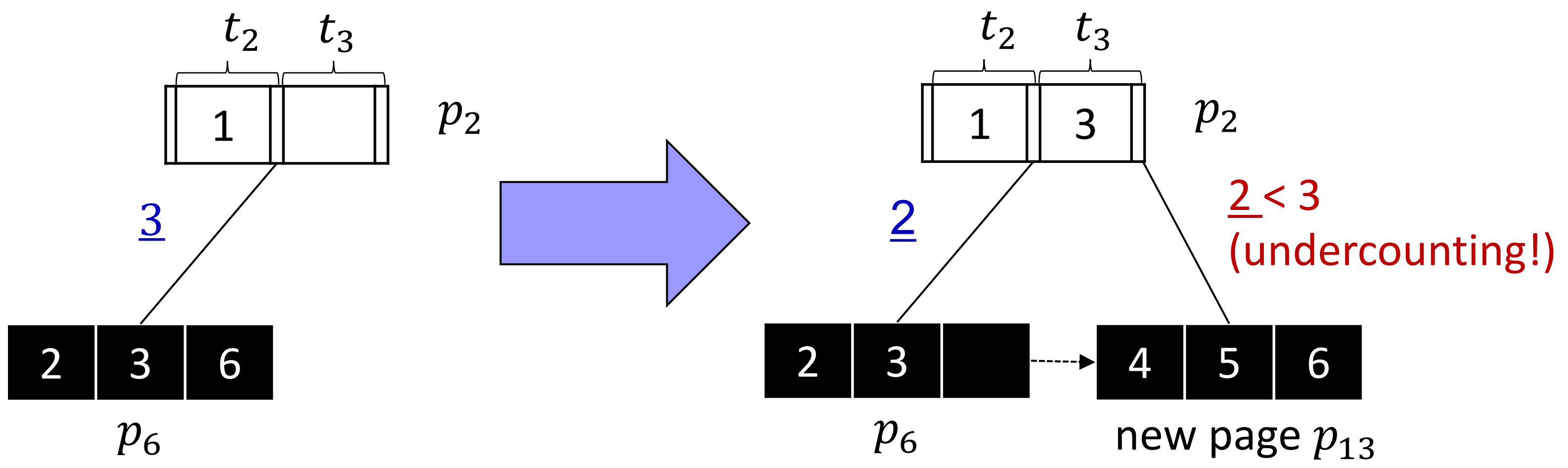


Challenge 2: consistent weights for sampling (cont'd)

- Consistent weights needed for sampling purpose
  - perform rejection sampling as in [Olken'93]
- However, we cannot update weight in parent before insertion
  - Concurrent Structural Modification Operation (SMO) *may undo* the change

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:
(1) T_1 increments $\widetilde{w}_{t_2}$
(2) T_2 increments $\widetilde{w}_{t_2}$
(3) T_2 splits p_6 and inserts t'
(4) T_1 inserts t



Challenge 2: Ensuring Consistent Weights for Sampling

- Consistent weights needed for sampling purpose
  - perform rejection sampling as in [Olken'93]
- However, we cannot update weight in parent before insertion
  - Concurrent Structural Modification Operation (SMO) *may undo* the change

- Solution: two-pass insertion
  - Pass 1: regular key insertion
    - assign zero weight to new key
  - Pass 2: descend in the tree again and modify weights
    - redo weight modification on certain pages in case of concurrent SMO
    - use page and tuple update counters to detect concurrent SMO -- see paper for details

Challenge 3: Sampling under MVCC

- Sampling under an old snapshot with MVCC could suffer from “live version bloat”
  - Many live versions of tuples are
    - not visible to that sampling thread
    - but are physically present in the index
    - → high rejections rates → decreased sampling throughput

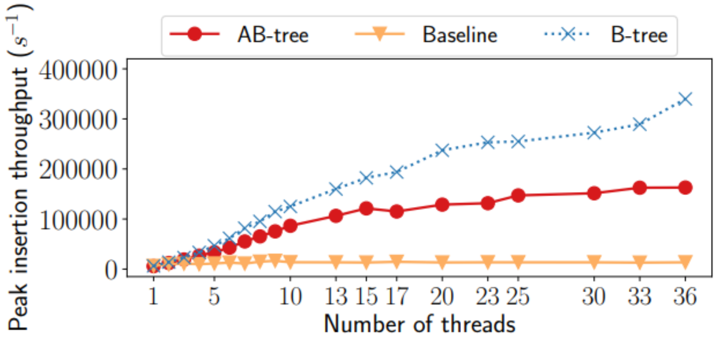
- Solution: build an in-memory multi-version weight store to allow
  - Querying upper bound of weights under an old snapshot
    - Tight enough for minimizing rejection due to live version bloat
  - No logging/persistency required
    - Only queries by active transactions
    - Old snapshots do not live across crashes
  - Details in the paper

Experiments

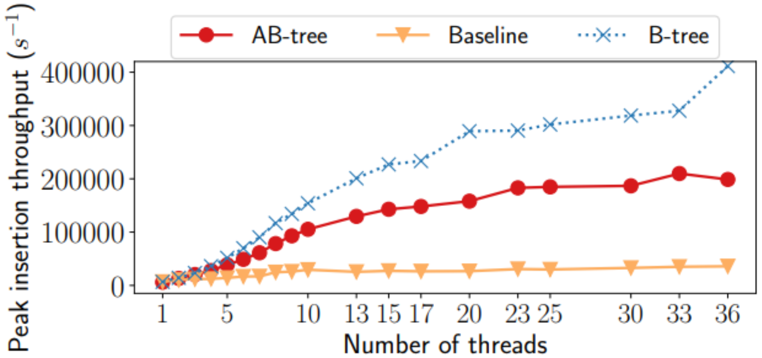
- A two-column table $A(x, y)$, AB-tree/baseline built on y
  - Fan-out is up to about 300, height = 4
  - Preloaded with 1 billion random tuples
- Runs random insertions/random sampling/mixed workload

```
SELECT COUNT(*) FROM A TABLESAMPLE SWR(?);         -- AB-tree
SELECT COUNT(*) FROM A TABLESAMPLE BERNOULLI(?);   -- Baseline heap scan
INSERT INTO A VALUES (?, ?);
```

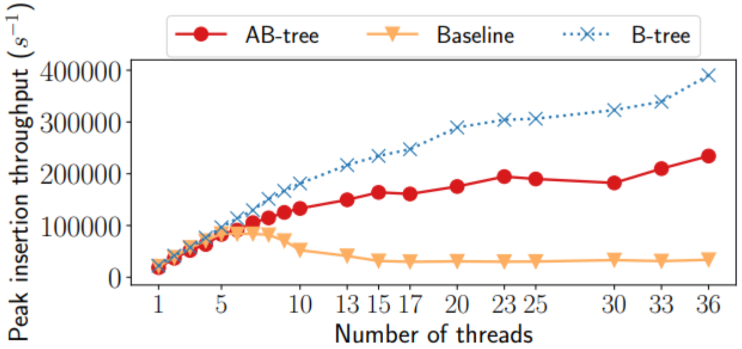
Scalability



(a) Small buffer (128MB)



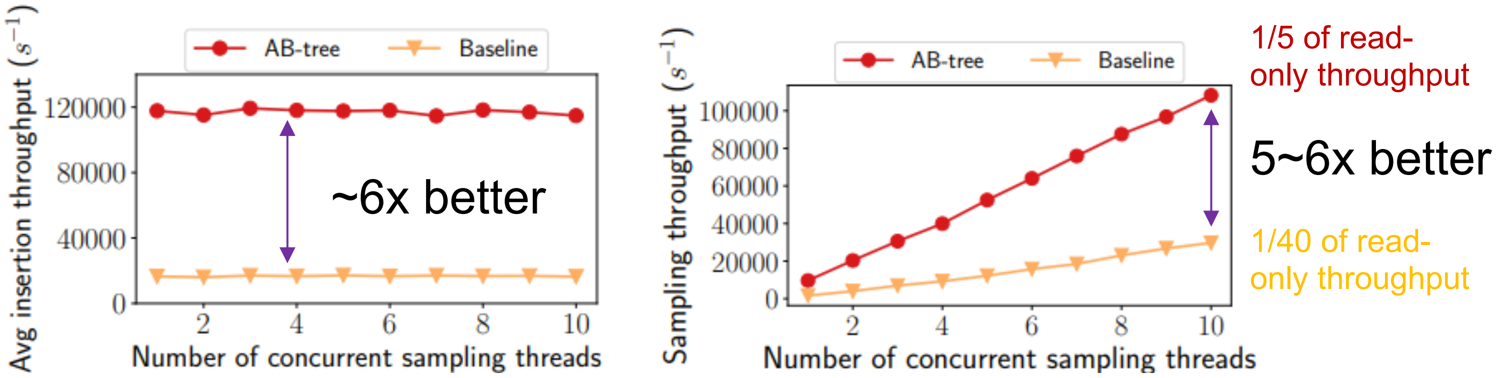
(b) Large buffer (32GB)



(c) In-memory
(32 GB, simulated with same seed)

B-tree is the original B-link tree without aggregates in PostgreSQL. Its insertion throughput is an *upper bound*.

Read-write workload



Read-write workload with 10 insertion threads and varying # of sampling threads

Summary

- We designed AB-tree, an aggregate B-tree that supports efficient concurrent random sampling and updates

- Future direction
  - Improve scalability to many-core systems
  - Use AB-tree to enable HTAP use cases with AQP

Thank you!
Q& A

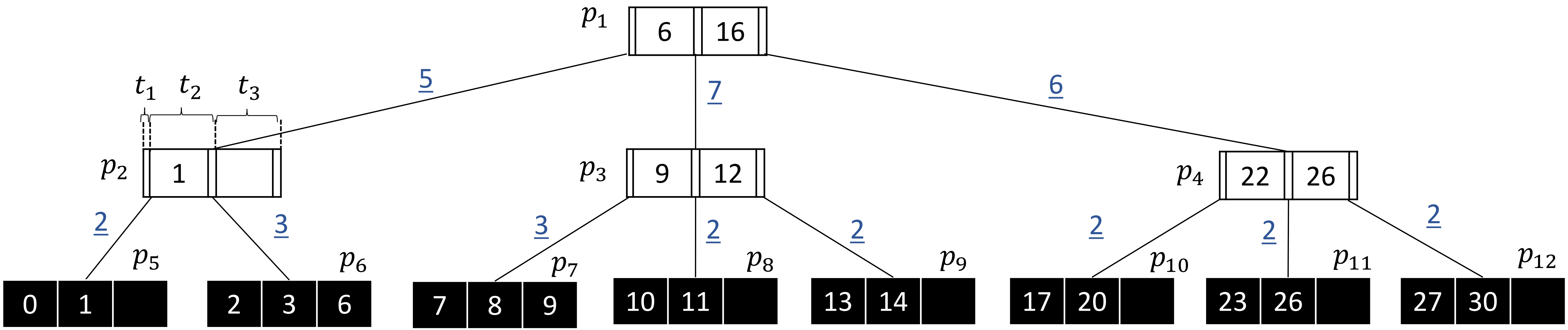
Existing Random Sampling Access Methods

- Sampling has been supported as TABLESAMPLE since SQL 2003
  - × Scan-based: scales linearly to data size (slow!)
  - × Limited support for random sampling operators needed by AQP
    - System/Block sample: sampling pages instead of tuples (non-independent/non-uniform)
    - Bernoulli sample: flipping a biased coin (no control on sample size and slow)
    - No support for weighted sampling
  - ✓ Works seamlessly with concurrent updates
    - standard concurrency control mechanism applies

```
SELECT SUM(y) / 0.01
FROM A TABLESAMPLE BERNOULLI(1)
WHERE X >= 5 AND X <= 10
```

Existing Random Sampling Access Methods

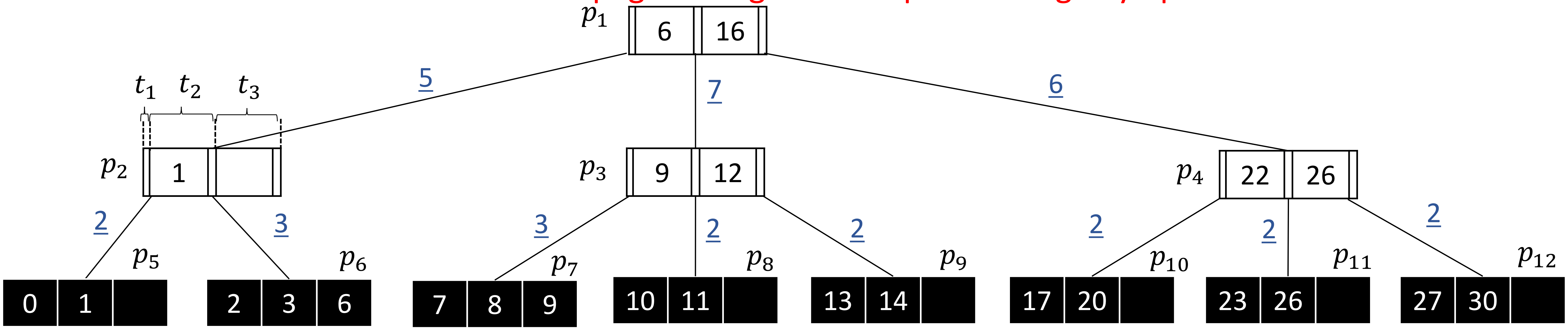
- Index structure for random sampling
  - Aggregate B-tree (aka Ranked B-Tree, see Frank Olken's PhD thesis, 1993)
    - Maintains sub-tree weights w_c along with page pointer c
    - Randomly traverse sub-trees with probability $\propto w_c$
  - ✓ $O(\log_B N)$ time per sample (fast)
  - ✓ Supports uniform and weighted samples
  - × Unable to perform concurrent updates



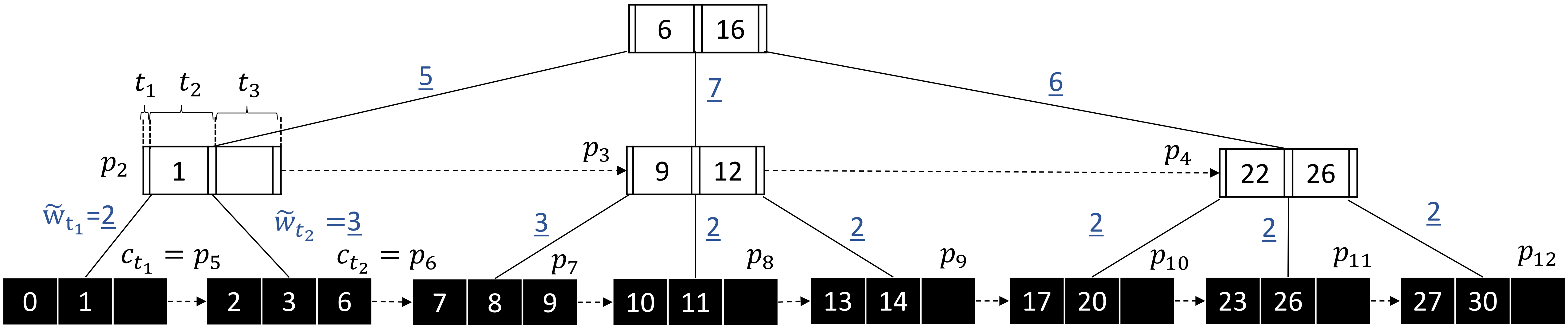
Aggregate B-tree Indexes for Random Sampling

- Aggregate B-tree is more efficient when taking a small sample of size m from N tuples
  - $O(m\lceil\log_B N\rceil)$ time, B is the fan-out
  - In contrast, the standard SQL tablesample Bernoulli operator requires $O(N)$ time

- Question: how to enable concurrent updates and sampling in the same aggregate B-tree?
  - Three challenges from correctly maintaining and querying the aggregated weights
  - Naïve solution: x-lock all the pages along a search path during any update



Notations



6
16
5
7
6
t_1 t_2 t_3
p_2
1
p_3
9
12
p_4
22
26
$\widetilde{w}_{t_1}$=2
$\widetilde{w}_{t_2}$ =3
3
2
2
2
2
2
$c_{t_1} = p_5$
$c_{t_2} = p_6$
p_7
p_8
p_9
p_{10}
p_{11}
p_{12}
0
1
2
3
6
7
8
9
10
11
13
14
17
20
23
26
27
30

Our solution

- Our solution: *AB-tree*
  - Based on the B-link tree [Lehman & Yao, TODS'81] implementation in PostgreSQL
  - We focus on the insertions (deletions are done in bulks and in background)
    - Two-pass insertions: updating weights after inserting the leaf tuples
    - Only shared-latch pages when updating weights → allows higher concurrency on root
      - Use Compare-And-Swap or Fetch-And-Add to update the aggregate weights and page LSN
  - Multi-version weight store
    - Allows a sampling thread to query an upper bound of the stored weight at an old snapshot
    - Avoids rejections due to live version bloat

Challenge 2: consistent weights for sampling (cont'd)

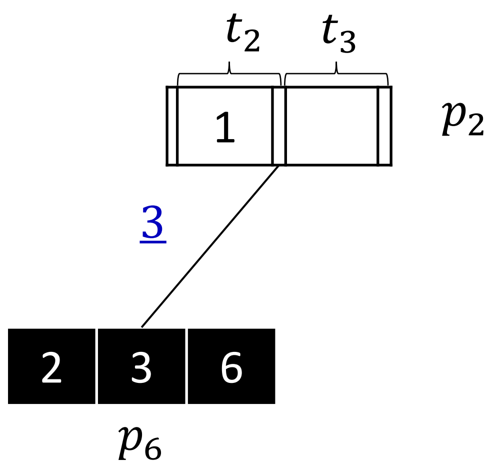
- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 1: updating weights before leaf insertion → undercounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:



Challenge 2: consistent weights for sampling (cont'd)

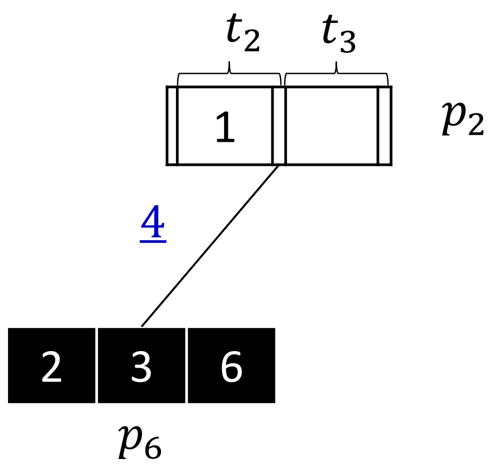
- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 1: updating weights before leaf insertion → undercounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:
(1) T_1 increments $\widetilde{w}_{t_2}$



Challenge 2: consistent weights for sampling (cont'd)

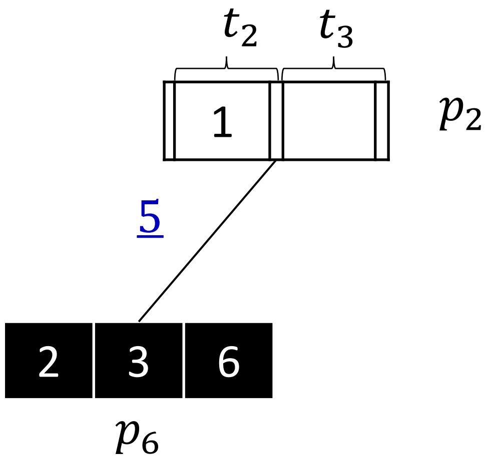
- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 1: updating weights before leaf insertion → undercounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:
(1) T_1 increments $\widetilde{w}_{t_2}$
(2) T_2 increments $\widetilde{w}_{t_2}$



Challenge 2: consistent weights for sampling (cont'd)

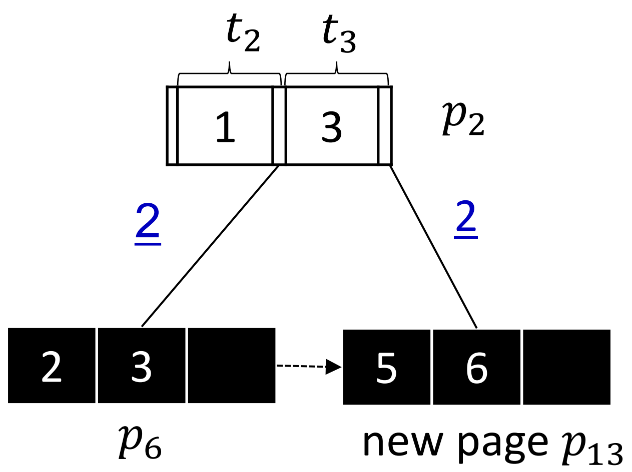
- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 1: updating weights before leaf insertion → undercounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:
(1) T_1 increments $\widetilde{w}_{t_2}$
(2) T_2 increments $\widetilde{w}_{t_2}$
(3) T_2 splits p_6 and inserts t'



Challenge 2: consistent weights for sampling (cont'd)

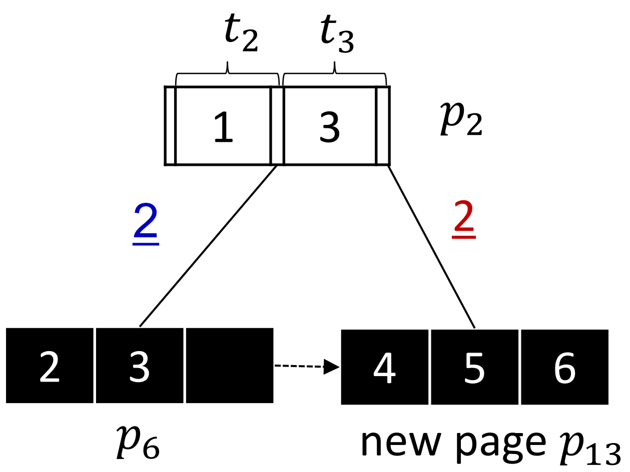
- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 1: updating weights before leaf insertion → undercounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:
(1) T_1 increments $\widetilde{w}_{t_2}$
(2) T_2 increments $\widetilde{w}_{t_2}$
(3) T_2 splits p_6 and inserts t'
(4) T_1 inserts t



Challenge 2: consistent weights for sampling (cont'd)

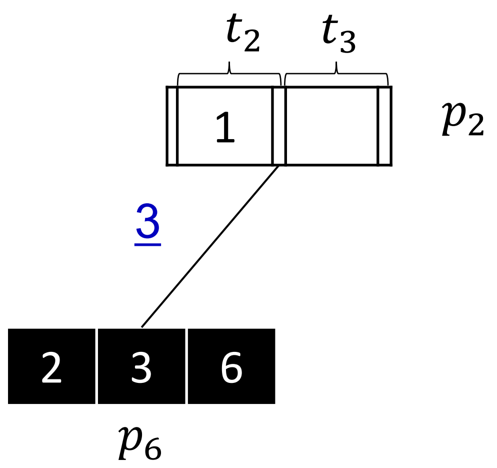
- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 2: updating weights after leaf insertion → both undercounting and overcounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:



Challenge 2: consistent weights for sampling (cont'd)

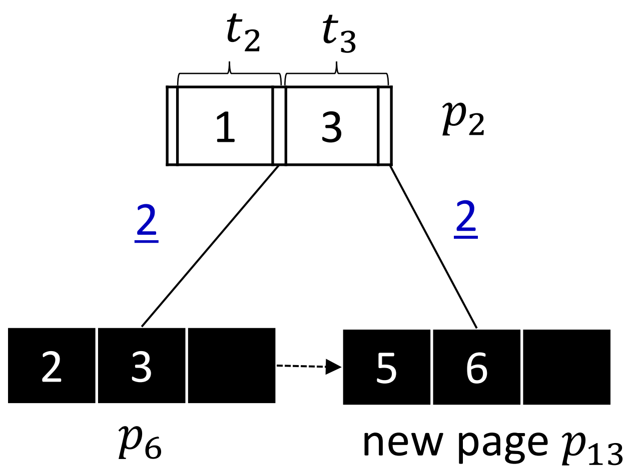
- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 2: updating weights after leaf insertion → both undercounting and overcounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:
(1) T_2 splits p_6 and inserts t'



Challenge 2: consistent weights for sampling (cont'd)

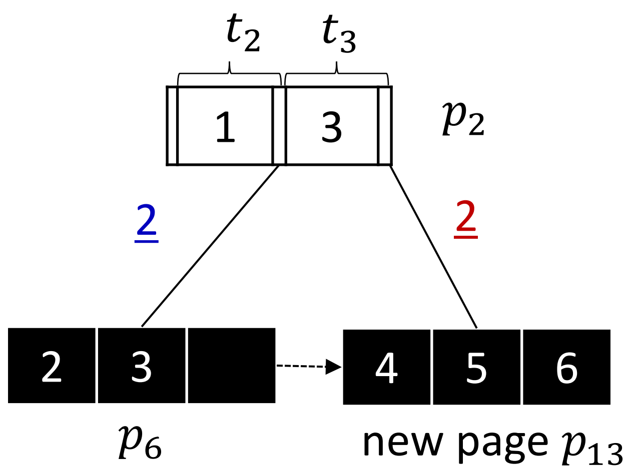
- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 2: updating weights after leaf insertion → both undercounting and overcounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

Steps:
(1) T_2 splits p_6 and inserts t'
(2) T_1 inserts t



Challenge 2: consistent weights for sampling (cont'd)

- Consistent weights needed for sampling purpose

Definition 1: An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 2: updating weights after leaf insertion → both undercounting and overcounting

T_1: insert $k_t = 4$

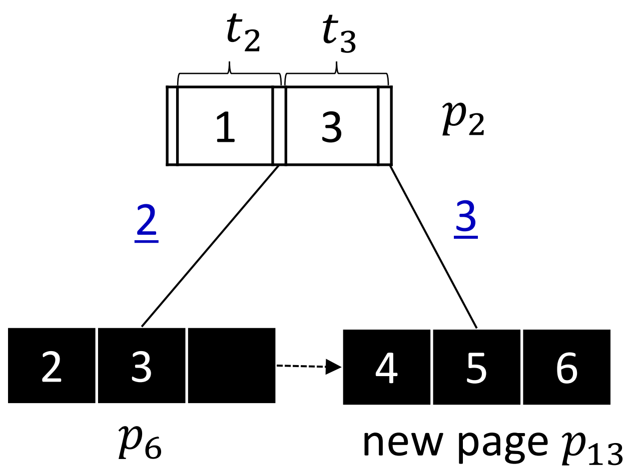
T_2: insert $k_{t'} = 5$

Steps:

(1) T_2 splits p_6 and inserts t'

(2) T_1 inserts t

(3) T_1 increments $\widetilde{w}_{t_3}$



Challenge 2: consistent weights for sampling (cont'd)

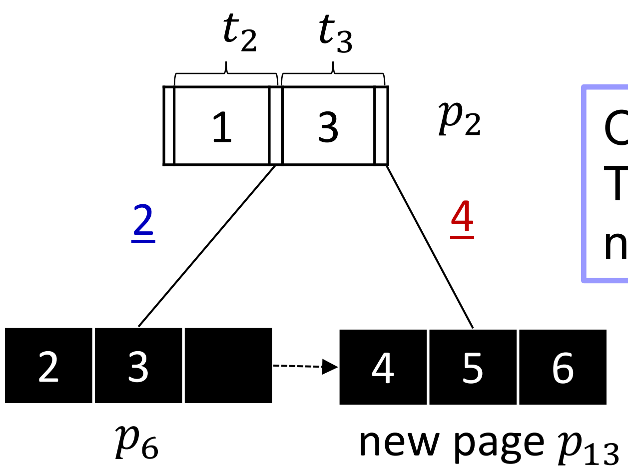
- Consistent weights needed for sampling purpose

> **Definition 1:** An aggregate B-tree T is said to be consistent for sampling purpose if and only if for any index tuple $t \in T$: $\widetilde{w}_t \geq \sum_{t' \in c_t} \widetilde{w}_{t'}$.

- Scenario 2: updating weights after leaf insertion → both undercounting and overcounting

T_1: insert $k_t = 4$
T_2: insert $k_{t'} = 5$

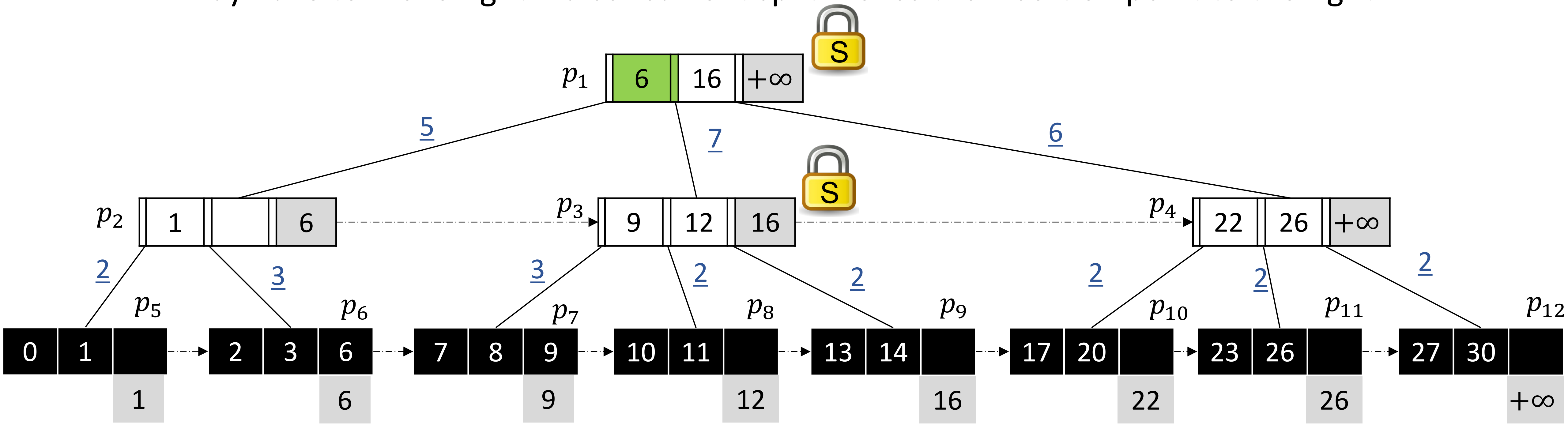
Steps:
(1) T_2 splits p_6 and inserts t'
(2) T_1 inserts t
(3) T_1 increments $\widetilde{w}_{t_3}$
(4) T_2 increments $\widetilde{w}_{t_3}$



Overcount is not Okay either:
The weight of the index tuple pointing to p_2 is now smaller than the sum in p_2.

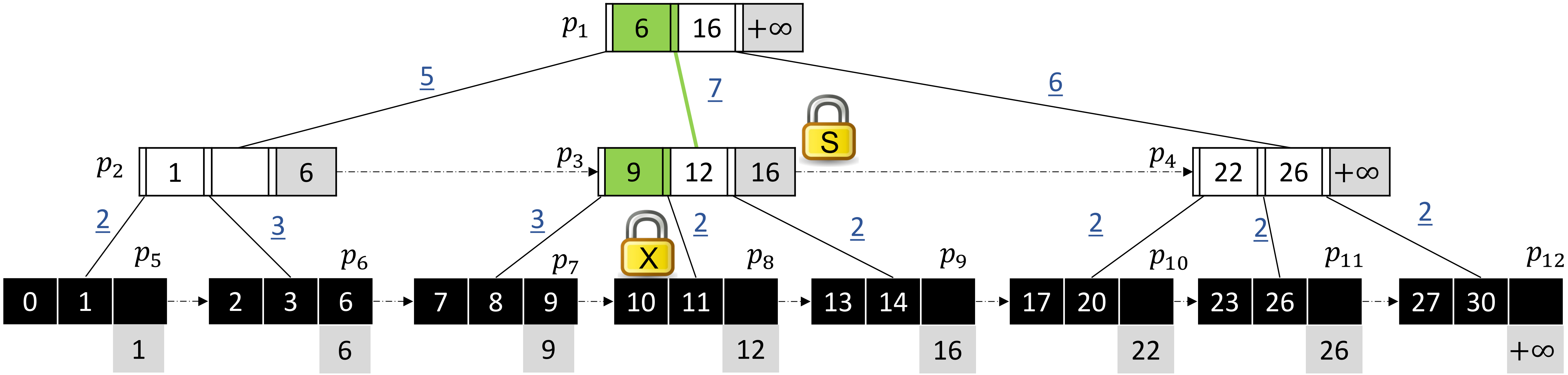
Insertion in AB-tree

- Running example: inserting $k_t = 12$
- First descent: search for insertion location
  - No latch is held across pages during search
  - S-latch the internal pages; X-latch the leaf page
  - May have to move right if a concurrent split moves the insertion point to the right



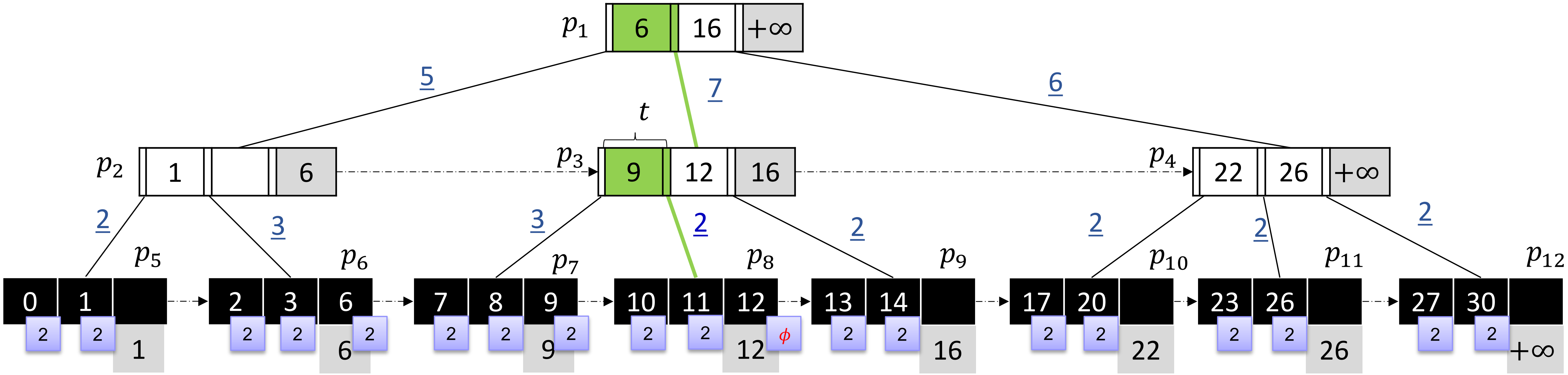
Insertion in AB-tree: first descent

- Running example: inserting $k_t = 12$
- First descent: search for insertion location
  - No latch is held across pages during search
  - S-latch the internal pages; X-latch the leaf page
  - May have to move right if a concurrent split moves the insertion point to the right



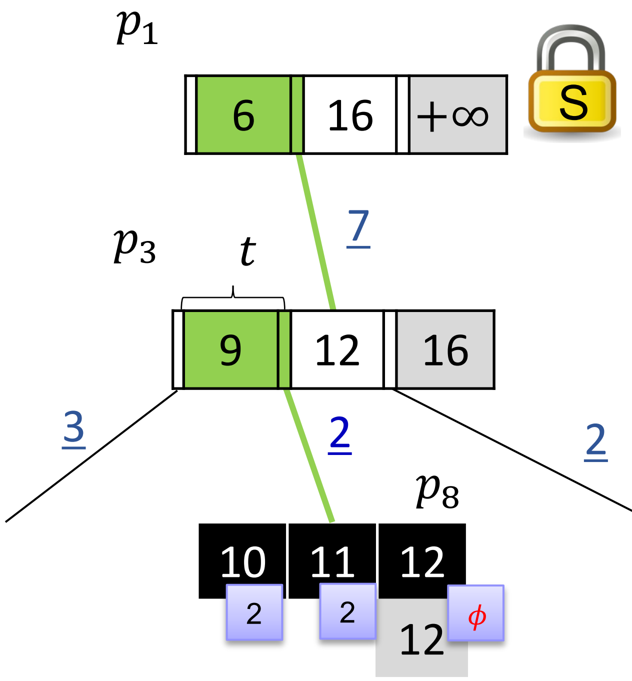
Insertion in AB-tree: second descent

- Running example: inserting $k_t = 12$
- Second descent: updating the aggregate weights
  - Use the same search key to re-descend the tree
  - S-latch pages.
  - Atomically update $\widetilde{w}$ on the internal pages and $xmin$ on the leaf pages.



Insertion in AB-tree: second descent

- Running example: inserting $k_t = 12$
- Second descent: updating the aggregate weights
  - Use the same search key to re-descend the tree
  - S-lock pages.
  - Atomically update $\widetilde{w}$ on the internal pages and $xmin$ on the leaf pages.

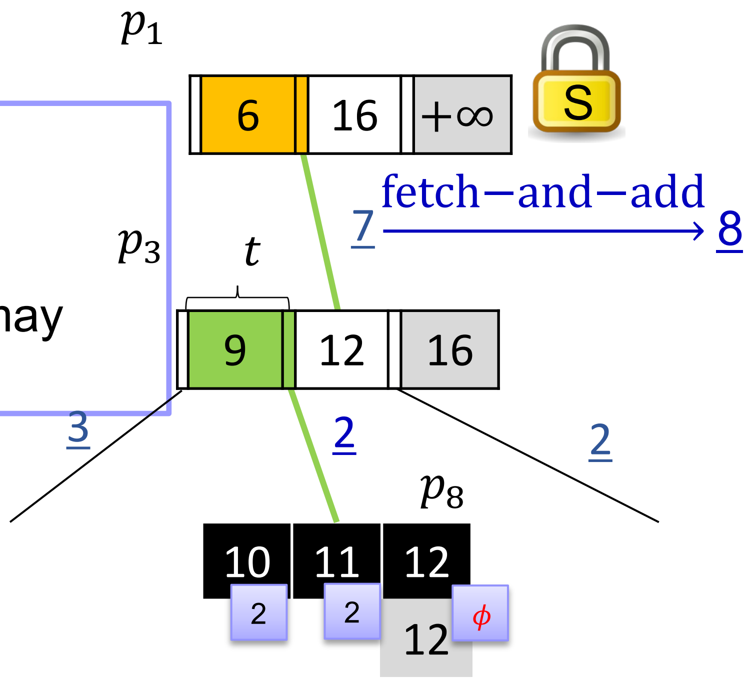


Insertion in AB-tree: second descent

- Running example: inserting $k_t = 12$
- Second descent: updating the aggregate weights
  - Use the same search key to re-descend the tree
  - S-lock pages.
  - Atomically update $\widetilde{w}$ on the internal pages and $xmin$ on the leaf pages.

Wrong!

A concurrent structural modification operation (SMO) on the child page may undo the increment.

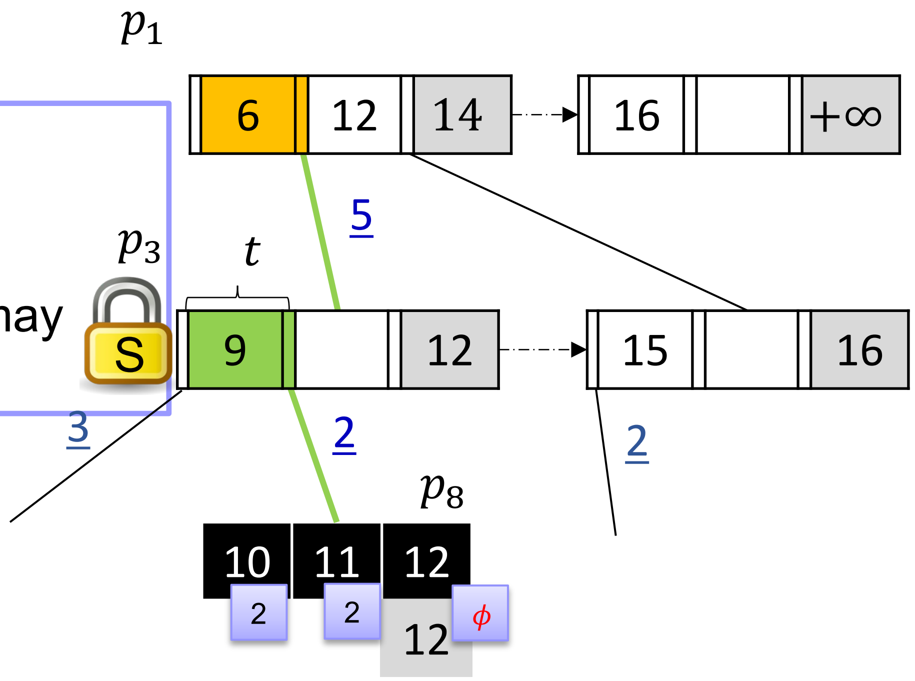


Insertion in AB-tree: second descent

- Running example: inserting $k_t = 12$
- Second descent: updating the aggregate weights
  - Use the same search key to re-descend the tree
  - S-lock pages.
  - Atomically update $\widetilde{w}$ on the internal pages and $xmin$ on the leaf pages.

Wrong!

A concurrent structural modification operation (SMO) on the child page may undo the increment.

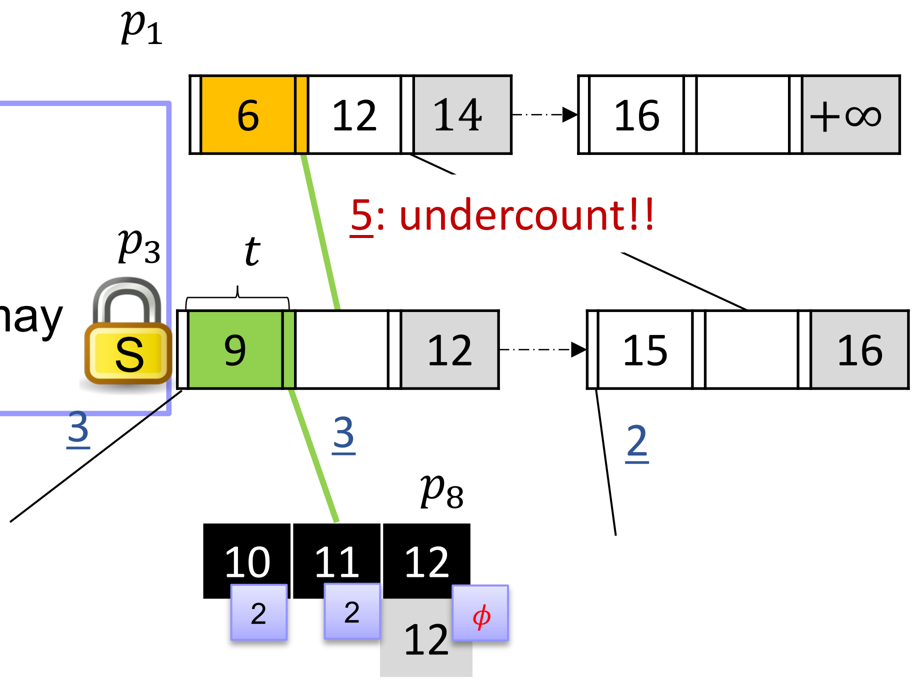


Insertion in AB-tree: second descent

- Running example: inserting $k_t = 12$
- Second descent: updating the aggregate weights
  - Use the same search key to re-descend the tree
  - S-lock pages.
  - Atomically update $\widetilde{w}$ on the internal pages and $xmin$ on the leaf pages.

Wrong!

A concurrent structural modification operation (SMO) on the child page may undo the increment.



Insertion in AB-tree: second descent

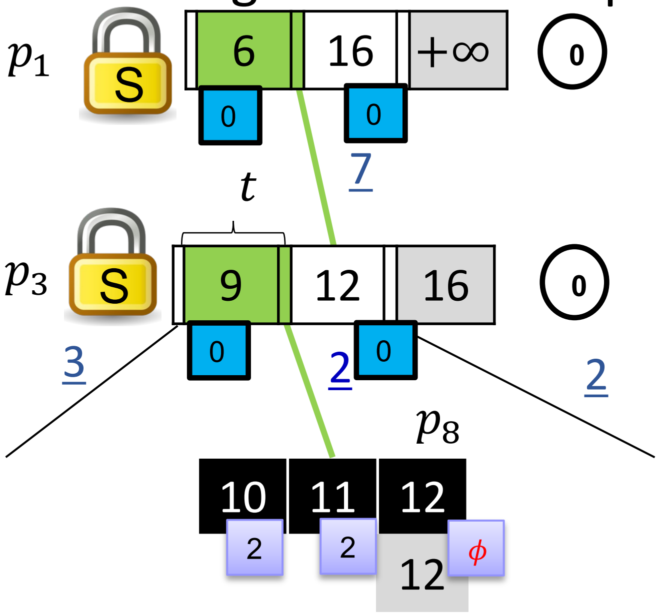
- Running example: inserting $k_t = 12$
- Update the weight only when holding an S-latch on the correct child page as well
  - B-link tree obtains latches from bottom to up during split → need deadlock avoidance
  - Rewind to some parent page if there're concurrent splits that
    - undo the increments in the parent/ancestor pages
    - or moves the search point to the right of the child page

Detect concurrent SMO

SID: 16-bit SMO ID for internal page
+= 1 for any SMO on some children

RID: 16-bit Recompute ID for index tuple
+= 1 for a split on its child page

No WAL on SID or RID – only concurrent threads are interested



Insertion in AB-tree: second descent

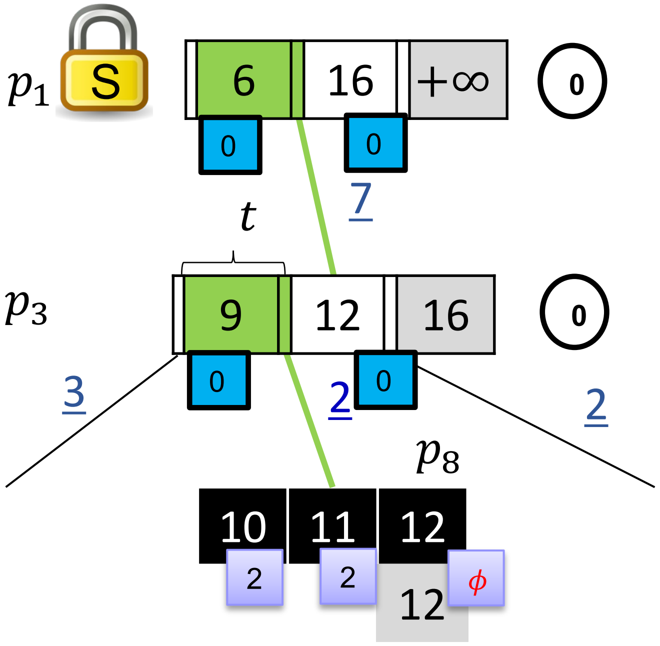
- Running example: inserting $k_t = 12$
- Update the weight only when holding an S-latch on the correct child page as well

Detect concurrent SMO

SID: 16-bit SMO ID for internal page
+= 1 for any SMO on some children

RID: 16-bit Recompute ID for index tuple
+= 1 for a split on its child page

No WAL on SID or RID – only concurrent threads are interested



Insertion in AB-tree: second descent

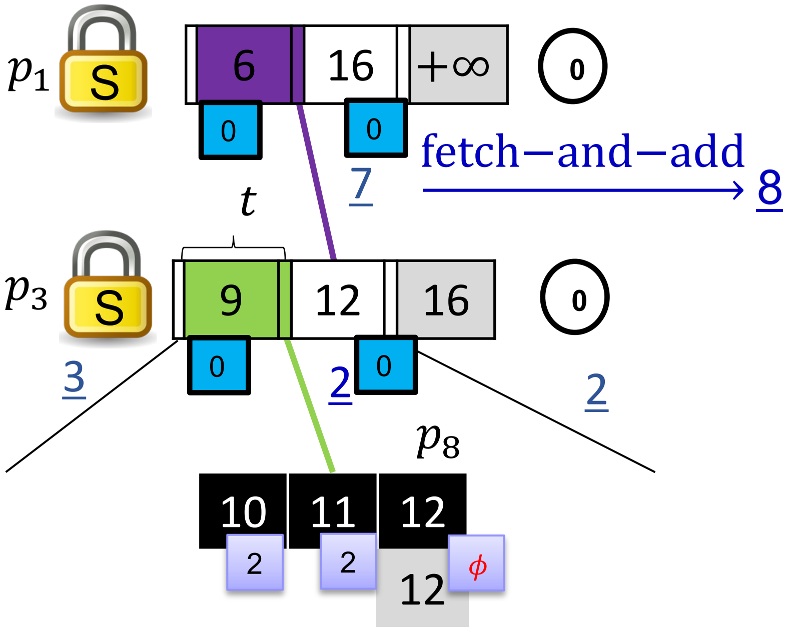
- Running example: inserting $k_t = 12$
- Update the weight only when holding an S-latch on the correct child page as well
- Case 1: SID_{p_1} does not change → safe to perform the update

Detect concurrent SMO

SID: 16-bit SMO ID for internal page
+= 1 for any SMO on some children

RID: 16-bit Recompute ID for index tuple
+= 1 for a split on its child page

No WAL on SID or RID – only concurrent threads are interested



Insertion in AB-tree: second descent

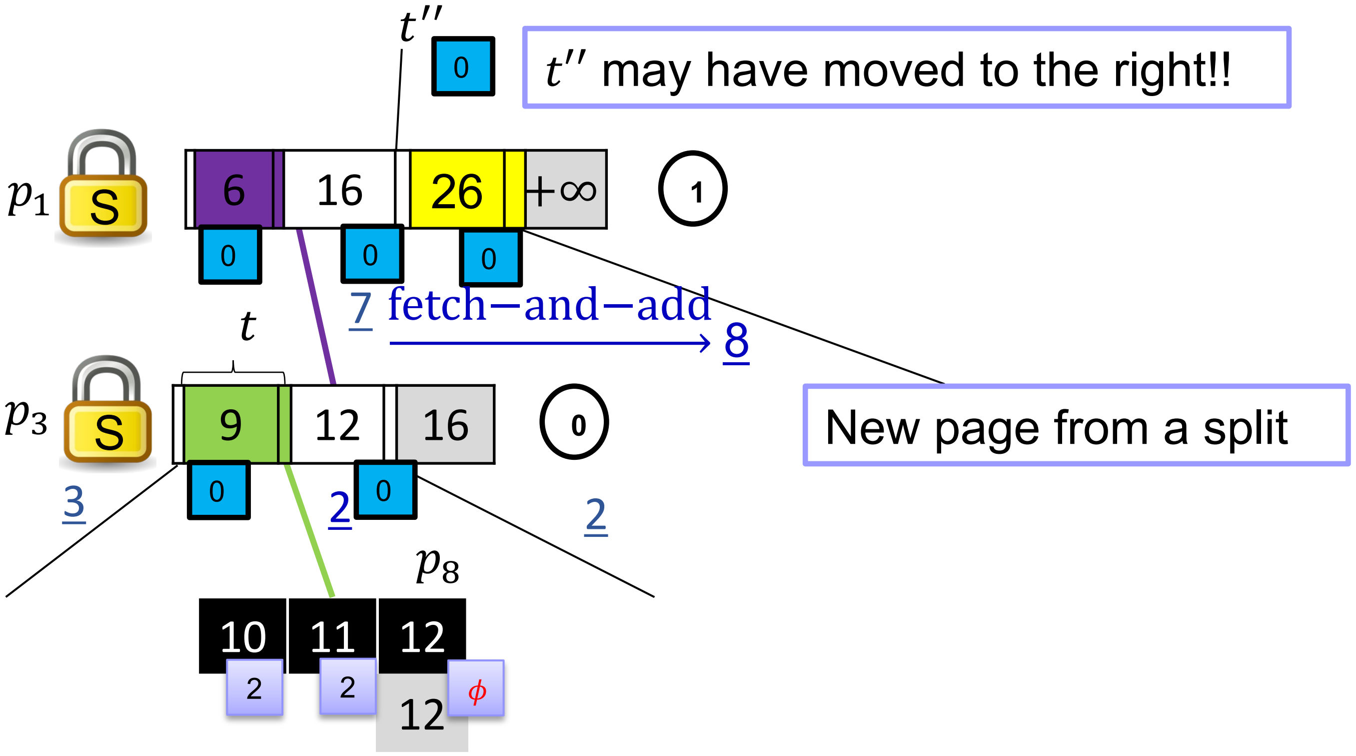
- Running example: inserting $k_t = 12$
- Update the weight only when holding an S-latch on the correct child page as well
- Case 2: SID_{p_1} changes but p_3 still has the search point, and
  - The SID of the parent page *or* the RID of the index tuple t'' that points to p_1 did not change → safe to update

Detect concurrent SMO

SID: 16-bit SMO ID for internal page
+= 1 for any SMO on some children

RID: 16-bit Recompute ID for index tuple
+= 1 for a split on its child page

No WAL on SID or RID – only concurrent threads are interested



Insertion in AB-tree: second descent

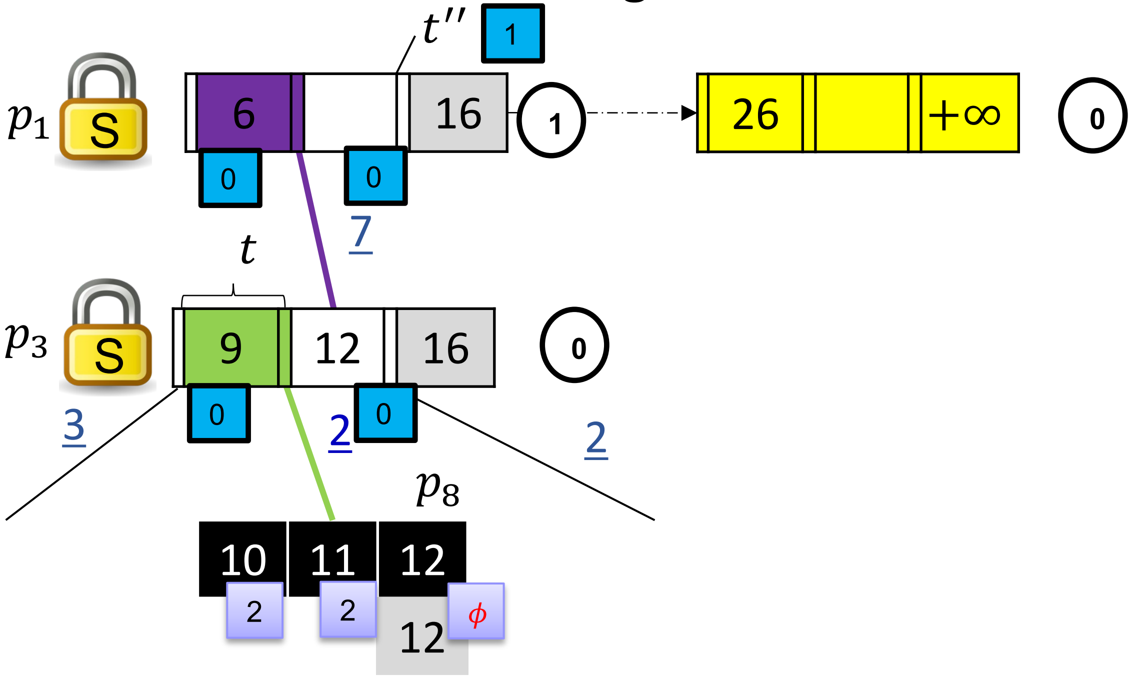
- Running example: inserting $k_t = 12$
- Update the weight only when holding an S-latch on the correct child page as well
- Case 3: SID_{p_1} changes and any of the following happens
  - p_3 does not have the search point *or* p_1 no longer contains a link to p_3
  - the SID of the parent page *and* the RID of t'' both change
  - root splits → must rewind

Detect concurrent SMO

SID: 16-bit SMO ID for internal page
+= 1 for any SMO on some children

RID: 16-bit Recompute ID for index tuple
+= 1 for a split on its child page

No WAL on SID or RID – only concurrent threads are interested



Insertion in AB-tree: second descent

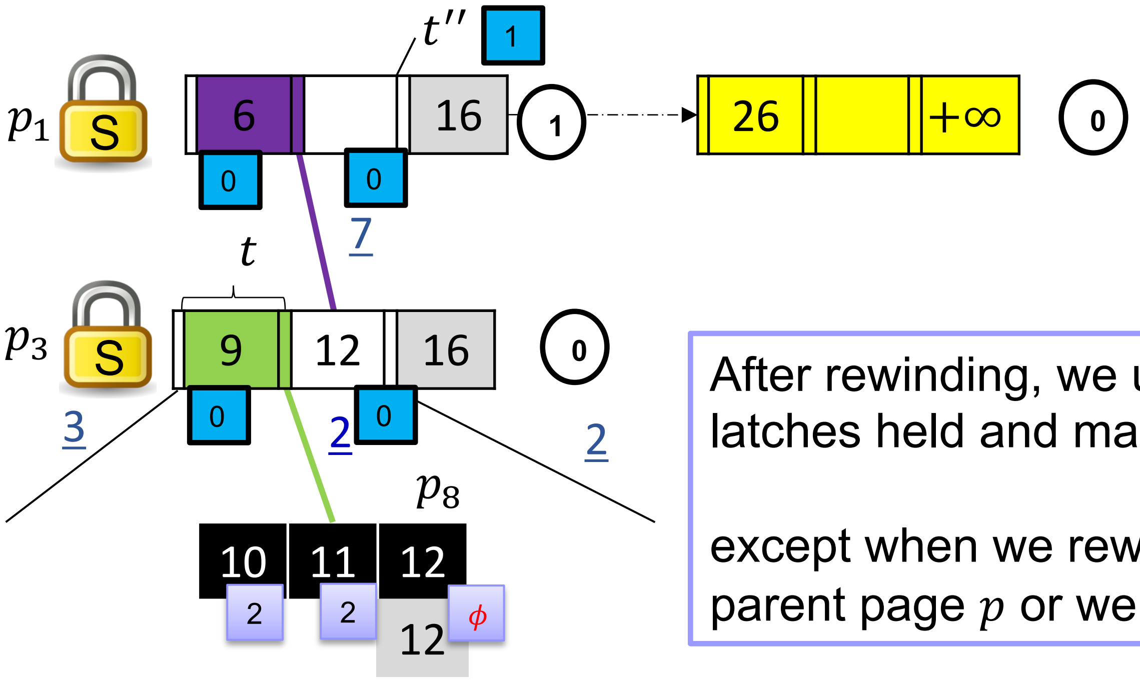
- Running example: inserting $k_t = 12$
- Update the weight only when holding an S-latch on the correct child page as well
- Rewind: find some page p on a higher level, such that
  - the SID of its parent page p'' does not change
  - or the RID of the index tuple that points to p does not change
- Or we restart from the root

Detect concurrent SMO

SID: 16-bit SMO ID for internal page
+= 1 for any SMO on some children

RID: 16-bit Recompute ID for index tuple
+= 1 for a split on its child page

No WAL on SID or RID – only concurrent threads are interested

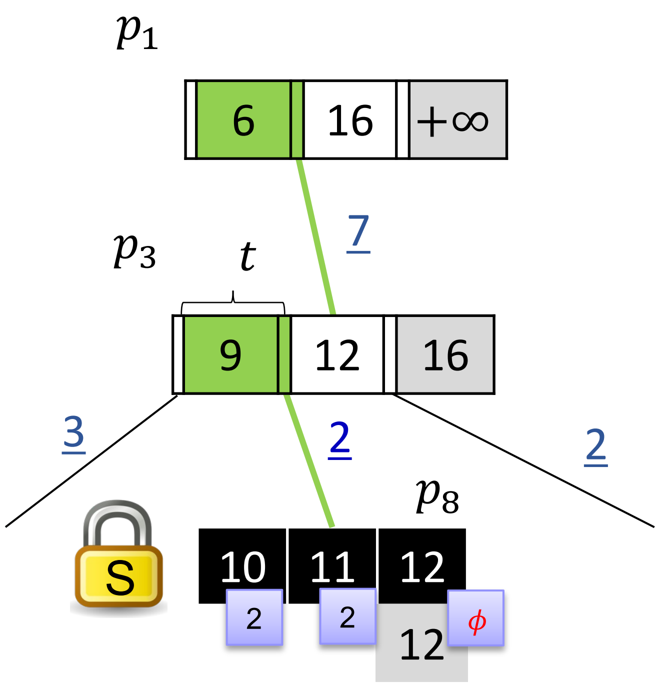


After rewinding, we usually have two latches held and may do update,

except when we rewind to the original parent page p or we restart from root.

Insertion in AB-tree: second descent

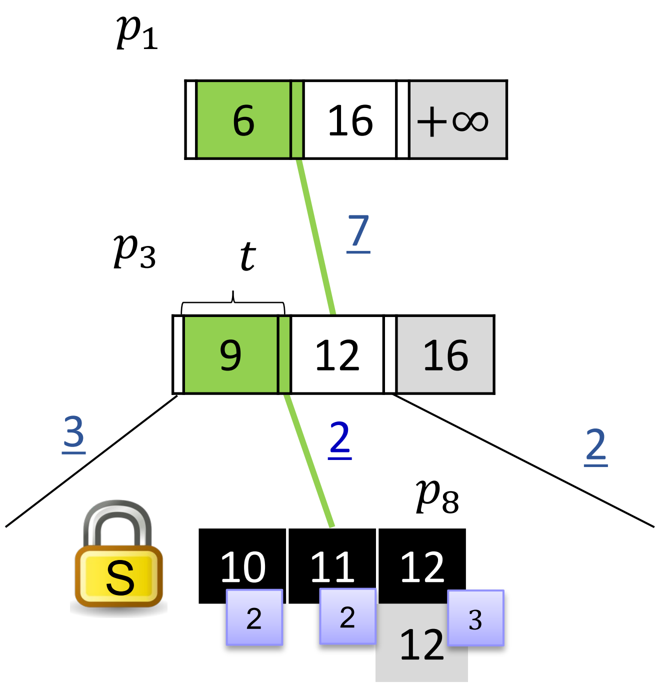
- Running example: inserting $k_t = 12$
- When we reach a leaf page (e.g., p_8)



Insertion in AB-tree: second descent

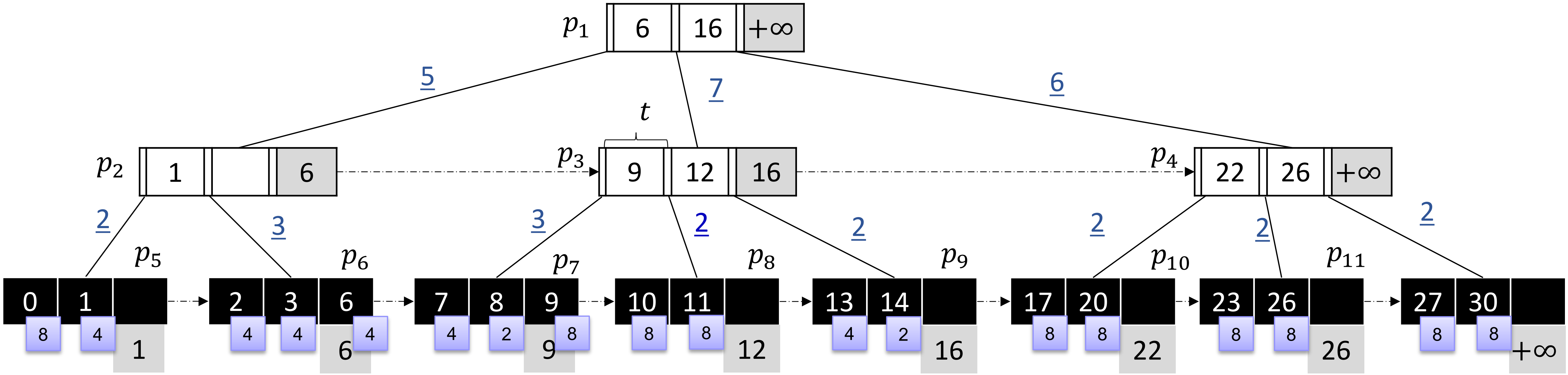
- Running example: inserting $k_t = 12$
- When we reach a leaf page (e.g., p_8)
  - Use Compare-and-Swap (CAS) to update $xmin$ to the running transaction ID

The insertion algorithm maintains an AB-tree that is always consistent for sampling purpose at all times and can correctly insert a tuple and update the aggregated weights.



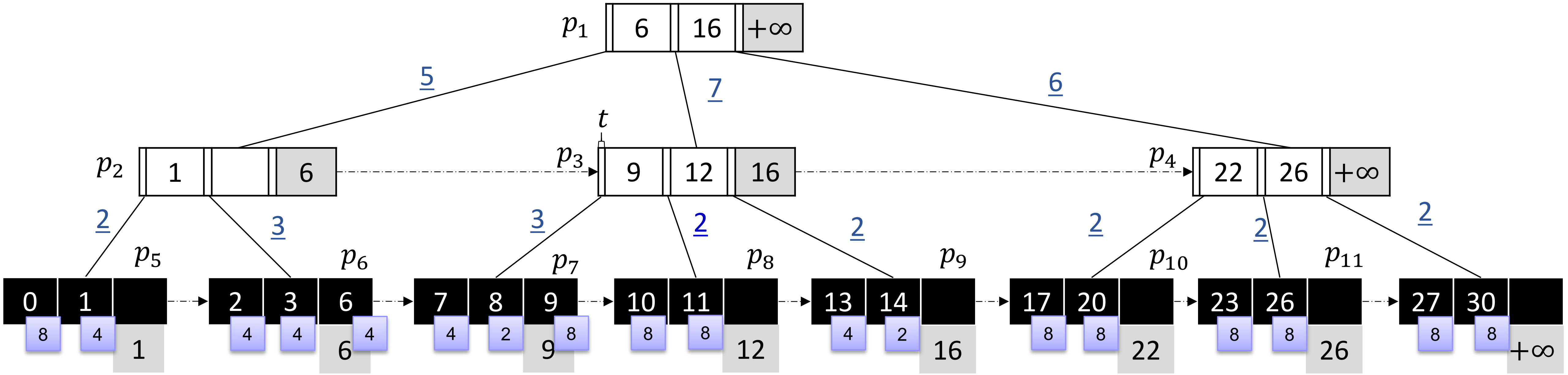
Multi-version weight store

- Live version bloat
  - Many new tuples in the index invisible to an old snapshot



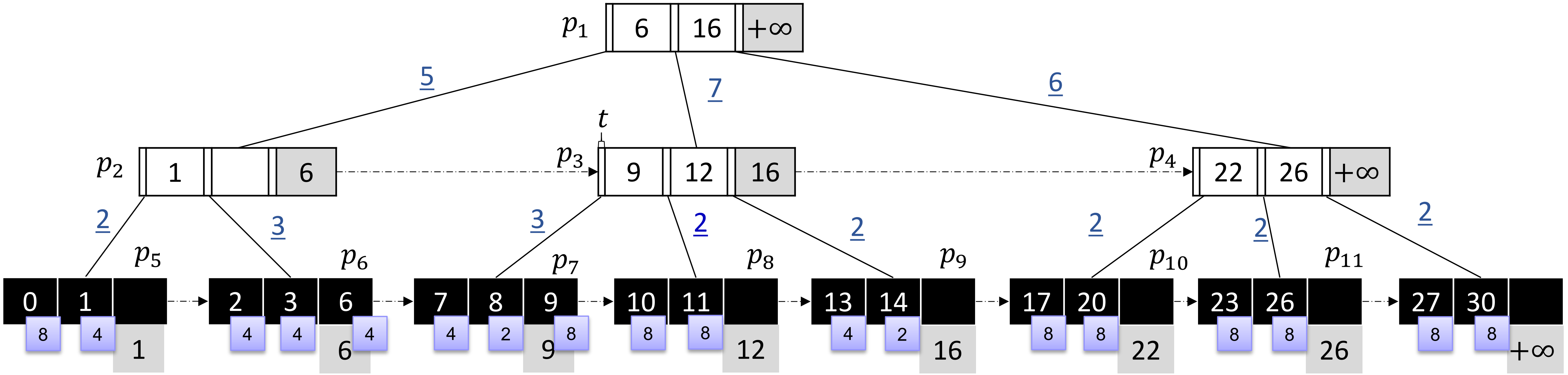
Multi-version weight store

- Based on PostgreSQL MVCC model
  - Snapshot S "$xmin_S$:$xmax_S$:xip_list_S"
    - a set of concurrent transaction ID in $[xmin_S, xmax_S)$, union all transactions >= $xmax_S$
  - RW transactions are assigned transaction IDs (xid)
  - Each tuple has a $xmin$ (creating transaction ID), and a $xmax$ (deleting transaction ID)
  - A tuple t is visible ↔ $xmin_t \notin S \wedge xmin_t$ commits ∧ ($xmax_t \in S$ or aborts or is invalid)



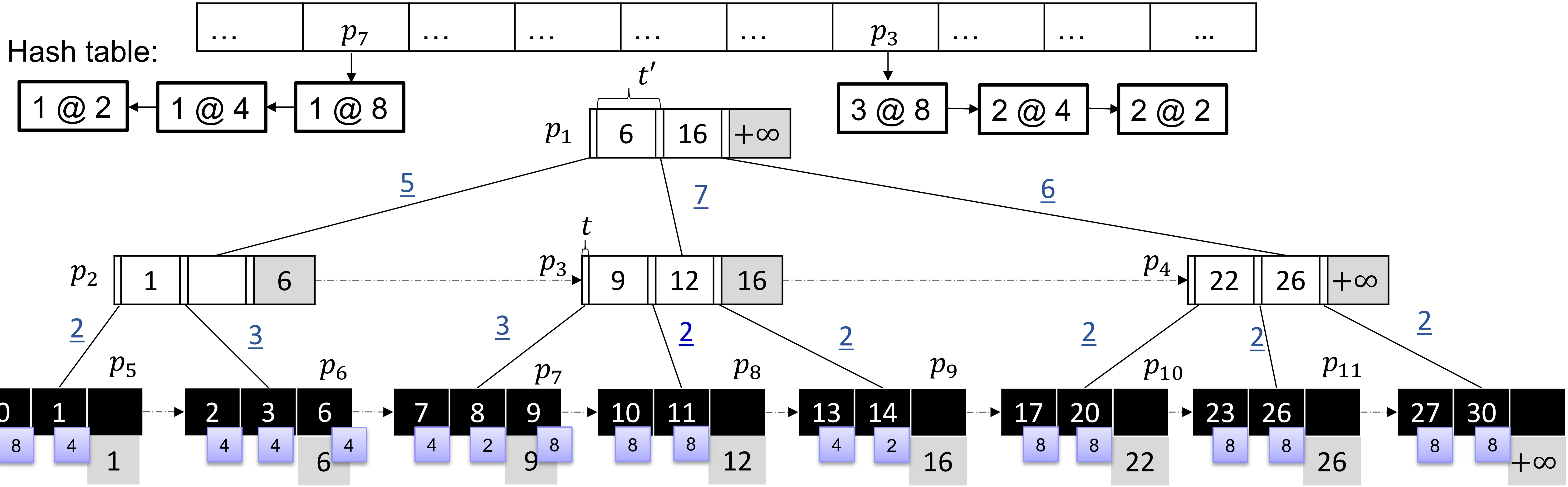
Multi-version weight store

- Based on PostgreSQL MVCC model
  - Snapshot S "$xmin_S$:$xmax_S$:xip_list_S"
    - a set of concurrent transaction ID in $[xmin_S, xmax_S)$, union all transactions >= $xmax_S$
  - RW transactions are assigned transaction IDs (xid)
  - Each tuple has a $xmin$ (creating transaction ID), and a $xmax$ (deleting transaction ID)
  - A tuple t is visible $\rightarrow xmin_t \notin S$, i.e., $xmin_t < xmax_S \wedge xmin_t \notin xip\_list_S$



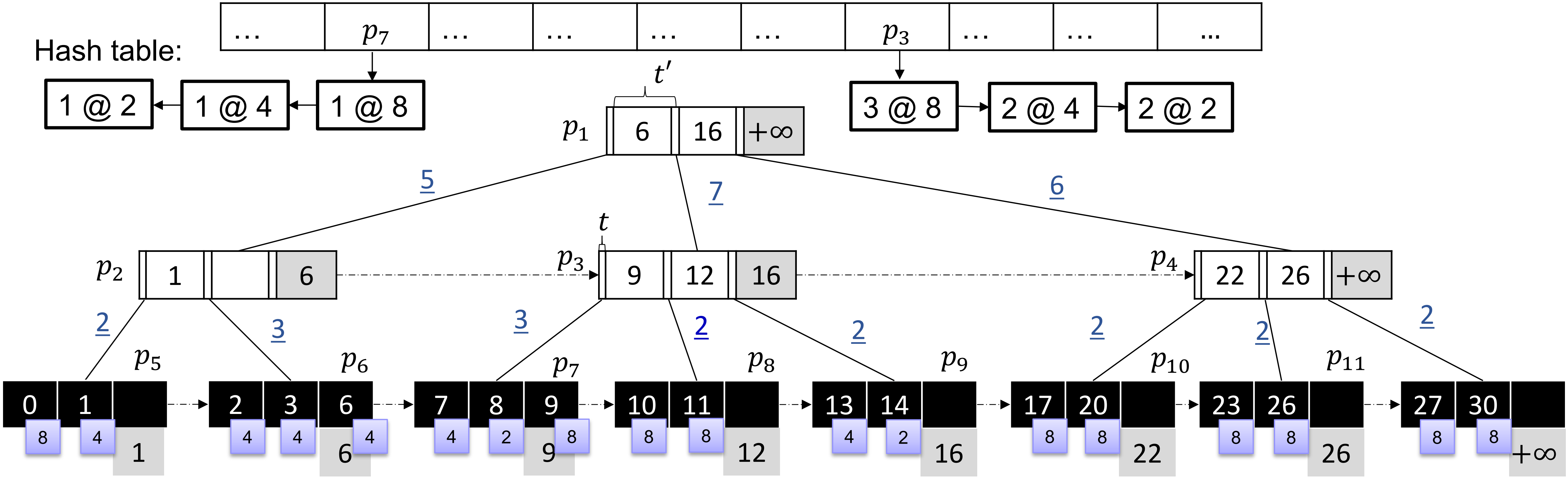
Multi-version weight store

- Solving live version bloat using the necessary condition for visibility:
  - $xmin_t < xmax_s \wedge xmin_t \notin xip_{list_s}$
- Only include leaf tuples whose $xmin$ satisfies the above condition in sampling
  - Maintain delta weights at different transaction IDs *in memory* (No persistence/WAL needed)



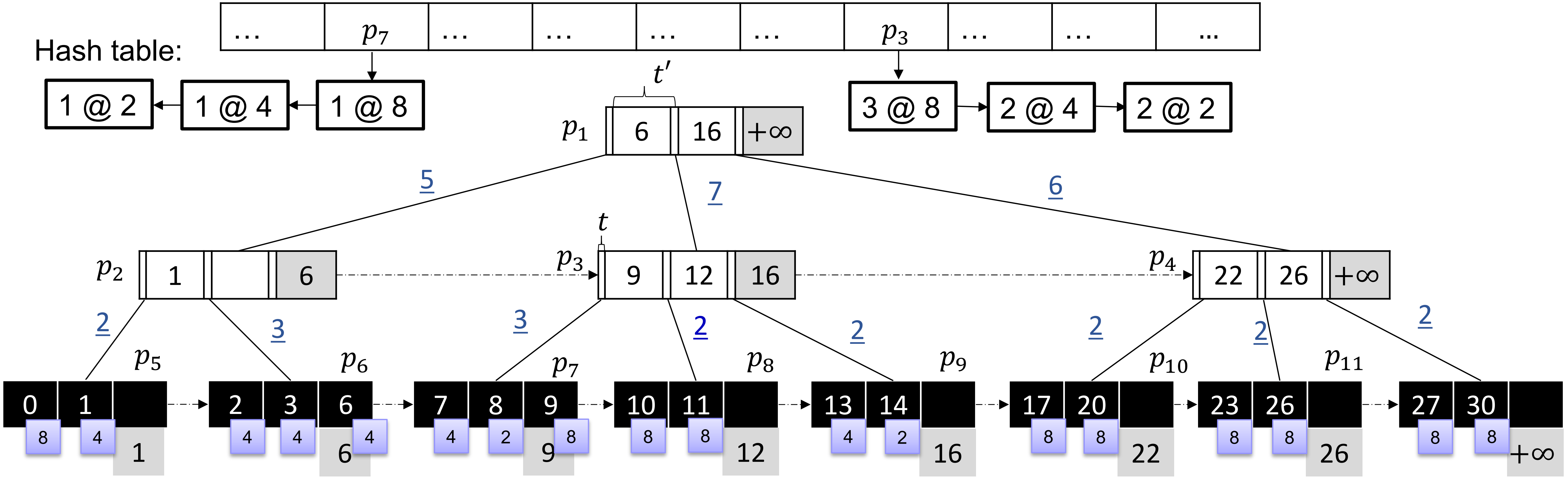
Multi-version weight store

- Say we have a sampling thread at snapshot $S = 2{:}\,2{:}\,\{\,\}$
  - Only committed tuples with $xmin \leq 2$ may be visible
  - $\tilde{w}^S_{t'} = 7 - 3 - 2 = 2$; $\tilde{w}^S_t = 3 - 1 - 1 = 1$



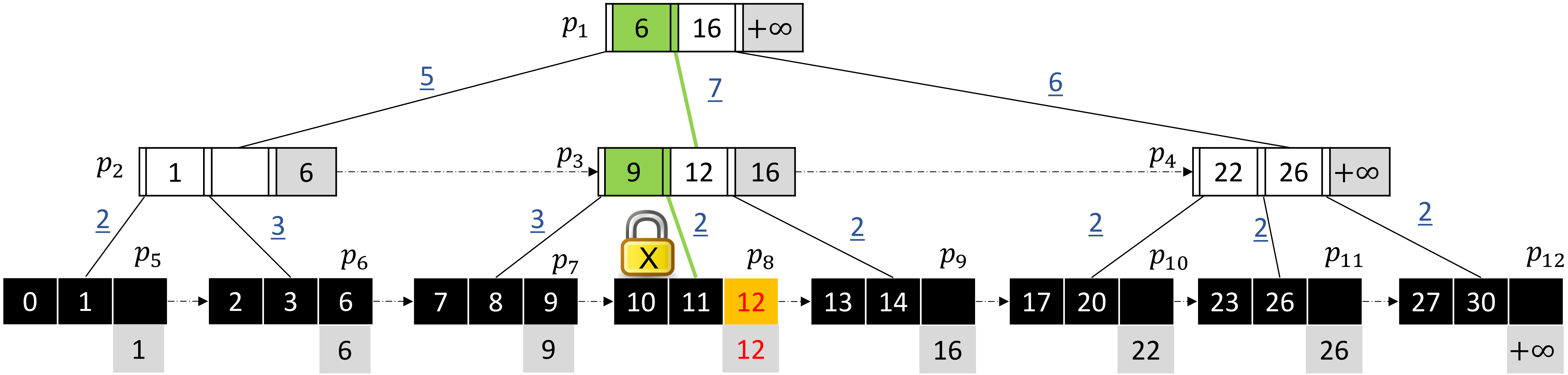
Multi-version weight store

- GlobalXmin – smallest $xmin$ of any active snapshot in the system
  - Any version < GlobalXmin may be discarded
  - Background GC thread scans the chains periodically



Insertion in AB-tree: first descent

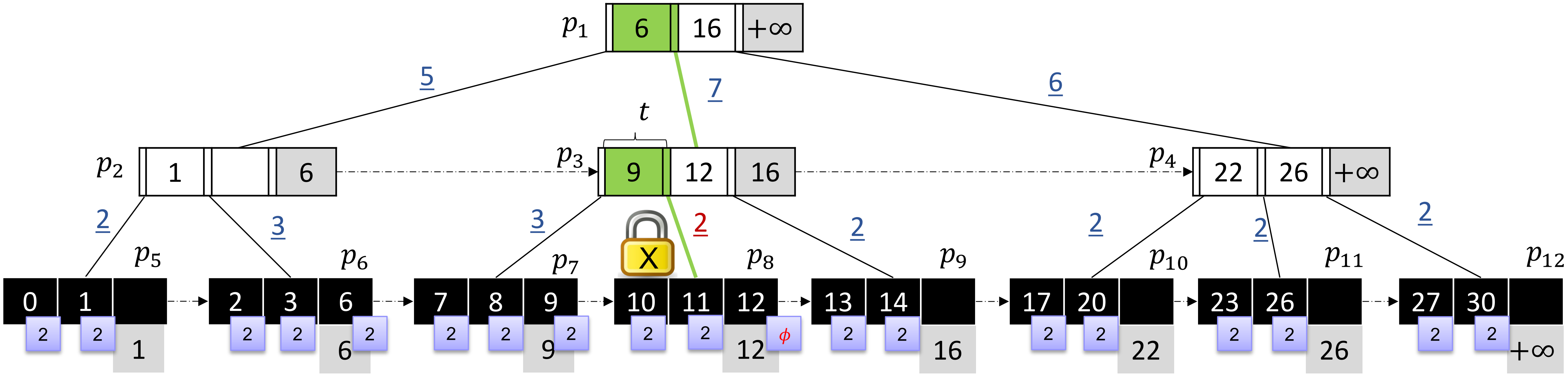
- Running example: inserting $k_t = 12$
- First descent: search for insertion location
  - No latch is held across pages during search
  - S-latch the internal pages; X-latch the leaf page



Insertion in AB-tree: first descent (cont'd)

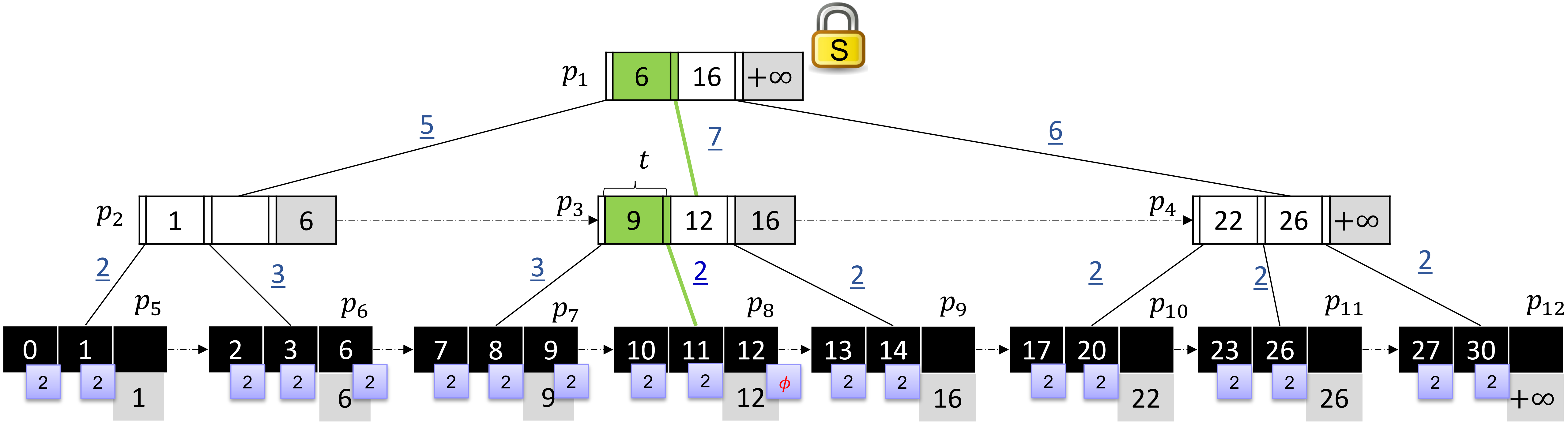
- Running example: inserting $k_t = 12$
- Inconsistent for sampling: $\tilde{w}_t = 2 < 3 = \sum_{t' \in p_8} \tilde{w}_{t'}$
  - Attach the creating transaction ID $xmin$ to leaf tuples
  - Newly inserted leaf tuples have invalid $xmin = \phi$
  - Leaf tuples with $xmin = \phi$ may not be counted or sampled

Valid $xmin$ are used in multi-version weight store later.



Insertion in AB-tree: second descent

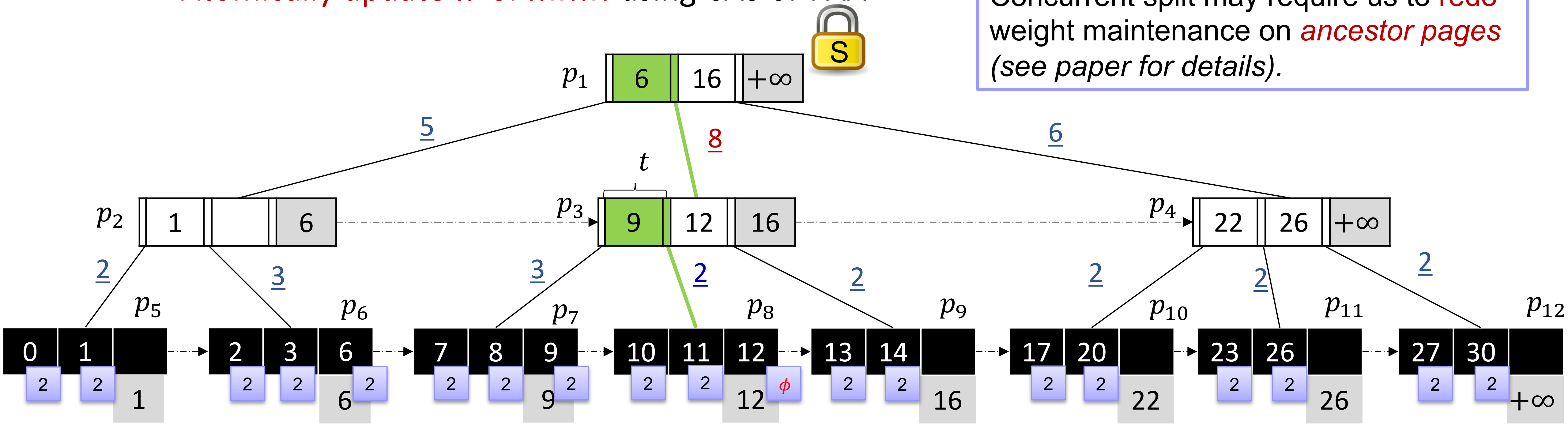
- Running example: inserting $k_t = 12$
- Second descent: updating the aggregate weights
  - Use the same search key to re-descend the tree
  - S-latch pages -- ensures index entry not concurrently moved



Insertion in AB-tree: second descent

- Running example: inserting $k_t = 12$
- Second descent: updating the aggregate weights
  - Use the same search key to re-descend the tree
  - S-latch pages -- ensures index entry not concurrently moved
  - Atomically update $\widetilde{w}$ or $xmin$ using CAS or FAA

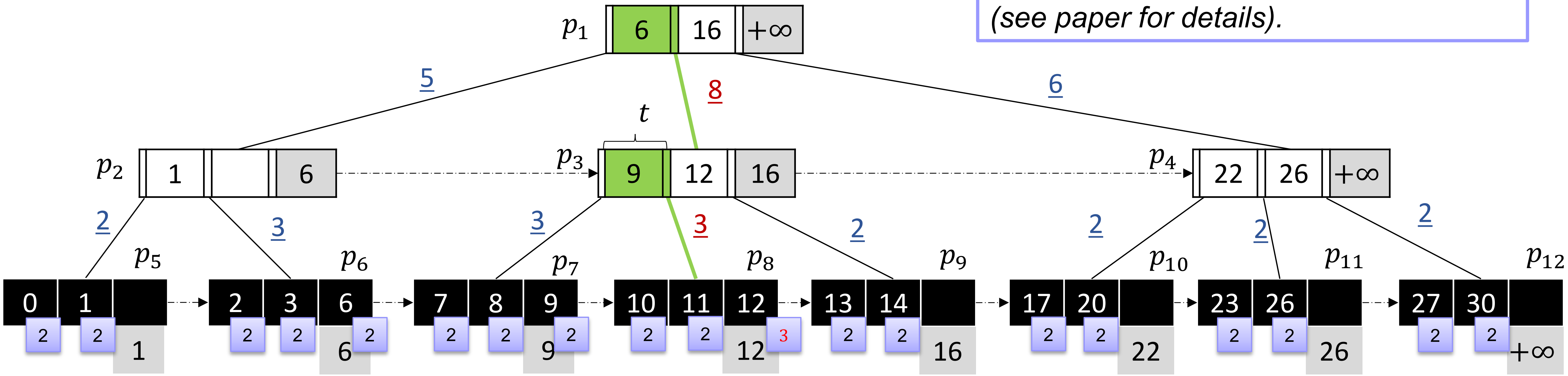
Concurrent split may require us to redo weight maintenance on *ancestor pages* *(see paper for details).*



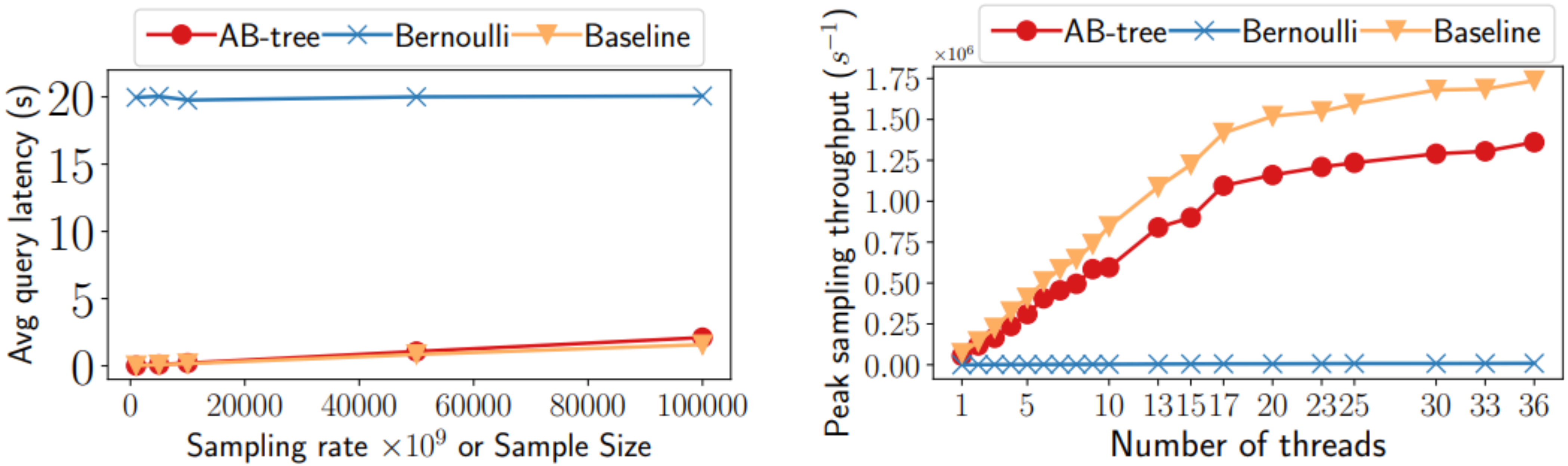
Insertion in AB-tree: second descent

- Running example: inserting $k_t = 12$
- Second descent: updating the aggregate weights
  - Use the same search key to re-descend the tree
  - S-latch pages -- ensures index entry not concurrently moved
  - Atomically update $\widetilde{w}$ or $xmin$ using CAS or FAA

Concurrent split may require us to redo weight maintenance on *ancestor pages* *(see paper for details).*



Read-only workload



(a) Average query latency (b) Peak sampling throughput

Figure 9: Sampling with varying number of threads

TPC-H

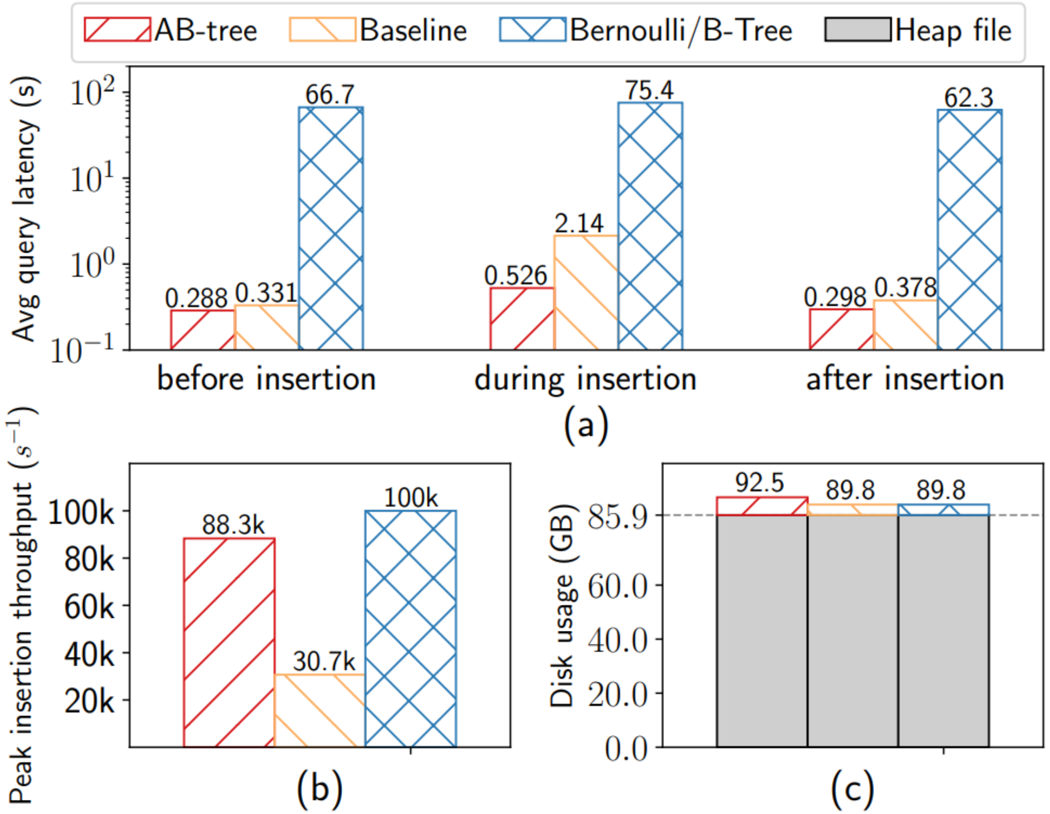


Figure 16: Mixed workload on TPC-H lineitem (SF = 100)